Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

The History of the ALGOL Effort

by

HT de Beer

supervisors:

C. Hemerik
L.M.M. Royakkers

Eindhoven, August 2006

Abstract

This report studies the importancy of the ALGOL effort for computer science. Therefore the history of the ALGOL effort is told, starting with the computational context of the 1950s when the ALGOL effort was initiated. Second, the development from IAL to ALGOL 60 and the role the BNF played in this development are discussed. Third, the translation of ALGOL 60 and the establishment of the scientific field of translator writing are treated. Finally, the period of ALGOL 60 maintenance and the subsequent period of creating a successor to ALGOL 60 are described.

Preface

This history on the ALGOL effort was written as a Master thesis in Computer Science and Engineering (Technische Informatica) at the Eindhoven University of Technology, the Netherlands.

Contents

0 Introduction
On Sources, Perspective and the ALGOL Effort

What was the importance of the ALGOL effort for computer science? • On the sources used to answer this question • My own perspective on ALGOL and the ALGOL effort • And the meaning of the term "ALGOL effort"

On May 20th, 2006, Peter Naur received the 2005 ACM Turing Award. This award is often regarded as the "Nobel Prize" in Computer Science. Naur was rewarded for his 'pioneering work on defining the Algol 60 programming language. (...) [He] was editor in 1960 of the hugely influential "Report on the Algorithmic Language Algol 60." He is recognised for the report's elegance, uniformity and coherence, and credited as an important contributor to the language's power and simplicity.'[0]

Forty-five years after the publication of the ALGOL 60 report, it is still regarded as one of the most influential papers in computer science. Peter Naur was not the only Turing Award winner related to the ALGOL effort: A.J. Perlis, J. McCarthy, E.W. Dijkstra, J. Backus, R.W. Floyd, C.A.R. Hoare, and N. Wirth were all awarded before him.[1] It may be not for their work in the ALGOL effort per sé, but they were all part of the effort, either as a member of the committee defining ALGOL 58 or ALGOL 60, as an implementor of ALGOL 60, as a contributor to the theory of formal languages stimulated by ALGOL 60, or as a member of the committee working on a new ALGOL in the 1960s.

The awarding of these computer scientists indicates that the ALGOL effort was an important part of the development of computer science itself. This importance is often stated without elaborating exactly why it was so important. ALGOL 60 was not the first programming language and it certainly was not the most used one. So, the question arises: *What was the importance of the ALGOL effort for computer science?*

To answer this question, the history of the ALGOL effort is told. First, in Chapter 1, the start of the effort is discussed and the central question in that chapter is: Why was there a need for an effort like the ALGOL effort?

This question will be answered by treating the computational context in which this effort was initiated; the algebraic programming languages from the 1950s are compared including the first language created by the ALGOL effort, ALGOL 58. Was this International Algebraic Language really better than these other algebraic languages?

Then, in Chapter 2, the development from ALGOL 58 leading to the publication of the *Report on the algorithmic language ALGOL 60*[2] in May 1960 is treated. In this period, the notation used to describe and define programming languages changed fundamentally. The resulting ALGOL 60 report would become *the* standard for defining programming languages. Both the interesting new features in the programming language ALGOL 60 and the reasons why this new notation was developed during the ALGOL effort, are discussed.

ALGOL 60 is often related to the development of formal language theory in computer science. This development was a result of the common effort to create translators for ALGOL 60. How did this development take shape, and, more importantly, what was the importance of this development for computer science? These questions are answered in Chapter 3.

After the publication of the ALGOL effort, a period of maintenance of the ALGOL 60 language was started. During this period, the responsibility for ALGOL was transferred to an international body: the famous Working Group 2.1 of IFIP. Later, this working group developed a new ALGOL resulting in dismay and the end of the ALGOL effort. This end of the ALGOL effort did not set a good example for computer science, and the question arises: what was the importance, if any, of the effort to create a new ALGOL? (See Chapter 4)

Finally, in Chapter 5, the conclusion of this research on the ALGOL effort is drawn by answering the main question: What was the the importance of the ALGOL effort for computer science? Before the history of the ALGOL effort is told some remarks on the sources used, on my perspective, and on the term “ALGOL effort” are made.

0.0 On the sources used

For this research both primary and secondary sources are used. The primary sources are almost all scientific publications. Most of them are articles from the *Communications of the ACM* and the *ALGOL Bulletin* related to the ALGOL effort. Using scientific publications as primary sources, in particular shorter articles, has a disadvantage: they were prepared to be published, and they give an academic view on developments in the ALGOL effort.

The *ALGOL Bulletin* is somewhat different from the other sources because it was edited by one person. The editor's perspective has influenced some of the content. Especially when reporting on institutional matters or when asking questions to the audience the information presented in the *ALGOL Bulletin* can give a coloured view on historical events in the ALGOL effort.

The secondary sources I have used consisted of historical works on computer science, on programming languages in general, on the ALGOL effort, and on some aspects of the ALGOL effort. Many of these works were written by people involved in the historical subject they wrote about. This gives the reader a special and direct view on the matter. Although the authors try to reflect and try to be as objective as possible, these histories are personal and, hence, subjective.

In this category, the proceedings of the two *ACM SIGPLAN conferences on History of Programming Languages* are great recourses on the history of the selected programming languages. The papers in these proceedings were written by the people behind those languages and are extremely useful to get an inside view on the early developments of those languages.

Furthermore, F.L. Bauer needs special attention because this German computer scientist has taken up writing history of computer science. Although he writes often about developments where he himself was actively involved in, he is the only source on the topic of early ALGOL translation and the developments leading to the initiation of the ALGOL effort. As a result, his historical works are very useful if they are used with care.

Finally, there are historical papers written on special occasions, like the ACM Turing Award Lectures and anniversaries, wherein the author looks back on his personal involvements in the ALGOL effort and computer science. These articles are interesting because they are able to place other sources in a historical context and give a more anecdotical view on the matter.

Lacking are sources on the use of the ALGOL languages and the reception of the ALGOL effort. The exception is the journal *Datamation* that, although explicitly American, gives a view on the computing community from a user's perspective in industry. It is, however, not enough to reconstruct a realistic view on the use and reception of ALGOL in Europe or in the USA.

0.1 My own perspective

Besides the perspective in the sources my own perspective is also important to know. I am a twenty-first century computer scientist with no involvement in ALGOL at all. Although the language was mentioned in some courses during my computer science education at the Eindhoven University of Technology, I do not know the language but by name.

The computer science education in Eindhoven focuses on structured programming and designing systems using formal methods. In other words, my perspective is that of an academic computer scientist rather than that of a programmer.

Besides studying computer science, I am also studying History at the University of Utrecht. I am experienced in doing historical research and will combine my knowledge of computer science with my experience in writing history to write this history of the ALGOL effort.

0.2 The ALGOL effort: a definition

Finally, I want to clarify the term "ALGOL effort". The ALGOL effort did, at first, not exist as an official institution. Later in the development of the ALGOL effort, it became more official and institutionalised, but at the start it was just an idea. Some computer scientists wanted to create a new universal programming language to write algorithms in. The effort that was undertaken to create this programming language, and everything related to it, is what I call the "ALGOL effort".

The ALGOL effort started somewhere in the mid-1950s. Where it ends is debatable: did it end in 1988 when the last ALGOL Bulletin was published, or did it end after the maintenance period of ALGOL 60? I have chosen the publication of the ALGOL 68 report as the end of the ALGOL effort. After the publication of this report in late 1969 the official developments on ALGOL did not end. The spirit of the ALGOL effort, however, was broken with the publication of a Minority Report.[3] More than half of the members of Working Group 2.1 responsible for the development of ALGOL 68 did not agree with the final result. With this disagreement the idea of an universal language was no longer part of the ALGOL effort.

0.3 Notes

[0]ACM Pressroom, 'Software Pioneer Peter Naur Wins ACM's Turing Award. Dane's Creative Genius Revolutionized Computer Language Design' (2006), ⟨URL: http://campus.acm.org/public/pressroom/press_releases/3_2006/turing_3_01_2006.cfm⟩

[1]ACM, 'A. M. Turing Award' (2006), ⟨URL: http://awards.acm.org/turing/⟩

[2]J. W. Backus et al., 'Report on the algorithmic language ALGOL 60', *Commun. ACM* 3:5 (1960)

[3]Dijkstra et al., 'Minority Report', *ALGOL Bull.* 31 (1970)

1 Creation
The Start of the ALGOL Effort

The start of the ALGOL effort • The need for a universal algorithmic language • The difference between the computing community in the USA and in Europe • Comparing early programming languages: IT, MATHMATIC, FORTRAN and IAL • And why IAL did have to be the universal algorithmic language

1.0 The start of the ALGOL effort

In October 1955, an international symposium on automatic computing was held in Darmstadt, Germany. In the 1950s, the term "automatic computing" referred to almost anything related to computing with a computer. During this symposium 'several speakers stressed the need for focusing attention on unification, that is, on *one universal, machine-independent algorithmic language* to be used by all, rather than to devise several such languages in competition.'[0]

To meet this need the Gesellschaft für Angewandte Mathematik und Mechanik (GAMM; association for applied mathematics and mechanics) set up a subcommittee for programming languages. This committee consisted of eight members: Bauer, Bottenbruch, Graeff, Läuchli, Paul, Penzlin, Rutishauser, and Samelson.[1] In 1957, it had almost completed its task; instead of creating yet another algorithmic language, however, it was decided to 'make an effort towards worldwide unification.'[2]

Indeed, the need for a universal algorithmic language was not felt only in Europe. Some computer user organisations in the USA, like SHARE, USE, and DUO also wanted one standard programming language for describing algorithms. In 1957, they asked the Association of Computing Machinery (ACM) to form a subcommittee to study such a language.[3] In June 1957, the committee was formed consisting of fifteen members from the industry, the universities, the users, and the federal government: Arden, Backus, Desilets, Evans, Goodman, Gorn, Huskey, Katz, McCarthy, Orden, Perlis, Rich, Rosen, Turanski, and Wegstein.[4]

Before this committee had held any meeting at all, a letter of the GAMM subcommittee for programming languages was send to the president of the ACM to propose a joint meeting to create jointly one international algebraic language[5] instead of two different but similar languages. The ACM agreed and three preparatory meetings were held to create a proposal for the language. On the third meeting, held at Philadelphia, 18 April 1958, F.L. Bauer presented the GAMM proposal to the ACM subcommittee.[6] Both proposals shared many features but the American proposal was more practical than the European counterpart that was more universal.[7]

The joint meeting was held at Zürich from May 27 to June 1, 1958, and was attended by F.L. Bauer, H. Bottenbruch, H. Rutishauser and K. Samelson from the GAMM subcommittee and by J. Backus, C. Katz, J. Perlis and J.H. Wegstein from the ACM counterpart. In the letter from the GAMM to the ACM where this meeting was proposed, it was stated that, 'we hope to expand the circle through representatives from England, Holland and Sweden'.[8] In the end, however, the meeting was attended by Americans, Germans and Swiss only. Nonetheless, international interest was growing, especially after the publication of the preliminary report on the new language.[9]

The discussions at the Zürich meeting were based on the two proposals for the new language and it was decided that:

1. 'The new language should be as close as possible to standard mathematical notation and be readable with little further explanation.
2. It should be possible to use it for the description of computing processes in publications.
3. The new language should be mechanically translatable into machine programs.'[10]

In addition, the language was to be machine independent. It was designed with no particular machine in mind.

With these decisions made some problems arose because of the difference between a publication language and a programming language to instruct computers. In addition, there was disagreement on the symbols to use. For example, the decimal point was problematic: the Americans were used to a period and the Europeans to a comma.[11] To solve all these representational issues Wegstein proposed to define the language on three different levels of representation: reference, hardware and publication. After this solution the joint meeting ended successfully with the publication of the *Preliminary Report: International Algebraic Language*.[12]

The Internal Algebraic Language (IAL), or ALGOL 58, as it was called later, was the result of an international effort to create a universal algorithmic language. This short history, however, does not explain why there was a need for a language like IAL. Or, for that matter, why creating a new language was preferred over using one of the other already existing algorithmic languages of that time. To answer these questions, the state of automatic computing in the 1950s is discussed in the next section. After that, the features of IAL are discussed and a comparison between IAL and some of the other algorithmic languages is made. While treating these subjects the differences between the American and European approaches towards automatic programming and programming languages in general are treated as well.

1.1 The need for a universal algorithmic language

1.1.0 The American field of computing

The need for a universal algorithmic language was stressed at the Darmstadt symposium in 1955. Almost two years earlier, in December 1953, John Backus proposed the FORTRAN project to his chief at IBM.[13] The goal of this project was to create a language to formulate 'a problem in terms of a mathematical notation and to produce automatically a high speed [IBM] 704 program for the solution of the problem.'[14] In other words, FORTRAN was to be an algorithmic language with a practically usable translator: the programs generated by this system should be as efficient as hand-coded programs written in machine code or in an assembly language.

The computers of the early 1950s were instructed by using the machine's own code, the machine code. To aid the programmer these binary machine codes were improved: the binary representation of the operations of the computing machine were replaced with meaningful symbols. Later, the operands, or addresses, of these operations were also transformed. First, the binary representation of addresses became a decimal representation. The addresses, however, were to be specified in an absolute manner, and the next improvement was the introduction of relative addressing. Finally, the addressing itself also became symbolic: many often used memory locations, like registers, were represented by a symbol. Bit by bit, machine codes evolved into assembly languages.

Although these assembly languages were an improvement over machine codes, it was not enough to prevent the programmer from making errors. To decrease the number of errors, subroutine libraries consisting of useful operations and program parts were developed.[15] New programs could be written by including these subroutines in the code. Creating complex programs had now become a much easier task than writing all the code by hand.[16]

Another way to improve programming was using an automatic coding system. These systems extended a machine with useful features, like floating point operations and input and output operations, by providing a virtual machine that was easier to program than the real computing machine.[17] The ultimate goal of the research groups creating these automatic coding systems was to create a language looking 'like English or algebra, but which the computer could convert into binary machine instructions.'[18]

The biggest problem of automatic coding systems was efficiency, or the lack thereof. Programs generated with these systems were up to five times less efficient than hand-coded assembly programs.[19] This resulted in an atmosphere in which the idea of automatic coding was conceived as fundamentally wrong: 'efficient programming was something that could not be automated'[20] was an often heard statement. This attitude towards automatic programming was put aptly by Backus in 1980:

> Just as freewheeling weste[r]ners developed a chauvinistic pride in their frontiership and a corresponding conservatism, so many programmers of the freewheeling 1950s began to regard themselves as members of a priesthood guarding skills and mysteries far too complex for ordinary mortals. (...) The priesthood wanted and got simple mechanical aids for the clerical drudgery which burdened them, but they regarded with hostility and derision [to] more ambitious plans to make programming accessible to a larger population. To them, it was obviously a foolish and arrogant dream to imagine that any mechanical process could possibly perform the mysterious feats of invention required to write an efficient program.[21]

Unfortunately for this priesthood programming did become accessible to a larger population during the 1950s; hundreds of computers were being built and used on a commercial basis.[22] Programming in assembly language, including the inevitable debugging, accounted for almost three quarters of the total cost of running a computing machine.[23] This observation was the reason why Cuthbert Hurd, Backus's chief at IBM, accepted Backus's proposal for the IBM Mathematical FORmula TRANslating system FORTRAN.[24]

Because automatic coding systems were, in the end, economically more profitable to use automatic coding became accepted. Using automatic coding systems would reduce the time spent on programming and debugging; programmers could spent more time on solving new problems and the computer could spent more time on running these programs rather than on debugging them.

Although many computers were being used for data processing and other business applications during the late 1950s, scientific computing was and would be one of the main application areas. As a result, most automatic coding systems of that time were algorithmic systems. When the attitude towards automatic coding changed in favour of higher level programming languages, 'it almost seemed that each new computer, and even each new programming group, was spawning its own algebraic language or cherished dialect of an existing one.'[25] This diversity of algebraic languages caused the need for a universal algorithmic language. A universal language would enable users of different machines to communicate about programming with each other and to share their programs.

1.1.1 The European field of computing

The first contribution to programming languages in Europe was made by Konrad Zuse. In the years 1945–1946, he developed a programming language called Plankalkül. This language was far more sophisticated than most of the languages developed in the 1950s and even 1960s. Among the features were procedures with input and output parameters, compound statements, records, control statements, variables, subscripts, and an assignment statement.[26][27][28] His work on programming languages, however, was not recognised outside the German-speaking area until 1972.[29]

In 1934, Zuse started working on computing machines, and he produced three successive machines before he started with the Z4 machine.[30] At the end of the Second World War, Zuse fled from Berlin to Bavaria with his Z4 computing machine. Because of the war, Zuse was unable to go on with his work on the Z4 and he started working on several theoretical ideas about computing and programming, one of these ideas was his Plankalkül.[31] After the war, the Z4 was rented by the Eidgenössische Technische Hochschule (ETH) in Zürich and in 1950 it was put into operation.[32] Most of the computations performed on the Z4 were numerical calculations for all kind of complex technical problems, from matrix calculations to computations on a dam.[33]

At that time, Heinz Rutishauser was working at the ETH. To make programming of the Z4 easier he invented a method to let the computer produce its own programs. This method was based on the fact that many programs consisted of repeating sequences of instructions with addresses changing by an underlying pattern.[34] Later, Rutishauser (1952) observed that 'den Rechenplan für eine bestimmte Formel oder Formelgruppe durch die Rechenmaschine selbst "ausrechnen" zu lassen [ist], das heisst die Rechenmaschine [ist] als ihr eigenes Planfertigungsgerät zu benützen.'[35] With "Planfertigungsgerät" Rutishauser referred to another idea of Konrad Zuse on a code generation machine or Planfertigungsgerät for his Plankalkül.[36]

```
**Für** $j = 1(1)n$:
    **Für** $i = 1$:
        **Für** $k = 1$:        $\frac{1}{a_{ik}^{j-1}} = a_{nn}^j.$
        **Für** $k = 2(1)n$:    $a_{nn}^j \times a_{1k}^{j-1} = a_{n,k-1}^j.$
    **Für** $i = 2(1)n$:
        **Für** $k = 1$:        $-a_{ik}^{j-1} \times a_{nn}^j = a_{i-1,n}^j.$
        **Für** $k = 2(1)n$:    $a_{ik}^{j-1} - (a_{i1}^{j-1} \times a_{n,k-1}^j) = a_{i-1,k-1}^j.$
```

Figure 1.0: Example of a program to compute the inversion of matrix a_{ik}^0 in Rutishauser's language.[38]

From 1949 till 1951, Rutishauser developed an algebraic language for an hypothetical computer and two compilers for that language.[39] In this language there was only one control structure: the **Für** statement. In Figure 1.0 the meaning and use of this statement becomes clear: **Für** $k = 2(1)n$: $a_{nn}^j \times a_{1k}^{j-1} = a_{n,k-1}^j$. means that for all k from 2 till n (with step 1), the value of $a_{n,k-1}^j$ becomes the value of a_{nn}^j multiplied with the value of a_{1k}^{j-1}.

At the same time and in the same place, Corrado Böhm, although aware of Rutishauser's work (not vice versa), was doing similar research. Böhm also developed an hypothetical machine and a language to instruct that machine. For the first time, however, the translator itself was written in its own language.[40]. In this language everything was a kind of assignment statement. For example, **go to** B was encoded as $B \rightarrow \pi$: the value of B is assigned to the program counter.[41]

In 1955, Bauer and Samelson decided to start working on formula translation based on Rutishauser's work.[42] They invented a method to translate formulae based on the stack principle.[43] At the end of 1955, after the Darmstadt symposium, the GAMM subcommittee on programming languages was set up. The members of this subcommittee, Bauer and Samelson from Münich, Bottenbruch from Darmstadt, and Rutishauser from Zürich, formed the ZMD group.[44] The proposal for the new algorithmic language created by this group was influenced by their previous work on programming languages and formula translation. Actually, they developed both language and translation technique side by side, as will be explained in Section 3.1.

1.1.2 The difference between Europe and the USA and the need for universalism

The situation in Europe was totally different from the situation in the USA. In Europe the field of computing was just emerging in the late 1950s. In almost every country of Europe, research centres started with building and using their own computing machines. In the USA, on the other hand, a commercial computer industry was already selling computers to research centres, government, and industry.

The computing machines being built in Europe were intended for scientific computing and, eventually, an algorithmic language was needed to make these machines productive. In the USA, several algorithmic languages were already developed and used as a result of this need. Many research groups had created their own algorithmic language. For almost every computing machine an algorithmic language, or variant of another language, existed. In this atmosphere the need for universalism was felt.

When, in late 1958, the report on IAL was published it was received with great interest from the computing community. During the development of ALGOL 60 this enthusiasm would only grow: ALGOL was to be the language of choice for many of the computing machines being built in Europe. In America, however, the market demanded a working implementation of a good language, not an academic promise for a better language. During the development of ALGOL 60 IBM became the dominant player in the computer industry.[45] FORTRAN matured into a usable and efficient language and was ported to many machines. As a result, FORTRAN became the de facto standard programming language for scientific computing.

1.2 Why was IAL chosen over other algorithmic languages?

In 1958, at the Zürich meeting, FORTRAN was not considered to be the ideal universal algorithmic language; IAL was developed instead. The question is: Why did it have to be IAL? Why did FORTRAN, or one of the other languages, not satisfy the those present at the Zürich meeting? Why creating a new algorithmic programming language if there were already other algorithmic languages? To answer these questions four algorithmic languages of the late 1950s are discussed and compared: IT, MATH-MATIC, FORTRAN, and IAL.

IT and MATH-MATIC are discussed because they did belong, according to Jean Sammet, to the group of automatic coding systems which were used more widely.[46] Besides, these languages were created, among others, by respectively C. Katz and A.J. Perlis, both participating in the ALGOL effort. FORTRAN is chosen because it would become the de facto standard and J.W. Backus, the leader of the group developing FORTRAN, also participated in the creation of IAL at the Zürich meeting.

1.2.0 IT

During 1955 and 1956, ideas about an algebraic language and its compiler for the Datatron computer were developed at Purdue University Computing Laboratory. Two members of the group, A. Perlis and J.W. Smith moved to Carnegie Institute of Techology in the summer of 1956. They adapted the ideas about the language to a new machine, the IBM 650. In October 1956, the compiler was ready and named Internal Translator (IT).[47] According to Knuth and Pardo (1975) this was 'the first really *useful* compiler'[48]. It was used on many installations of the IBM 650, and it was even ported and used on some other machines too.[49]

Symbol	*Name*	*Representation*
(	Left parenthesis	L
)	Right parenthesis	R
.	Decimal point	J
$\leftarrow$	Substitution	Z
$=$	Relational equality	U
$>$	Greater than	V
$\geq$	Greater than or equal	W
$+$	Addition	S
$-$	Substraction	M
$\times$	Multiplication	X
/	Division	D
exp	General exponentiation	P
,	Comma	K
"	Quotes	Q
	Type	T
	Finish	F

Figure 1.1: Translation of some of IT's mathematical symbols.[50]

Although IT[51] was popular, it had some disadvantages. The main problem was the primitive hardware representation of the language; only alfanumeric characters were allowed and many symbols were translated into one single character. In Figure 1.1, some of the symbols used in IT are paired with their translation into the hardware representation. Variables were written as a number of I's followed by a number, and, in case of floating point variables, prefixed with a Y or a C. A subroutine was called by its line number followed by a comma separated list of parameters, and all this together was put between quotes.

Besides this primitive hardware representation the scanning technique used was also problematic. This process resulted in incorrect and difficult evaluation of mathematical expressions[52] and it was very time consuming too.[53]

IT itself was a simple arithmetic language; it consisted of simple mathematical operators and some basic control structures like selection and a form of looping. The **IF** statement was used as: k: G I3 **IF** (Y1 + Y2) = 9 meaning that if the value of Y1 + Y2 is equal to 9 then the statement with number equal to the value of variable I3 is executed, otherwise the next statement is executed. Every statement was preceded with a number lower than 626 (here represented by k).

The looping construct was written like k: j, v1, v2, v3, v4. Here j is the number of the last statement of the loop, the loop itself started at statement k+1. v1 is the looping variable, with starting value v2, step v3 and end value v4.

```
   reference                        hardware
1  READ                             READ                F
2  Y2 ← 0                           Y2 Z OJ             F
3  4, I1, 11, -1, 1                 4K I1K 11K M1K 1K   F
4  Y2 ← CI1 + Y1 × Y2               Y2 Z CI1 Y1 X Y2    F
5  H                                H                   FF
```

denoting the computation of $y = \sum_{i=0}^{10} a_i x^i$

Figure 1.2: Example of an IT program.[54]

In Figure 1.2, an example program written in the IT language is given. The reference version of the program is a little bit cryptic but it can be understood. Unfortunately, the hardware representation of the program is worse than cryptic. The programmer had to translate his programs by hand into the hardware representation and, as a result, using IT was difficult and error prone.

1.2.1 MATH-MATIC

The language AT-3 was developed by a group headed by Charles Katz at UNIVAC, starting in 1955. In April 1957, when its preliminary report was published, the language was renamed to MATH-MATIC.[55] The language was intended for use at the UNIVAC I computer, a computing machine without floating point operations. For this reason, all floating point computations had to be done by subroutine calls. Another factor influencing the efficiency in a negative way was the translation of MATH-MATIC programs into A-3 programs. A-3, itself an extension of A-2, was the third version of a series of compilers for the UNIVAC I and well known for its inefficiency. This series of compilers was created by the group headed by Grace Hopper.[56].

```
 (1)  READ-ITEM A(11) .
 (2)  VARY I 10 (-1) 0 SENTENCE 3 THRU 10 .
 (3)  J = I + 1 .
 (4)  Y = SQR | A(J) | + 5 * A(J)^3 .
 (5)  IF Y > 400, JUMP TO SENTENCE 8 .
 (6)  PRINT-OUT I, Y .
 (7)  JUMP TO SENTENCE 10 .
 (8)  Z = 999 .
 (9)  PRINT-OUT I, Z .
(10)  IGNORE .
(11)  STOP .
```

Figure 1.3: The TPK algorithm in MATH-MATIC.[58] The TPK algorithm is not a useful algorithm, but it is used in Knuth's and Pardo's article in which they compare and discuss different early programming languages.

Although the MATH-MATIC implementation was very inefficient, the language did have some interesting features. First and foremost, it was very readable (see Figure 1.3): control statements were written as English words or phrases and expressions were written in the standard mathematical notation. Besides basic control statements the language also contained over twenty input and output statements.[59]

For each of the relational operators $<$, $>$, $=$, and two combinations of these operators, there was a different **IF** statement. There were also five looping constructs, all variations of the start-step-end variant, and one variation with a list of values to iterate over.[60]

The biggest problem for the acceptation of MATH-MATIC was the small number of UNIVAC machines in use and, hence, the small number of users of the language. Sammet remarks rightly that this language could have become the standard for scientific computing if it would have been implemented on a more popular and powerful machine.[61]

1.2.2 FORTRAN

As said earlier, in December 1953, J.W. Backus proposed FORTAN to his chief at IBM.[62] The idea behind the FORTRAN automatic coding system was that programs written in FORTRAN would run as efficient as hand-coded programs in a machine code or in an assembly language.[63] Initially the language was designed for the IBM 704 and some of the statements in the language reflected this machine directly. For example, the statement **IF** (**SENSE LIGHT** *i*) $n1$, $n2$:[64] if sense light i on the IBM 704 is burning, the next statement executed is $n1$; in the case the light is off, the next statement is $n2$.

Besides these machine dependent statements (there were six of them), many input and output statements were included. Famous was the **FORMAT** statement. With this statement it was possible to define and use formatted input and output. Among the control statements were an **IF** statement and a **DO** statement. The statement **IF** (a) $n1$, $n2$, $n3$ denotes that the next statement executed is respectively $n1$, $n2$, or $n3$ if the value of a is less than, equal to, or greather than 0. The use of these statements is clarified in Figure 1.4.

```
C    THE TPK ALGORITHM, FORTRAN STYLE
     FUNF(T) = SQRTF(ABSF(T))+5.0*T**3
     DIMENSION A(11)
1    FORMAT(6F12.4)
     READ 1, A
     DO 10 j = 1,11
     I = 11 − J
     Y = FUNF(A(I+1))
     IF (400.0−Y) 4,8,8
4    PRINT 5, I
5    FORMAT(I10, 10H TOO LARGE)
     GO TO 10
8    PRINT 9, I, Y
9    FORMAT(I10, F12.7)
10   CONTINUE
     STOP 52525
```

Figure 1.4: Again the TPK algorithm from the article of Knuth and Pardo, but now written in FORTRAN.[66]

In June 1958, the second version of FORTRAN was released. The main improvement was the inclusion of subroutines and functions in the language itself. In addition, programs written in assembly language could be linked directly into FORTRAN programs. This new version was also made available on other IBM machines (the IBM 709 and IBM 650 in 1958; the IBM 1620 and IBM 7070 in 1960). In the 1960s, it was implemented on machines of other manufacturers as well. When the language evolved to FORTRAN IV it became quite commonplace. Unfortunately, the different FORTRAN implementations were not always compatible with each other.

1.2.3 IAL

The history of the development of IAL has already been told in Section 1.0. The development distinguished itself from the development of the other languages because it was created at a meeting by two different committees: the ACM subcommittee and the GAMM subcommittee. In addition, these two committees were from different countries, respectively the USA and the German-speaking countries in Europe. Finally, IAL was designed with no particular machine in mind, it was machine independent.

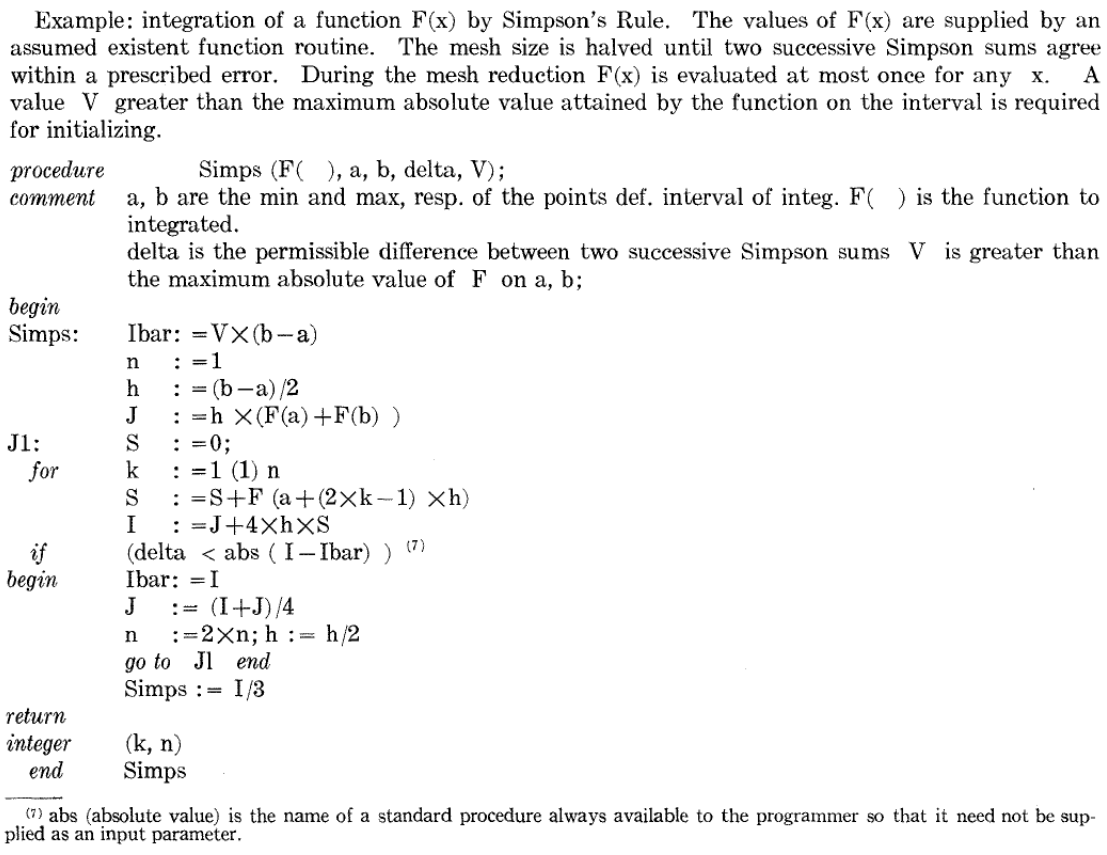
One of the most interesting features of IAL was the compound statement. A sequence of statements separated by semicolons was treated as one statement when it was enclosed by the keywords **begin** and **end**. It fitted very well in combination with control structures like the **if** statement and the **for** statement.

In Figure 1.5 the use of these control structures and the compound statement is exemplified. An **if** statement is formed as **if** B, with B a boolean expression. If B is true, the next statement is executed, otherwise the next statement is omitted. The **for** statement is formed accordingly. **for** i := 0 (1) N means that to the variable i, the values between 0 and N are assigned iteratively. That is, for every value, the statement following the **for** statement is executed once.

One of the problems of the language was the total lack of input and output statements. Because IAL was to be used to describe computational processes only these statements were considered unnecessary.[69] Another problematic aspect was the way procedures were defined. In Section 2.3 this problem is explained in more detail.

1.2.4 Why did IAL have to be the international algebraic language?

At the start of this section, the question was asked why IAL did have to be the universal algorithmic language. After comparing four algorithmic languages



Example: integration of a function F(x) by Simpson's Rule. The values of F(x) are supplied by an assumed existent function routine. The mesh size is halved until two successive Simpson sums agree within a prescribed error. During the mesh reduction F(x) is evaluated at most once for any x. A value V greater than the maximum absolute value attained by the function on the interval is required for initializing.

```
procedure      Simps (F( ), a, b, delta, V);
comment   a, b are the min and max, resp. of the points def. interval of integ. F( ) is the function to
          integrated.
          delta is the permissible difference between two successive Simpson sums V is greater than
          the maximum absolute value of F on a, b;
begin
Simps:    Ibar: =V×(b−a)
          n   : =1
          h   : =(b−a)/2
          J   : =h ×(F(a)+F(b) )
J1:       S   : =0;
  for     k   : =1 (1) n
          S   : =S+F (a+(2×k−1) ×h)
          I   : =J+4×h×S
  if      (delta < abs ( I−Ibar) ) (7)
begin     Ibar: =I
          J    := (I+J)/4
          n    :=2×n; h := h/2
          go to  Jl  end
          Simps := I/3
return
integer   (k, n)
  end     Simps
```

(7) abs (absolute value) is the name of a standard procedure always available to the programmer so that it need not be supplied as an input parameter.

Figure 1.5: An example IAL program taken from the *Preliminary Report: International Algebraic Language*.[68]

this question can be answered.

First of all, IT, MATH-MATIC, and FORTRAN were machine dependent. One of the goals of the ALGOL effort was to create a machine independent language. Of course, one of the already existing language could have been used as a basis for a new machine independent language.

FORTRAN was, by far, the most used language due to the dominance of IBM in the computing industry. Exactly this dominance was the reason for the American subcommittee to ignore the existence of FORTRAN. Choosing FORTRAN as basis for their proposal would only increase the dominant position of IBM.[70] Ironically, FORTRAN would become the de facto universal algorithmic language in the early 1960s.

Second, both IT and MATH-MATIC had some serious problems. IT did not satisfy because of its its primitive hardware representation. The language was too cryptic to be usable as the universal algorithmic language; one of the initial goals of IAL was that it should resemble standard mathematical notation as much as possible. MATH-MATIC, on the other hand, satisfied this requirement very well, it was a neat language. Most of its statements were less machine dependent than the statements in FORTRAN and it had good input and output facilities. Unfortunately, MATH-MATIC's implementation was very inefficient and the language was not very well known; the number of UNIVAC installations and, because of that, the number of MATH-MATIC systems and users was just too small.

Finally, and most importantly, the Europeans had no reason whatsoever to choose an American language as the basis for their proposal. Actually, the result of the Zürich meeting would be a combination of the American and European proposals. As both subcommittees did not choose an existing American programming language the universal international algorithmic language had to be a new language.

The new language, however, shared most of its features with the already existing languages. There was a widespread agreement on the typical elements that should be included in an algorithmic language: algebraic expressions and control structures like selection and a looping construct.

The reason why IAL did have to be the universal algorithmic language was not that the other languages were bad, they just did not satisfy the requirements for an *international* and *universal* language.

1.3 Conclusion

This chapter started with a short history on the start of the ALGOL effort. In the middle of the 1950s, there was a need for a universal algorithmic language and in both the USA and the German-speaking countries of Europe a committee was set up to create a proposal for such a language. In 1958, at the Zürich meeting, these proposals were combined into a new language, called IAL. Although this history seemed to be complete, some questions can be asked: why was there a need for a universal algorithmic language and why did it have to be IAL?

The answer to the first question can only be given by describing the differences between USA and Europe. In the USA the field of computing became a professional field in the 1950s. There was a computer industry selling computers to companies, the government and some research centres. In this commercial atmosphere it became clear that problem-oriented programming languages were needed to make profitable use of computers.

Unfortunately, these programming languages were not accepted by many programmers because they were slow and generated inefficient programs.

It was believed that creating efficient programs could not be automated. J.W. Backus, however, did believe that this was possible, and he proposed the FORTRAN system to his chief at IBM. Indeed, this system would eventually generate programs that ran almost as efficiently as hand-coded programs. Because problem-oriented languages were seen as a necessity, many research groups and computer companies started creating such languages. This development resulted in a huge diversity of similar languages and in this situation the cry for universalism arose.

In Europe, the situation was different. The field of computing was emerging, and in many countries the first computers were being built in the late 1950s. The main application for these machines was scientific computing. Instructing these computers was a difficult and error prone task. To solve this problem, work was started on formula translation and eventually on an algorithmic language. After the international Darmstadt symposium in 1955, Rutishauser, Bauer, Samelson, and Bottenbruch started working on an algorithmic language. Instead of creating yet another algorithmic language, they proposed to jointly create an international algebraic language to the ACM.

Both the Americans and the Europeans did not base their proposal on an already existing language. IT and MATH-MATIC were not sufficient or well known. FORTRAN, on the other hand, was well known, but the Americans wanted to undermine the dominance of IBM. Hence, the ACM subcommittee proposed a new language. The combination of the two proposals became IAL: a new algorithmic language like other algorithmic languages.

The reason why IAL did have to be the international algebraic language was not that it was better than the other programming languages, but that it was machine independent and a compromise between two computing communities with the potential to become part of a truly international effort.

1.4 Notes

[0]Heinz Rutishauser, *Description of ALGOL 60*, volume 1, edited by F. L. Bauer et al. (Springer-Verlag, 1967), p. 5

[1]Peter Naur, *Transcripts of Presentations*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), pp. 148, Frame 3

[2]Rutishauser, *Description of ALGOL 60*, p. 5

[3]R. W. Bemer, 'A Politico-Social History of Algol', in: Mark I. Halpern and Christopher J. Shaw, editors, *Annual review in automatic programming*, volume 5 (London: Pergamon, 1969), p. 160

[4]Alan J. Perlis, *The American side of the development of Algol*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), pp. 4–5

[5]Bemer, ‘A Politico-Social History of Algol’, p. 160

[6]Perlis, ‘The American side of the development of Algol’, p. 5

[7]Rutishauser, *Description of ALGOL 60*, p. 5

[8]Bemer, ‘A Politico-Social History of Algol’, p. 161

[9]Rutishauser, *Description of ALGOL 60*, p. 6

[10]A. J. Perlis and K. Samelson, ‘Preliminary Report: International Algebraic Language’, *Commun. ACM* 1:12 (1958), p. 9

[11]Perlis, ‘The American side of the development of Algol’, p. 6

[12]Perlis and Samelson, ‘Preliminary report: IAL’

[13]John Backus, *The history of FORTRAN I, II, and III*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), p. 166

[14]Programming Research Group IBM, ‘Preliminary Report – Specifications for the IBM Mathematical FORmula TRANslating System FORTRAN’, Technical report (New York: IBM, 1954), ⟨URL: http://community.computerhistory.org/scc/projects/FORTRAN/BackusEtAl-PreliminaryReport-1954.pdf⟩, p. 1

[15]Martin Campbell-Kelly, *Computer: a history of the information machine* (BasicBooks, 1996), p. 186

[16]Jean E. Sammet, *Programming languages : history and fundamentals*, Series in Automatic Computation (Englewood Cliffs, N. J.: Prentice-Hall, 1969), pp. 3–4

[17]Backus, ‘The history of FORTRAN I, II, and III’, p. 165

[18]Campbell-Kelly, *Computer: a history of the information machine*, p. 187

[19]Backus, ‘The history of FORTRAN I, II, and III’, p. 165

[20]ibid.

[21]Idem, ‘Programming in America in the 1950s – Some Personal Impressions’, in: N. Metropolis, J. Howlett and Gian-Carlo Rota, editors, *A History of Computing in the twentieth century* (Academic Press, 1980), pp. 127–128

[22]Saul Rosen, ‘Programming Systems and Languages. A Historical Survey’, in: Idem, editor, *Programming systems and languages* (London: McGraw-Hill, 1967), p. 3

[23]Backus, ‘The history of FORTRAN I, II, and III’, p. 166

[24]Campbell-Kelly, *Computer: a history of the information machine*, p. 188

[25]Perlis, ‘The American side of the development of Algol’, p. 4

[26]Friedrich L. Bauer, ‘Between Zuse and Rutishauser – The Early Development of Digital Computing in Central Europe’, in: N. Metropolis, J. Howlett and Gian-Carlo Rota, editors, *A History of Computing in the twentieth century* (Academic Press, 1980), p. 514-515

[27]Konrad Zuse, ‘Some Remarks on the History of Computing in Germany’, in: N. Metropolis, J. Howlett and Gian-Carlo Rota, editors, *A History of Computing in the twentieth century* (Academic Press, 1980), pp. 620–627

[28]F. L. Bauer and H. Wössner, ‘The ‘Plankalkül’ of Konrad Zuse: a forerunner of today’s programming languages’, *Commun. ACM* 15:7 (1972)

[29]Donald E. Knuth and Luis Trabb Pardo, ‘Early Development of Programming Languages’, in: Jack Belzer, Albert G. Holzman and Allen Kent, editors, *Encyclopedia of Computer Science and Technology*, volume 7 (Marcel Dekker INC., 1975), p. 425

[30]Zuse, ‘Some Remarks on the History of Computing in Germany’, pp. 611–612

[31]Knuth and Pardo, ‘Early Development of Programming Languages’, p. 424

[32]H. R. Schwarz, ‘The Early Years of Computing in Switzerland’, *Annals of the History of Computing* 3:2 (1981), p. 121

[33]ibid., pp. 123–125

[34]ibid., p. 125

[35]H. Rutishauser, 'Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen', *Z. Angew. Math. Mech.* 32:3 (1952), p. 312

[36]Bauer, 'Between Zuse and Rutishauser', p. 516

[37]Rutishauser, 'Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen', p. 313

[38]ibid.

[39]Knuth and Pardo, 'Early Development of Programming Languages', pp. 438–439

[40]ibid., p. 440

[41]ibid., p. 443

[42]K. Samelson and F. Bauer, *The ALCOR project*, in: Gordon and Breach, editors, *Symbolic languages in data processing: Proc. of the Symp. organized and edited by the Int. Computation Center, Rome, 26-31 March 1962* (New York, 1962), p. 207

[43]Friedrich L. Bauer, *From the Stack Principle to ALGOL*, in: Manfred Broy and Ernst Denert, editors, *Software pioneers : contributions to software engineering* (Berlin: Springer, 2002), pp. 30–33

[44]ibid., p. 34

[45]Kenneth Flamm, *Creating the Computer* (The Brookings Institution, 1988), p. 102

[46]Sammet, *Programming languages : history and fundamentals*, p. 134

[47]Knuth and Pardo, 'Early Development of Programming Languages', pp. 474–475

[48]ibid., p. 475

[49]Rosen, 'Programming Systems and Languages. A Historical Survey', p. 7

[50]Sammet, *Programming languages : history and fundamentals*, p. 139

[51]This text about IT is, unless stated otherwise, based on: ibid., pp. 139–143

[52]ibid., p. 139

[53]Rosen, 'Programming Systems and Languages. A Historical Survey', p. 7

[54]This example does (probably) contain errors, however, it is taken as it is from: Sammet, *Programming languages : history and fundamentals*, p. 141

[55]Knuth and Pardo, 'Early Development of Programming Languages', p. 479

[56]ibid., pp. 452–455

[57]ibid., p. 479

[58]ibid.

[59]Sammet, *Programming languages : history and fundamentals*, p. 136

[60]ibid.

[61]ibid., p. 137

[62]Backus, 'The history of FORTRAN I, II, and III', p. 166

[63]ibid., p. 167

[64]The text of this section is, unless stated otherwise, based on: Sammet, *Programming languages : history and fundamentals*, pp. 143–172

[65]Knuth and Pardo, 'Early Development of Programming Languages', p. 478

[66]ibid.

[67]Perlis and Samelson, 'Preliminary report: IAL', p. 22

[68]ibid.

[69]Sammet, *Programming languages : history and fundamentals*, p. 175

[70]Rosen, 'Programming Systems and Languages. A Historical Survey', p. 10

2 Notation
From IAL to ALGOL 60

Why notation matters? • The description of early languages: FORTRAN and IAL • Backus's notation to describe IAL • Used as basis for the definition of ALGOL 60 • The ALGOL 60 report • And the language ALGOL 60

2.0 Why notation matters?

Communication between the members of the various working groups on aspects of the new algorithmic language was important because the ALGOL effort was an international effort consisting of different people and groups. A clear and unambiguous understanding of the language in all its facets by all members participating in the effort was necessary to be able to develop and discuss the ALGOL language. Using English or any other natural language to describe a programming language was, and is, insufficient. Natural languages are too ambiguous to define formally or even describe a programming language completely.

A description of a programming language consists of a description of two related aspects of the language: the syntax and the semantics. The syntax of a language is about how to form a string in the language. The semantics is about the meaning of syntactically correct strings in the language. The most obvious way to define formally the semantics is an implementation of the language on some machine. This implementation then fixes *the* meaning of the language.

Such an implementation is not a workable description of a language. A typical implementation is too large to be understood easily. Often it is written in some low level programming language or even assembly language. In addition, a compiler is written for a particular machine for which the details should be known to understand the details of the implementation.

Fortunately, describing the syntax formally appeared to be an easier task than describing the semantics of a language. During the development of AL-

GOL a formal notation was invented to describe the syntax of ALGOL. This notation, or metalanguage, was not complete or perfect but it fulfilled its task to prevent the occurrence of many ambiguities in the discussions about ALGOL. Although the new notation was primarily intended for describing the syntax, the semantics were not completely ignored. Actually, the description of the semantics was mixed in with the description of the syntax.

Usually a special notation to describe the syntax also influences the language concepts to be described. Some language concepts are more easily described in one notation than in an other one. A formal notation will result in more coherent and simple concepts: the notation forces the concepts to be described along the rules of the notation.

In this chapter, both the development of the notation used to describe the syntax and semantics of ALGOL and the development of the language concepts of ALGOL 60 are discussed. To that end, the notation used for describing FORTRAN and IAL is explained first. After that the focus is on the first stage of the development of the notation: Backus's notation. The third topic is the notation of the ALGOL 60 report. Finally, the development of some problematic programming language concepts in ALGOL, especially that of the procedure concept, are treated in more detail.

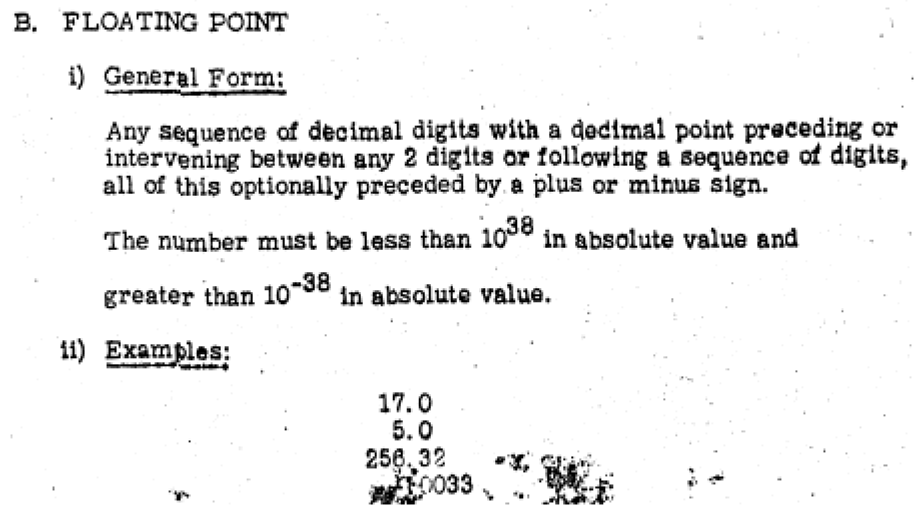
2.1 Notations used to describe early programming languages

The notations used to describe the early programming languages were, like the languages they described, primitive. During the ALGOL effort this notation was developing into an important aspect of the field of programming languages. Actually, one of the results of this effort was a way to define programming languages: do it as it was done in the ALGOL 60 report.

To describe this development of notation, the notation used before the ALGOL effort is explained and compared to the notation used to describe IAL. This comparison answers the question if this development of notation started during or before the ALGOL effort. In other words: Was the nature of the ALGOL effort responsible for this development of notation?

2.1.0 The notation of IAL and FORTRAN compared

In 1954, the first document describing FORTRAN[0] was published: *Preliminary Report – Specifications for the IBM Mathematical FORmula TRANslating System FORTRAN.*[1] Two years later, at 15 October 1956, a more finalised version of the language was published.[2] The notation used in this later document was the same as in the preliminary report; the notation to describe a programming language had not changed fundamentally during these two years.


B. FLOATING POINT

i) General Form:

Any sequence of decimal digits with a decimal point preceding or intervening between any 2 digits or following a sequence of digits, all of this optionally preceded by a plus or minus sign.

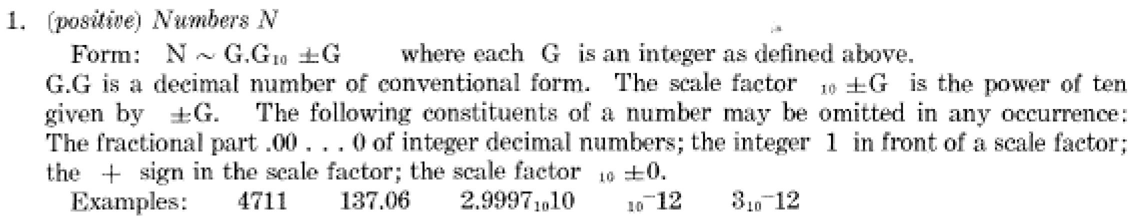
The number must be less than 10^{38} in absolute value and greater than 10^{-38} in absolute value.

ii) Examples:

17.0
5.0
256.32
[illegible]0033 [illegible]

Figure 2.0: The description of real numbers in the FORTRAN report.[3]

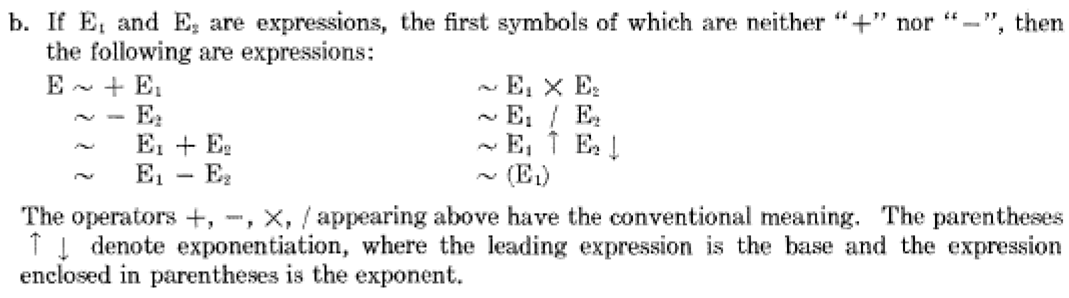
As was the case with FORTRAN, the first publication of IAL was also a preliminary report: *Preliminary Report: International Algebraic Language*[4]. Actually, both preliminary reports do resemble each other in the sense that both documents were set up in a similar way. To compare the notations used in both reports, the descriptions for some language elements are given and compared.


1. *(positive) Numbers N*

Form: $N \sim G.G_{10} \pm G$ where each G is an integer as defined above.
G.G is a decimal number of conventional form. The scale factor $_{10} \pm G$ is the power of ten given by $\pm G$. The following constituents of a number may be omitted in any occurrence: The fractional part .00 . . . 0 of integer decimal numbers; the integer 1 in front of a scale factor; the + sign in the scale factor; the scale factor $_{10} \pm 0$.
Examples: 4711 137.06 $2.9997_{10}10$ $_{10}{-12}$ $3_{10}{-12}$

Figure 2.1: The description of real numbers in the IAL report [6], here G is a string containing digits only.

Real numbers in IAL (Figure 2.1) are described using a pattern reflecting the form of a real number. Compared to the description of real numbers in FORTRAN (Figure 2.0), where only natural language is used, the form of a number is more obvious.



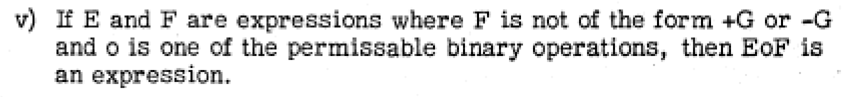
b. If E_1 and E_2 are expressions, the first symbols of which are neither "+" nor "−", then the following are expressions:

$$\begin{array}{llll} E \sim + E_1 & & & \sim E_1 \times E_2 \\ \sim - E_2 & & & \sim E_1 \; / \; E_2 \\ \sim & E_1 + E_2 & & \sim E_1 \uparrow E_2 \downarrow \\ \sim & E_1 - E_2 & & \sim (E_1) \end{array}$$

The operators +, −, ×, / appearing above have the conventional meaning. The parentheses ↑ ↓ denote exponentiation, where the leading expression is the base and the expression enclosed in parentheses is the exponent.

Figure 2.2: Part of the description of expressions in the IAL report.[7]

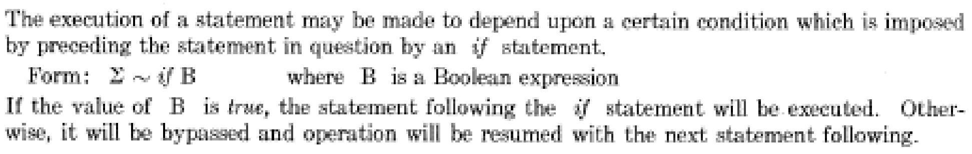
A more complicated language construct was the expression. Again, in IAL (Figure 2.2) patterns are used to clarify the form of an expression whereas in FORTRAN (Figure 2.3) natural language is used. The difference between the two notations is much smaller than before: a pattern is used in both descriptions. In IAL's description this pattern is more explicit, however.



v) If E and F are expressions where F is not of the form +G or -G and o is one of the permissable binary operations, then EoF is an expression.

Figure 2.3: Part of the description of expressions in the FORTRAN report.[9] The 'permissible binary operations' were described earlier and include the normal ones.

The notations used for statements were also alike. For example, the **if** statement is described in both preliminary reports with a pattern (compare Figure 2.5 with Figure 2.4). Although this version of FORTAN did not have that much declarations, the description of declarations was more or less similar with those in the IAL report. In a later version of FORTRAN more declarations were added to the language.



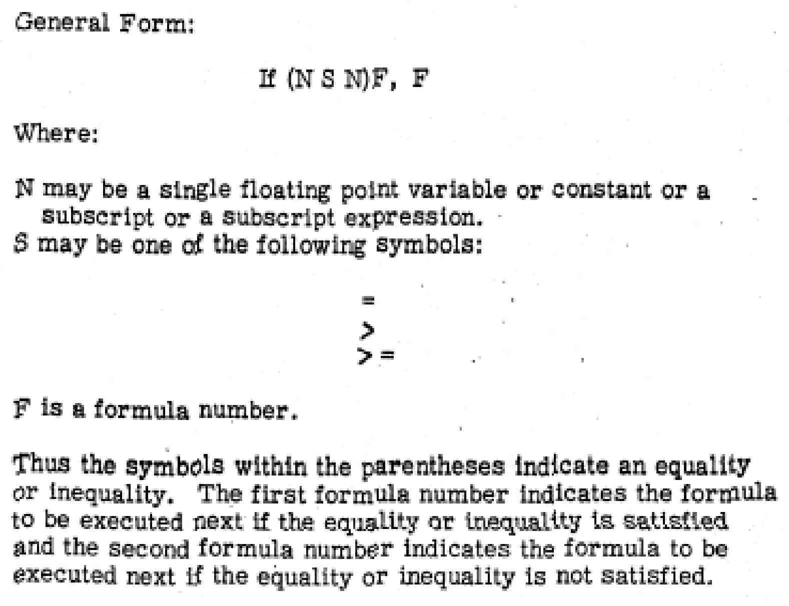
The execution of a statement may be made to depend upon a certain condition which is imposed by preceding the statement in question by an *if* statement.

Form: $\Sigma \sim$ *if* B where B is a Boolean expression

If the value of B is *true*, the statement following the *if* statement will be executed. Otherwise, it will be bypassed and operation will be resumed with the next statement following.

Figure 2.4: Part of the description of the **if** statement in the IAL report.[10]

In short, the notation used to describe FORTRAN was similar to the notation used to describe IAL. In the description of FORTRAN natural language was used more often than in the description of IAL. The general form


General Form:

If (N S N)F, F

Where:

N may be a single floating point variable or constant or a subscript or a subscript expression.
S may be one of the following symbols:

=
>
> =

F is a formula number.

Thus the symbols within the parentheses indicate an equality or inequality. The first formula number indicates the formula to be executed next if the equality or inequality is satisfied and the second formula number indicates the formula to be executed next if the equality or inequality is not satisfied.

Figure 2.5: Part of the description of the **if** statement in the FORTRAN report.[12]

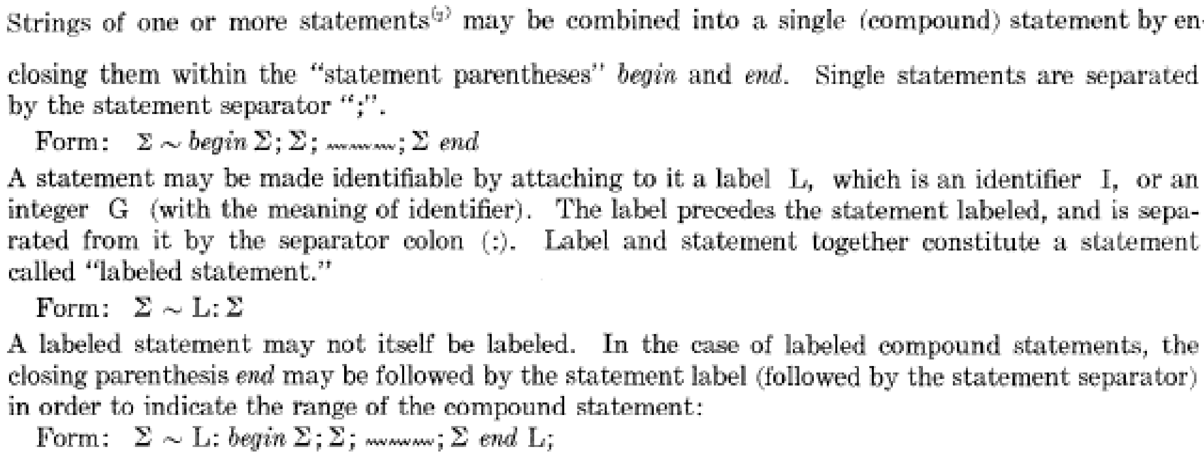
of a language element was described with a pattern in both descriptions. These patterns were the basis of the description of IAL. This was probably the influence of the European part of the designing committee. As we have seen before, the Europeans tended more to logic and mathematics than the Americans did.

2.1.1 The quality of IAL's notation

The notations used to describe both FORTRAN and IAL were similar. This does not say anything about the quality of the notation, however. The question is: Was this notation sufficient to describe a programming language formally and completely? To answer this question, a closer look at IAL and its notation is taken.

Even for simple language elements, like algebraic expressions, IAL's notation was not good enough. In Figure 2.2 a part of the definition of algebraic expressions using this notation is given. An algebraic expression E has simply one of the forms occurring on the right hand side of the "$\sim$" symbols. Although it is assumed that the operators in this definition do have the 'conventional meaning,'[13] this definition is ambiguous and it does not say anything about operator precedence, nor about associativity.

When defining an element with an unknown number of subelements or



Strings of one or more statements(9) may be combined into a single (compound) statement by enclosing them within the "statement parentheses" *begin* and *end*. Single statements are separated by the statement separator ";".

Form: $\Sigma \sim$ *begin* Σ; Σ; $\sim\!\sim\!\sim$; Σ *end*

A statement may be made identifiable by attaching to it a label L, which is an identifier I, or an integer G (with the meaning of identifier). The label precedes the statement labeled, and is separated from it by the separator colon (:). Label and statement together constitute a statement called "labeled statement."

Form: $\Sigma \sim$ L: Σ

A labeled statement may not itself be labeled. In the case of labeled compound statements, the closing parenthesis *end* may be followed by the statement label (followed by the statement separator) in order to indicate the range of the compound statement:

Form: $\Sigma \sim$ L: *begin* Σ; Σ; $\sim\!\sim\!\sim$; Σ *end* L;

Figure 2.6: Part of the definition of the compound statement.[14]

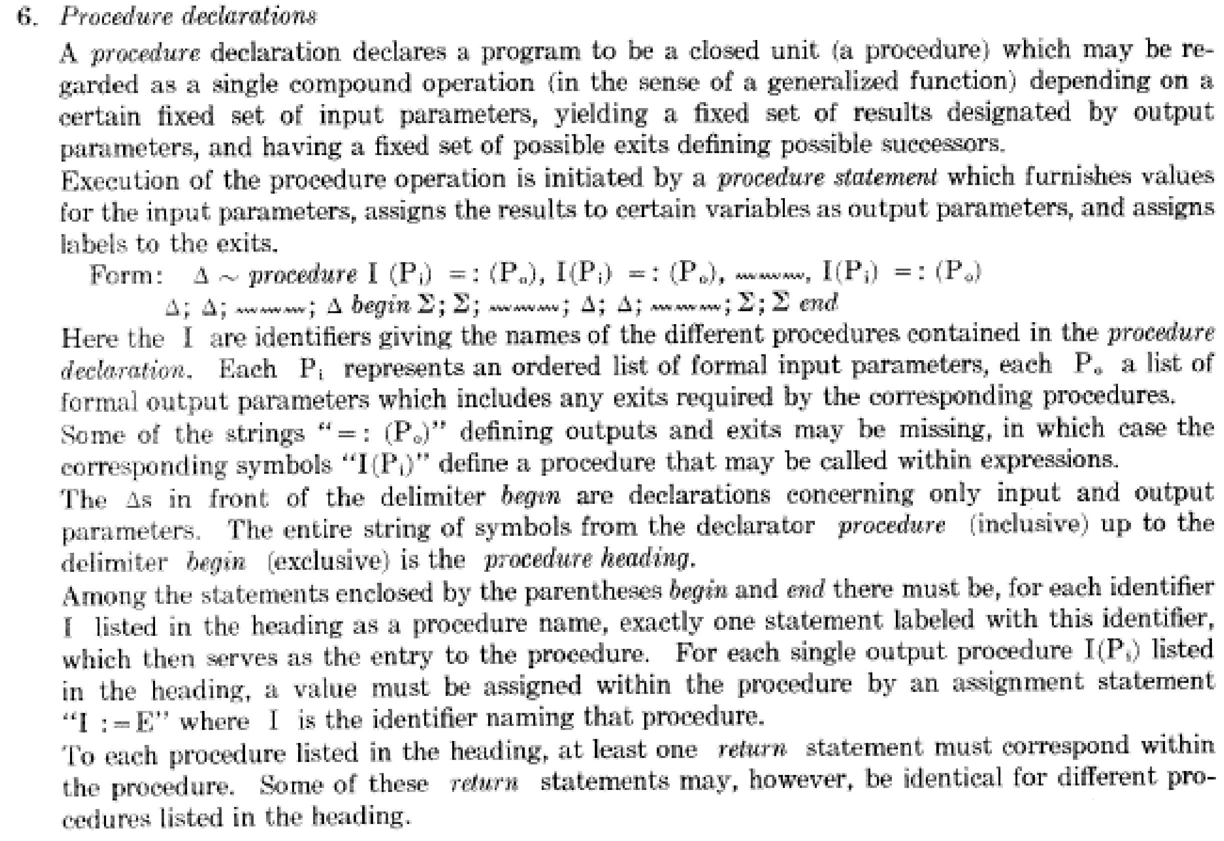
when there are constraints on the occurrence of some subelements this notation was problematic. Take, for example, the compound statement (Figure 2.6). The pattern of this compound statement is a number of statements between the **begin** and **end** keywords. At first glance, this seems to define this compound statement completely.

Unfortunately it is not clear if it was possible to have a compound statement containing no statements at all, only one statement, or even two statements. After all, there are three "Σ" symbols in the pattern describing the compound statement. The use of "$\cdots$" in itself may be clear and completely understandable for most of the readers of the report; those readers were almost all used to read mathematical texts in which these shortscripts occurred often. It does, however, not describe formally and fully what should occur on those "$\cdots$".

This compound statement consists of a number of unconstrained elements only and the use of "$\cdots$" would be acceptable if the only meaning was zero or more times the repeated element. Unacceptable for a formal notation are constraints on the elements in a pattern added outside the pattern using natural language.

As an example, part of the definition of the procedure declaration is given in Figure 2.7. Here the "$\cdots$" are used not only to denote zero or more occurrences of one element denoted by a single symbol, it has three different meanings in the same pattern:

1. One or more occurrences of a procedure signature. The procedure signature itself consisted of more than one symbol and a part of unknown



6. *Procedure declarations*

A *procedure* declaration declares a program to be a closed unit (a procedure) which may be regarded as a single compound operation (in the sense of a generalized function) depending on a certain fixed set of input parameters, yielding a fixed set of results designated by output parameters, and having a fixed set of possible exits defining possible successors.

Execution of the procedure operation is initiated by a *procedure statement* which furnishes values for the input parameters, assigns the results to certain variables as output parameters, and assigns labels to the exits.

Form: $\Delta \sim$ *procedure* $\mathrm{I}\ (\mathrm{P}_i)\ =:\ (\mathrm{P}_o),\ \mathrm{I}(\mathrm{P}_i)\ =:\ (\mathrm{P}_o),$ ∼∼∼∼, $\mathrm{I}(\mathrm{P}_i)\ =:\ (\mathrm{P}_o)$
$\Delta;\ \Delta;$ ∼∼∼∼; Δ *begin* $\Sigma;\ \Sigma;$ ∼∼∼∼; $\Delta;\ \Delta;$ ∼∼∼∼; $\Sigma;\ \Sigma$ *end*

Here the I are identifiers giving the names of the different procedures contained in the *procedure declaration*. Each P_i represents an ordered list of formal input parameters, each P_o a list of formal output parameters which includes any exits required by the corresponding procedures. Some of the strings "$=:\ (\mathrm{P}_o)$" defining outputs and exits may be missing, in which case the corresponding symbols "$\mathrm{I}(\mathrm{P}_i)$" define a procedure that may be called within expressions.

The Δs in front of the delimiter *begin* are declarations concerning only input and output parameters. The entire string of symbols from the declarator *procedure* (inclusive) up to the delimiter *begin* (exclusive) is the *procedure heading*.

Among the statements enclosed by the parentheses *begin* and *end* there must be, for each identifier I listed in the heading as a procedure name, exactly one statement labeled with this identifier, which then serves as the entry to the procedure. For each single output procedure $\mathrm{I}(\mathrm{P}_i)$ listed in the heading, a value must be assigned within the procedure by an assignment statement "$\mathrm{I}\ :=\mathrm{E}$" where I is the identifier naming that procedure.

To each procedure listed in the heading, at least one *return* statement must correspond within the procedure. Some of these *return* statements may, however, be identical for different procedures listed in the heading.

Figure 2.7: Part of the definition of procedure declarations.[15]

length, namely the parameter part of the procedure signature. Actually the part =: (P_o) is optional.

2. Zero or more occurrences of a declaration about the input and output parameters defined in the procedure signatures.

3. Zero or more occurrences of two different elements: statements Σ and declarations Δ. These different elements can be mixed in any order. For every procedure signature, however, there should be a label identical to the procedure identifier preceding minimally one statement. In addition, each procedure must have a return statement and all output parameters should have an assignment statement assigning the value to output the result.

This early notation to describe programming languages was not suitable to define a language like IAL fully and formally. To be able to do so was necessary because IAL was intended to be machine independent: IAL was to be implemented on various machines by different people. According to Backus (1959): 'there must exist a precise description of those sequences of symbols which constitute legal IAL programs.'[16] But, 'heretofore there has existed no formal description of a machine-independent language'.[17] For this reason, Backus started to work on such a formal description.

2.2 Backus's notation

At the UNESCO International Conference on Information Processing, held at Paris from 15 till 20 June 1959, J.W. Backus presented *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference*[18] about a formal description of the syntax of IAL. To be able to describe the syntax formally he invented a new metalanguage based on Emile Post's production system.[19] This notation became known as the Backus Normal Form and later as the Backus Naur Form,[20] it is, however, best known by its abbreviation BNF.

Using this notation, the syntax of a language could be described by "production rules". Each rule was of the shape <metalinguistic variable> :≡ pattern. A pattern was built up from metalinguistic variables and symbols of the language. All possible patterns for a metalinguistic variable were connected with the *or* symbol, denoting a choice between the different patterns for the metalinguistic variable.

$$\begin{aligned} \langle\mathsf{digit}\rangle &:\equiv \ \mathsf{0}\ or\ \mathsf{1}\ or\ \mathsf{2}\ or\ \mathsf{3}\ or\ \mathsf{4}\ or\ \mathsf{5}\ or\ \mathsf{6}\ or\ \mathsf{7}\ or\ \mathsf{8}\ or\ \mathsf{9} \\ \langle\mathsf{integer}\rangle &:\equiv \ \langle\mathsf{digit}\rangle\ or\ \langle\mathsf{integer}\rangle\ \langle\mathsf{digit}\rangle \end{aligned}$$

Figure 2.8: The formal definition of integers and digits using Backus's notation.[22]

A simple example of the application of Backus's notation is the description of integers (Figure 2.8). A digit is clearly a number, or better, a symbol representing a number, between 0 and 9. An integer is now built up from these simple digits: it is either one simple digit, or it is a integer followed by a simple digit, like 9237.

Another simple and clear example is the description of arithmetic expressions. Comparing Figure 2.2 with Figure 2.9, it is immediately clear that the latter is a less ambiguous description than the former description. Using his notation Backus was able to denote the operator precedence by splitting up the description of expressions into different parts: factors, terms and expressions.

$$
\begin{aligned}
\langle\text{factor}\rangle :\equiv\ & \langle\text{number}\rangle \textit{ or } \langle\text{function}\rangle \textit{ or } \langle\text{variable}\rangle \textit{ or } \langle\text{subscr var}\rangle \textit{ or } (\ \langle\text{ar exp}\rangle\ ) \textit{ or } \\
& \langle\text{factor}\rangle \uparrow \langle\text{ar exp}\rangle \downarrow \\
\langle\text{term}\rangle :\equiv\ & \langle\text{factor}\rangle \textit{ or } \langle\text{term}\rangle \times \langle\text{factor}\rangle \textit{ or } \langle\text{term}\rangle\ /\ \langle\text{factor}\rangle \\
\langle\text{ar exp}\rangle :\equiv\ & \langle\text{term}\rangle \textit{ or } + \langle\text{term}\rangle \textit{ or } - \langle\text{term}\rangle \textit{ or } \langle\text{ar exp}\rangle + \langle\text{term}\rangle \textit{ or } \langle\text{ar exp}\rangle - \langle\text{term}\rangle \\
\langle\text{ar exp A}\rangle :\equiv\ & \langle\text{ar exp}\rangle \\
\langle\text{relation}\rangle :\equiv\ & < \textit{ or } > \textit{ or } \leq \textit{ or } \geq \textit{ or } = \textit{ or } \neq \\
\langle\text{rel exp}\rangle :\equiv\ & (\ \langle\text{ar exp}\rangle\ \langle\text{relation}\rangle\ \langle\text{ar exp A}\rangle\ )
\end{aligned}
$$

Figure 2.9: The formal definition of arithmetic expressions using Backus' notation.[24]

$$
\langle\text{param list}\rangle :\equiv\ \langle\text{param}\rangle \textit{ or } \langle\text{param list}\rangle,\ \langle\text{param}\rangle
$$

Figure 2.10: The formal definition of a parameter list.[25]

Some other problems with the older notation were also solved by this new notation: the use of "$\cdots$" to denote the occurrence of an element a (unknown) number of times was not needed any more. In Backus's notation, it was possible to use recursion and, hence, to specify that an element could occur a number of times. In Figure 2.10 the use of this recursivity is made clear by describing a parameter list: a parameter list is either just one parameter, or it is a parameter list followed by a comma and one parameter.

Choice could be denoted by using the *or* connective: write two patterns, one with the element of choice and one without it. Both recursion and choice were used to describe procedure statements (Figure 2.11). Unfortunately, Backus was not able to write down a formal, *clear* and understandable description of the procedure statement. Even using his new notation, Backus had to write down some remarks about the statement using natural language to complete the *formal* description; the procedure statement was too difficult to describe formally. In addition, the description of the declaration of procedures was not included at all.

The new notation was an huge improvement over the one used earlier. Nonetheless, it was improved further by Peter Naur. He replaced *or* by | and $:\equiv$ by $::=$. With this, and with the use of complete words for metalinguistic variables instead of using abbreviations of the same words as did Backus, Naur improved the readability of the description.[26] The most important contribution of Peter Naur to Backus's notation, however, was the fact that he used it in the ALGOL 60 report.[27] Only after the publication of that report the BNF became more widely known. Before the publication it appeared to

⟨oe⟩ :≡ ⟨left element⟩
⟨out list⟩ :≡ ⟨oe⟩ *or* ⟨outlist⟩, ⟨oe⟩
⟨suc⟩ :≡ ⟨label⟩ *or* ⟨id⟩ [⟨exp⟩]
⟨succr list⟩ :≡ ⟨suc⟩ *or* ⟨succr list⟩, ⟨suc⟩
⟨A⟩ :≡ =:(⟨out list⟩) *or* ⟨blank⟩
⟨B⟩ :≡ :(⟨succr list⟩) *or* ⟨blank⟩
⟨proc stmt⟩ :≡ ⟨function⟩ ⟨A⟩ ⟨B⟩ *or* ⟨id⟩ =:(⟨outlist⟩) ⟨B⟩ *or* ⟨id⟩:(⟨succr list⟩)
⟨ppol⟩ :≡ ⟨blank⟩ *or* ⟨ppol⟩ ⟨oe⟩,
⟨pol⟩ :≡ ⟨ppol⟩ *or* ⟨pol⟩, *or* ⟨pol⟩, ⟨oe⟩
⟨A'⟩ :≡ =:(⟨pol⟩)
⟨ppsl⟩ :≡ ⟨blank⟩ *or* ⟨ppsl⟩ ⟨suc⟩,
⟨psl⟩ :≡ ⟨ppsl⟩ *or* ⟨psl⟩, *or* ⟨psl⟩, ⟨suc⟩
⟨B'⟩ :≡ :(⟨psl⟩)
⟨F*⟩ :≡ ⟨function⟩ *or* ⟨pure function⟩ *or* ⟨id⟩
⟨A*⟩ :≡ ⟨A⟩ *or* ⟨A'⟩
⟨B*⟩ :≡ ⟨B⟩ *or* ⟨B'⟩
⟨pure procedure⟩ :≡ ⟨pure function⟩ ⟨A*⟩ ⟨B*⟩ *or* ⟨F*⟩ ⟨A'⟩ ⟨B*⟩ *or* ⟨F*⟩ ⟨A*⟩ ⟨B'⟩

[a pure procedure may have any of the forms of a procedure statement but at least one position of one existing list must be empty; at least one input parameter position or one output position or one successor position].

Figure 2.11: The formal definition of the procedure statement.[30]

Peter Naur that Backus's description of IAL 'was received with a silence that made it seem that precise syntax description was an idea whose time had not yet come'.[28] Naur 'thus proved the usefulness of the idea in a widely read paper and it was accepted'.[29]

2.3 Developing ALGOL 60

During the eighteen months between the meeting in Zürich and the next joint meeting on the international algebraic language, held in Paris, January 1960, the name of the language had changed from the '"unspeakable" and pompous acronym, IAL'[31] to ALGOL. Furthermore, the language was discussed among interested people from America and various countries of Europe. In these months the ALGOL effort became a truly international effort, but it was still a separated effort. The main development took place at different meetings and in correspondence between members of the various subcommittees. The official communications channels of the development, however, were the *Communications of the ACM* in the USA and the *ALGOL Bulletin* in Europe.

Most of the American proposals were related to practical aspects of the language. They wanted to improve the language by extending it, by adding more types, and by adding input and output facilities. Another suggestion to improve the language was to tidy up the syntax a bit.[32] This practical attitude to the ALGOL effort was a result of the state of programming in the USA: programming was becoming a professional field and the experience gained with existing programming languages provided a good feedback to the ALGOL effort.

The European proposals were often focused on the procedure concept and the scopes of variables.[33] The Europeans aimed to improve the language fundamentally and the main target was the difficult procedure concept in IAL. Both in the preliminary report on IAL and in Backus's description of IAL this procedure concept could not be described fully and formally in the notation used. Aside from this notational problem, other problems with IAL's procedure concept were also noted and discussed.

The discussions on the procedure concept focused mainly on parameters. E.T. Irons and F.S. Acton (1959) sum up some problems with IAL's parameters in *A proposed interpretation in ALGOL*[34]. Parameters could occur in the procedure body at the left hand side of an assignment statement. When the procedure was called these parameters in the body were replaced with the argument supplied for that parameter. If an argument was not an assignable element (i.e. not a variable) it would result in undefined behaviour. Another problem mentioned was the use of one argument as both an input and an output parameter. Actually, one parameter could also be used as both an input and an output parameter.[35]

These problems with the procedure concept were resolved by various subcommittees at the final meeting in Paris. First, the distinction between input and output parameters was removed.[36] This solved a number of problems but not all of them. Eventually, under great time pressure, the distinction between *call-by-name* (enabling the so-called Jensen's device) and *call-by-value* was invented.[37] Every occurrence of a call-by-name parameter in the body of a procedure was being substituted by the name of the argument supplied to the procedure for that parameter. If a parameter was called by value, however, then the *value* of the argument was assigned to all occurrences of the corresponding parameter in the procedure body. This call-by-name parameter concept was one of the most controversial features of ALGOL 60.

Another issue at the Paris meeting was recursion. Recursive procedures were new in 1960 and the usefulness of it was not widely recognised. The

```
⟨formal parameter⟩::=⟨identifier⟩
⟨formal parameter list⟩::=⟨formal parameter⟩|
    ⟨formal parameter list⟩⟨parameter delimiter⟩⟨formal parameter⟩
⟨formal parameter part⟩::=⟨empty⟩|(⟨formal parameter list⟩)
⟨identifier list⟩::=⟨identifier⟩|⟨identifier list⟩, ⟨identifier⟩
⟨value part⟩::= value ⟨identifier list⟩; |⟨empty⟩
⟨specifier⟩::= string |⟨type⟩| array |⟨type⟩ array | label | switch |
    procedure | ⟨type⟩ procedure
⟨specification part⟩::=⟨empty⟩| ⟨specifier⟩⟨identifier list⟩;|
    ⟨specification part⟩⟨specifier⟩⟨identifier list⟩;
⟨procedure heading⟩::=⟨procedure identifier⟩⟨formal parameter part⟩;
    ⟨value part⟩⟨specification part⟩
⟨procedure body⟩::=⟨statement⟩|⟨code⟩
⟨procedure declaration⟩::= procedure ⟨procedure heading⟩⟨procedure body⟩|
    ⟨type⟩ procedure ⟨procedure heading⟩⟨procedure body⟩
```

Figure 2.12: The formal description of the ALGOL 60 procedure declaration.[39]

proposal to add the **recursive** keyword to the language to denote a recursive procedure was rejected.[40] When ALGOL 60 was published, however, it appeared that there was no restriction on the occurrence of calls to a procedure in its own procedure body. Without anyone knowing recursion was added to the language. According to Bauer (1978), this was the result of 'the Amsterdam plot in introducing recursivity.'[41]

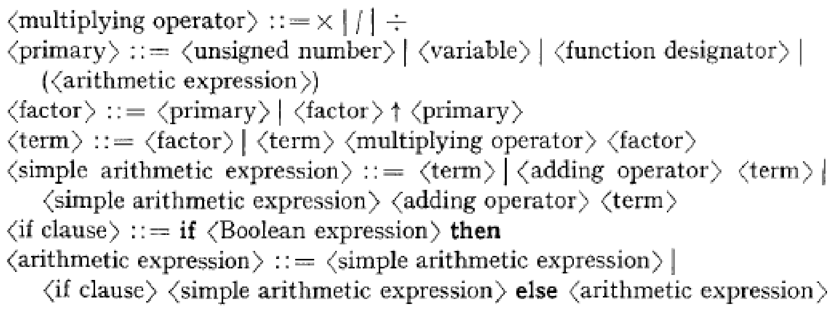
At the end of the meeting the procedure concept had become a clear concept. The formal description of the declaration of a procedure is given in Figure 2.12. As said earlier, Backus (1959) did not define the procedure declarations formally, but he did describe procedure statements (Figure 2.11, page 31). The description of procedure statements in the ALGOL 60 report consists of just six subparts. Backus, on the other hand, needed seventeen subparts for the same task. Using the BNF to define ALGOL 60 had a beneficial effect on the procedure concept and ALGOL 60 as a whole.

```
⟨actual parameter⟩::=⟨string⟩|⟨expression⟩|⟨array identifier⟩|
    ⟨switch identifier⟩|⟨procedure identifier⟩
⟨letter string⟩::=⟨letter⟩|⟨letter string⟩⟨letter⟩
⟨parameter delimeter⟩::=, |)⟨letter string⟩:(
⟨actual parameter list⟩::=⟨actual parameter⟩|
    ⟨actual parameter list⟩⟨parameter delimeter⟩⟨actual parameter⟩
⟨actual parameter part⟩::=⟨empty⟩|(⟨actual parameter list⟩)
⟨procedure statement⟩::=⟨procedure identifier⟩⟨actual parameter part⟩
```

Figure 2.13: The formal description of the ALGOL 60 procedure statement.[43]

Besides the types Boolean, integer, real, and array of those types, the string type was added to ALGOL 60. It could only be used as an actual parameter in procedures, however. Proposals to add extra types like complex numbers and lists, were rejected. Furthermore, IAL's assignment statement was extended into a multiple assignment statement: assignments like a := b := c := 9 were now also allowed.[44]

The compound statement introduced in IAL was extended and it became a special case of the block concept. In a block, variables, functions, and even procedures can be declared local to the scope of the block. These declarations are only known in the block they are declared in and in all subblocks. They are not known, however, in the encapsulating block. A compound statement was now a block without any declarations. Blocks could be nested to any particular level.[45]



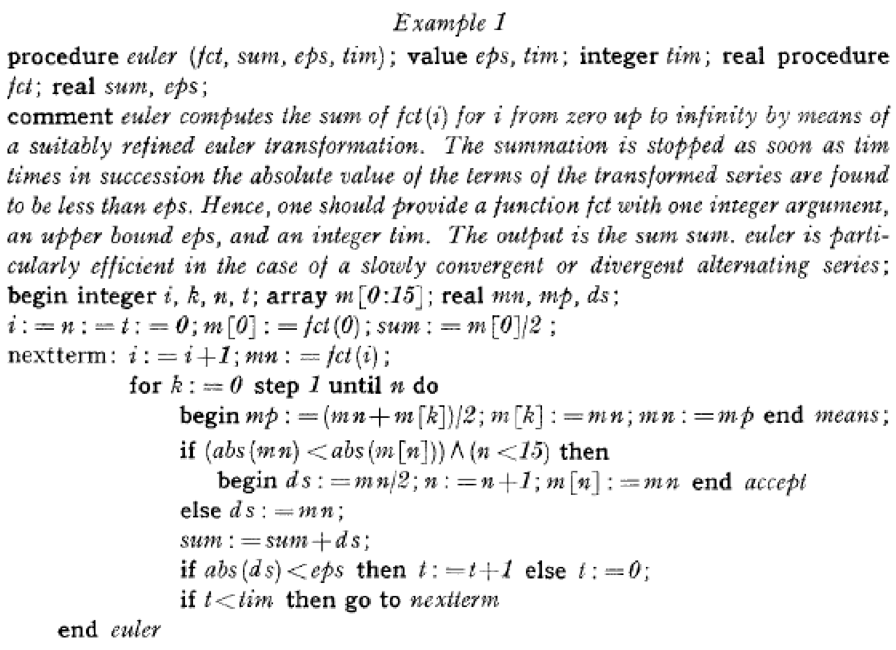
⟨multiplying operator⟩ ::= × | / | ÷
⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨variable⟩ | ⟨function designator⟩ |
    (⟨arithmetic expression⟩)
⟨factor⟩ ::= ⟨primary⟩ | ⟨factor⟩ ↑ ⟨primary⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨multiplying operator⟩ ⟨factor⟩
⟨simple arithmetic expression⟩ ::= ⟨term⟩ | ⟨adding operator⟩ ⟨term⟩ |
    ⟨simple arithmetic expression⟩ ⟨adding operator⟩ ⟨term⟩
⟨if clause⟩ ::= **if** ⟨Boolean expression⟩ **then**
⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ |
    ⟨if clause⟩ ⟨simple arithmetic expression⟩ **else** ⟨arithmetic expression⟩

Figure 2.14: The formal description of the ALGOL 60 arithmetic expressions.[47]

The case-like statement from IAL was removed and replaced by the else clause in the **if** statement and conditional arithmetical and Boolean expressions.[48] In Figure 2.14 the description of the arithmetic expression is given, including this conditional expression. An example of the use of a conditional expression is: a := **if** (b < 23) **then** b + 23 **else** b − 21;. To the variable a the value b + 23 is assigned if b < 23, else the value b − 21 is assigned.

In Figure 2.15 one of the example programs in the ALGOL 60 report is given exemplifying many of the programming language concepts discussed above.

The meeting in Paris was attended by Bauer, Naur, Rutishauser, Samelson, Vauquois, van Wijngaarden, and Woodger from Europe and by Backus, Green, Katz, McCarty, Perlis, and Wegstein from the USA. The seventh American member, Turanski, had died even before the meeting and was not present. According to Perlis (1978), 'The meetings were exhausting, interminable, and exhilarating. (...) diligence persisted during the entire period, The chemistry of the 13 was excellent. (...) Progress was steady and the



```
                              Example 1
procedure euler (fct, sum, eps, tim); value eps, tim; integer tim; real procedure
fct; real sum, eps;
comment euler computes the sum of fct(i) for i from zero up to infinity by means of
a suitably refined euler transformation. The summation is stopped as soon as tim
times in succession the absolute value of the terms of the transformed series are found
to be less than eps. Hence, one should provide a function fct with one integer argument,
an upper bound eps, and an integer tim. The output is the sum sum. euler is parti-
cularly efficient in the case of a slowly convergent or divergent alternating series;
begin integer i, k, n, t; array m[0:15]; real mn, mp, ds;
i := n := t := 0; m[0] := fct(0); sum := m[0]/2 ;
nextterm: i := i+1; mn := fct(i);
          for k := 0 step 1 until n do
              begin mp := (mn+m[k])/2; m[k] := mn; mn := mp end means;
              if (abs(mn) < abs(m[n])) ∧ (n < 15) then
                 begin ds := mn/2; n := n+1; m[n] := mn end accept
              else ds := mn;
              sum := sum+ds;
              if abs(ds) < eps then t := t+1 else t := 0;
              if t < tim then go to nextterm
    end euler
```

Figure 2.15: The first example program in the ALGOL 60 report.[49]

output, Algol60, was more racehorse than camel.'[50]

The difference between IAL and ALGOL 60 was huge. Instead of 'just adding a few corrections to ALGOL 58, it was necessary to redesign the language from the bottom up.'[51] In addition, the editor of the ALGOL 60 report, Peter Naur, used a changed version of Backus's notation. The use of the BNF was beneficial for the clean structure of the report. Peter Naur became the editor because he had prepared a draft of the language for the Paris meeting. According to Bauer (1978), the participants of the conference were surprised by Naur's work. 'It therefor sounds poetic if he has written that his draft Report was 'chosen' as the basis of the discussion; the Committee was simply forced to do so'.[52]

The result of the meeting, the ALGOL 60 report, 'was a fitting display for the language. Nicely organised, tantalisingly incomplete, slightly ambiguous, difficult to read, consistent in format, and brief, it was a perfect canvas for a language that possessed those same properties. Like the Bible it was meant, not merely to be read, but to be interpreted.'[53] And although the ALGOL 60 report and the formal notation used in it were frightening many people at first, the ALGOL 60 report would become the standard for defining programming languages.

2.4 Conclusion

A formal notation to define a programming language is important to allow everyone to read and interpret the definition in the same way. During the development of ALGOL 60, the notation used to describe the ALGOL languages changed fundamentally.

The notation to describe early programming languages like FORTRAN and IAL was natural language combined with some patterns denoting the form of the various language elements. The disadvantage of this notation was that it resulted in ambiguous descriptions. Even for simple language elements, like numbers, expressions, and simple control structures, this was problematic. For definition of complex structures, like the procedure statement and declarations, the notation was totally insufficient. The description of IAL's procedure concepts was long and incomplete because of the use of the "$\cdots$" symbol.

To give a more formal and complete description of the syntax of IAL, Backus invented a new notation: the Backus Normal Form. Using this simple notation, complex structures in the language could be described formally. Unfortunately, the procedure concept of IAL was too complex to describe in this notation. So we can conclude that either the notation was insufficient or the procedure concept in itself was wrong.

During the period between the definition of IAL and ALGOL 60, many proposals to improve ALGOL were made. One of the main topics, especially in Europe, was the complex procedure concept. In the ALGOL 60 report the procedure concept was simplified: input and output parameters were removed, call-by-name and call-by-value parameters introduced. Another important aspect of the new language was the notion of a block with its own scope. This block was an extension of the compound statement from IAL. Recursion was a new feature added without anybody knowing it; the proposal to add recursion explicitly was rejected.

The ALGOL 60 report was edited by Peter Naur. He wrote the draft version and used a slightly modified version of Backus' notation to describe the language. This draft was used as the basis for the ALGOL meeting. The final report would become the standard method of defining programming languages and Backus' notation became the standard method to describe the syntax of programming languages.

In this short period, the ALGOL effort became a major contributor to the field of programming languages. Examples of contributions include the BNF, a method to define a programming language, the block structure, recursive procedures, call by name, call by value, the block, the scope, etc.

2.5 Notes

[0]In this section, FORTRAN should be read as FORTRAN I as described in Section 1.2.2.

[1]IBM, 'Preliminary Report FORTRAN'

[2]J.W. Backus et al., 'The FORTRAN Automatic Coding System for the IBM 704 EDPM : Programmer's Reference Manual', Technical report (Applied Science Division and Programming Research Department, International Business Machines Corporation, 1956), ⟨URL: http://community.computerhistory.org/scc/projects/FORTRAN/704_FortranProgRefMan_Oct56.pdf⟩, p. 1

[3]IBM, 'Preliminary Report FORTRAN', p. 3

[4]Perlis and Samelson, 'Preliminary report: IAL'

[5]ibid., p. 11

[6]ibid.

[7]ibid., p. 12

[8]IBM, 'Preliminary Report FORTRAN', p. 6

[9]ibid.

[10]Perlis and Samelson, 'Preliminary report: IAL', p. 14

[11]IBM, 'Preliminary Report FORTRAN', p. 14

[12]ibid.

[13]Perlis and Samelson, 'Preliminary report: IAL', p. 12

[14]ibid., pp. 13–14

[15]ibid., p. 19

[16]John W. Backus, *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.*, in: *IFIP Congress* (1959), p. 129

[17]ibid.

[18]ibid.

[19]Backus, 'Programming in America in the 1950s – Some Personal Impressions', p. 132

[20]Donald E. Knuth, 'backus normal form vs. Backus Naur form', *Commun. ACM* 7:12 (1964), p. 736

[21]Backus, 'The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.', p. 129

[22]ibid.

[23]ibid., p. 130

[24]ibid.

[25]ibid.

[26]Backus, 'Programming in America in the 1950s – Some Personal Impressions', p. 133

[27]Knuth, 'backus normal form vs. Backus Naur form', p. 736

[28]Backus, 'Programming in America in the 1950s – Some Personal Impressions', p. 133

[29]ibid.

[30]Backus, 'The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.', p. 130

[31]Perlis, 'The American side of the development of Algol', p. 6

[32]Idem, *Transcripts of Presentations*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), p. 144

[33]Idem, 'The American side of the development of Algol', p. 9

[34]E. T. Irons and F. S. Acton, 'A proposed interpretation in ALGOL', *Commun. ACM* 2:12 (1959)

[35]ibid., p. 14

[36]Naur, ‘Transcripts of Presentations’, p. 147

[37]ibid., pp. 157–158

[38]J. W. Backus et al., ‘Report on the algorithmic language ALGOL 60’, *Numerische Mathematik* 2:1 (1960), pp. 128–129

[39]ibid.

[40]Perlis, ‘Transcripts of Presentations’, p. 159

[41]ibid., p. 160

[42]Backus et al., ‘Report on ALGOL 60’, p. 124

[43]ibid.

[44]Perlis, ‘The American side of the development of Algol’, p. 10

[45]Sammet, *Programming languages : history and fundamentals*, p. 193

[46]Backus et al., ‘Report on ALGOL 60’, p. 114

[47]ibid.

[48]Peter Naur, *The European side of the last phase of the development of ALGOL 60*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), p. 23

[49]Backus et al., ‘Report on ALGOL 60’, pp. 130–131

[50]Perlis, ‘The American side of the development of Algol’, pp. 11–12

[51]Rutishauser, *Description of ALGOL 60*, p. 7

[52]As remarked by Bauer in: Naur, ‘European side of development of ALGOL 60’, p. 41

[53]Perlis, ‘The American side of the development of Algol’, p. 12

3 Translation
From Craftsmanship to Science

From craftsmanship to science • With ALGOL as a catalyst • Starting with sequential formula translation • Implementing procedures and recursion • And the influential structure of ALGOL 60

3.0 From craftsmanship to science

It is easy to point to ALGOL 60 and herald this language as the main stimulus for the emerging field of translator writing in computer science. Is this image of ALGOL 60 and the ALGOL effort as the catalyst to the development of the field of translator writing true? And if so, what part of the ALGOL effort was the most influential, and why?

In the 1960s ALGOL 60 was, especially in the academic world, an influential language. In the same period the field of translator writing was developing with big leaps. A chronological order, however, does not mean a causal order per sé. More importantly, the field of translator writing was not new. As mentioned before (see Section 1.1), there was already experience with the implementation of algebraic languages. This experience was, on the other hand, more practically oriented and lacked a sound theoretical fundament.

Most of these early efforts to create and implement algebraic languages were isolated efforts. Many different research groups and corporations were, independently of each other, creating their own languages. Publication of the results was rare, using each other's work even more rare. Of course, there was communication between the different groups at symposia and conferences, for example. This contact, however, did not result in the development of a common theory of translation. The field of translator writing was a field of best practices and craftsmanship.

In the early 1960s, this situation would change drasticly. An article written by F.L. Bauer and K. Samelson, *Sequential formula translation*[0], about a translation technique used in their IAL translator was widely read and was often cited in subsequent articles on the translation of ALGOL 60 and translation of algebraic languages in general. It seemed that this article was the one sheep that was followed by many others.

One of the first to cite and use Bauer and Samelson's article was E.W. Dijkstra in his 1960 article *Recursive Programming*[1]. In this article, Dijkstra presented a solution for the implementation of one of the most controversial features of ALGOL 60: recursive procedures. He had developed this solution while he was working, together with J.A. Zonneveld, on an ALGOL 60 translator for the Electrologica X1 at the Mathematical Centre in Amsterdam.

This translator was important because it was the first complete ALGOL 60 translator, only the use of the **own** concepts was somewhat restricted.[2] In August 1960, ten months after the start of the project the translator was put into use.[3] Only some months after the publication of the ALGOL 60 report, Dijkstra and Zonneveld proved that ALGOL 60 could be implemented in an efficient way.

In 1961, Dijkstra published two articles on the making of this translator: *An ALGOL 60 Translator for the X1*[4] and *Making a Translator for ALGOL 60*.[5] These article were also often cited and used by others. In these years, publication and referring to each other's work became more and more the norm.

In 1962, the lack of a unifying theory in the field of translator writing would be solved by Ginsburg and Rice in *Two Families of Languages Related to ALGOL*[6]. In this article they connected a theory about different types of languages from theoretical linguistics with programming languages. From then on, formal languages became an intrinsic part of the field of translator writing. Eventually, theory, practice, experience, and research would lead to the development of powerful translation techniques and tools to do much of the work that, in the early 1960s, costed several man-years to complete.

In these years, the field of translator writing became a scientific field, and in this chapter, it is argued that the ALGOL effort was the catalyst to this transformation. To that end, the translation of algebraic expressions, in particular the technique pulished by Bauer and Samelson, is discussed first because it was the basis for both the language IAL and the development of many ALGOL translators.

Given this translation technique, basically three approaches to the translation of ALGOL 60 were taken: extending the technique, improving the technique, and, unrelated to the technique, developing other translation techniques. First, the approach of extending the technique is treated with the description of two articles about the implementation of procedures in ALGOL. With the treatment of procedures the most intriguing aspects of the ALGOL language are discussed: the procedure concept as such, recursion, call-by-value parameters, and call-by-name parameters.

The other two approaches towards the translation of ALGOL 60 are treated together because both were based on the structure of the definition of ALGOL 60 or the structure of the BNF. Some articles on the translation of ALGOL 60, published before 1962, are discussed in a more or less chronological order to give a good view of these approaches and the evolving field of translator writing.

Of course, after the publication of *Two Families of Languages Related to ALGOL* by Ginsburg and Rice (1962) there were many more developments in the field of translator writing. The focus in the field, however, shifted from translating ALGOL to translating ALGOL-like languages and even to the general problem of translation. Nonetheless, ALGOL 60 would often be the prime example or the first application of certain techniques and theories. And, long after 1962, ALGOL 60 translators were being build and improvements in techniques were being made. The field of translator writing stood on its own, free from the ALGOL effort and for the topic of this chapter, these later developments related to the translation of ALGOL are, though interesting, less relevant.

3.1 Sequential formula translation

During the 1950s, many algorithmic languages were invented and implemented. Eventually, a more general technique to translate algebraic formulae was developed independently by different people.[7] Due to the isolated nature of these early efforts and the lack of publications,[8] this technique was not widely known.

In March 1959, J.H. Wegstein was the first to publish this technique in a scientific journal. In *From formulas to computer oriented language*[11], Wegstein described the translation technique in less than three pages containing two tables and a huge flow-diagram (see Figure 3.0). This article did not gain much attention, probably due to the date of publication.

Some months later, in November 1959, Bauer and Samelson published their version of the technique as *Sequentielle Formelübersetzung*[12] in *Elektronische Rechenanlagen*, a new German scientific journal. Although this article was widely read in Europe, it would be the translated version published in early 1960 in the *Communications of the ACM* which became truly influential. Their *Sequential formula translation*[13] was referred to in almost all publications related to the translation of ALGOL and translation in general in the early 1960s.

As Bauer and Samelson were not the first to publish nor the only ones to

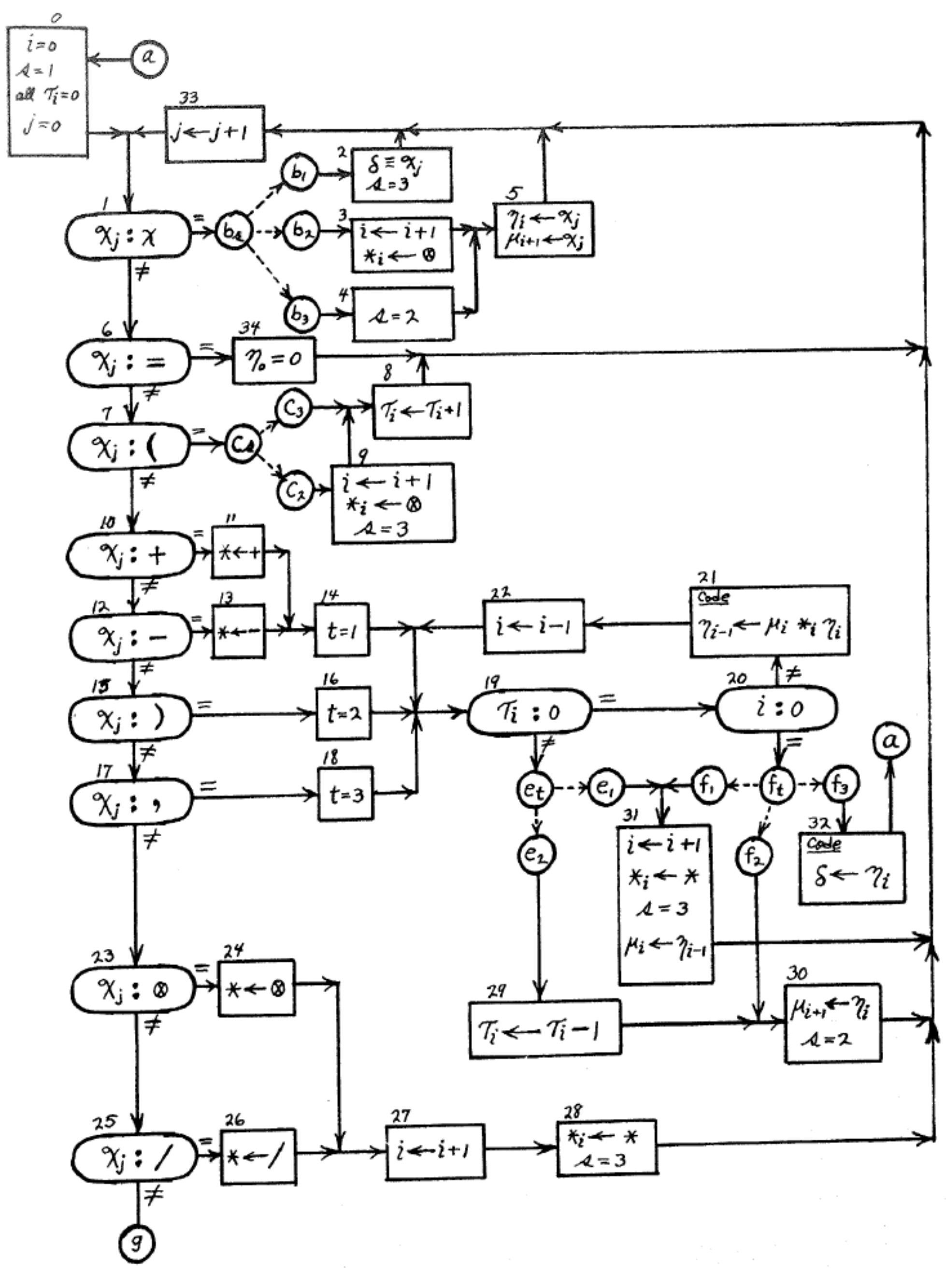


Figure 3.0: Wegstein's implementation of the sequential translation technique described in a flow-diagram.[10]

develop this translation technique, the question arises: Why did their article and translation technique gain so much more attention and why was it so influential? Before answering this question, it must be said that only Bauer himself has written some historical papers about his and Samelson's work on this translation technique and the developments leading to the publication of it. Unfortunately, no one else has written about Bauer and Samelson's work nor about any of the other people developing a similar technique. Nonetheless, Bauer's historical publications on this translation technique are, if used with care, an opportunity to describe the development of this technique in more detail.

3.1.0 The development of the sequential formula translation technique

Based on their early work on the STANISLAUS, a device to check a valuation of a propositional formula, Bauer and Samelson started in 1955 on a technique to translate arithmetic formulae.[14] They worked this technique into a design for a simple calculator (See Figure 3.1)[15], as we will now describe.

Formulae can be entered into the calculator using a keyboard. Numbers and operators are separated. The former go directly into the number stack (Numbers Cellar). Incoming operators are compared to the topmost operator on the operator stack (Operations Cellar) using the precedence orders in Figure 3.2: If the order of the incoming operator is lower than the order of the operator on the stack then the new operator is pushed onto the stack; If the orders are equal, the incoming operator is evaluated by the computer; If the order of the incoming symbol is greater than the operator on the stack, the latter is evaluated by the computer.

The computer component performs the evaluation of an operator using the topmost numbers on the number stack and the result is pushed back onto the number stack. When the end of an expression is reached, the result of the evaluation of the expression is available as the topmost element of the number stack and can then be printed.

As an example, the evaluation of the expression $(3 + 5) \times (-2 + 4)$ is described in Figure 3.3. Initially, the number stack contains a 0 for the purpose of computing the unary operators $+$ and $-$. If the number stack becomes empty during a computation, the number 0 is again pushed onto the stack for the same reason. The evaluation of the expression is performed sequentially, that is, like "how the expression is read", thus from left to right. The state of the calculator is given by the contents of the number stack, the operator stack and the current input symbol. The symbol $\perp$ denotes the end

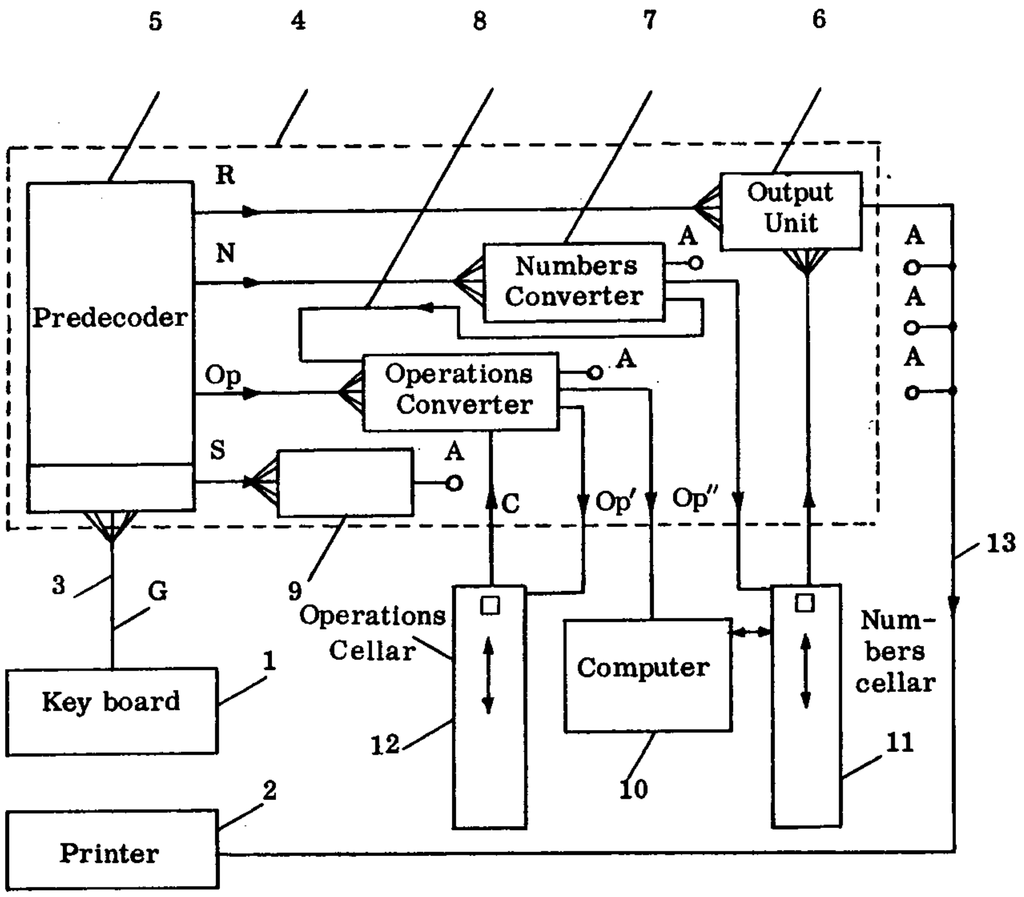


Figure 3.1: A diagram of the simple calculator designed by Bauer and Samelson in the 1950s.[17] They called a stack a *cellar*.

order	*incoming operator*	*topmost operator*
0	(	
1	$\times, \div$	$\times, \div$
2	$+, -$	$+, -$
3	)	(

Figure 3.2: The precedence orders used in the simple calculator.[18]

of the stack.

expression	(	3	+	5	)		×	(	−	2	+	4	)			
operation stack	⊥	(	(	+	+	(	⊥	×	(	−	−	+	+	(	×	⊥
		⊥	⊥	(	(	⊥		⊥	×	(	(	(	(	×	⊥	
				⊥	⊥				⊥	×	×	×	×	⊥		
										⊥	⊥	⊥	⊥			
number stack	0	0	3	3	5	8	8	8	0	0	2	−2	4	2	2	16
	⊥	⊥	⊥	⊥	3	⊥	⊥	⊥	8	8	0	8	−2	8	8	⊥
					⊥				⊥	⊥	8	⊥	8	⊥	⊥	
											⊥		⊥			

Figure 3.3: Evaluation of $(3+5)\times(-2+4)$ on the simple calculator developed by Bauer and Samelson.[20]

In 1956, Bauer, Bottenbruch, Rutishauser, and Samelson started a joint project to create an algebraic language and translator for different computing machines in Zürich (ERMETH), München (PERM) and Darmstadt, forming the so called ZMD group. Later, when Bauer and Samelson moved to Mainz (Z22), this group became known as the ZMMD group.[21] The first GAMM proposal for the international algebraic language was based on the technique used in the simple calculator described above. At the Zürich meeting, Bottenbruch presented a working compiler for their proposed language. It appeared that the method of translation presented was much more elaborated than the methods known and used by the Americans.[22] As we have seen earlier, however, Wegstein did know this technique, as did others. The question is: Did he get the idea of the technique on this meeting or was it developed independently by himself? Given the two totally different descriptions of the same technique, it is reasonable that he indeed did develop the same technique independently from Bauer and Samelson.

The origin of the block structure in ALGOL can also be found in the ZMMD effort. It was proposed by Samelson because it was a natural extension of the stack principle of their translation technique.[23] Unfortunately, the compromise resulting from the Zürich meeting was less focused on the stack principle.[24] In ALGOL 60, however, the block structure would be one of the basic elements of the language, but now not originating from the stack principle, but from the highly structured definition of the language created by Peter Naur.

After the definition and publication of IAL, the ZMMD group adapted their compilers. Already in 1958, they had working translators for their various machines.[25] In June 1959, the ICIP conference took place in Paris where Bauer and Samelson presented their translation technique. After this presentation, the ZMMD group expanded into the ALgol COnverteR group

(ALCOR). Computer manufactures, research centres and universities from Europe and even one from the USA joined the core group in Germany. All the members got the same information about the ZMMD translators and started working on their own compilers.[26]

Meanwhile, Bauer and Samelson published their technique in the *Communications of the ACM*. The explanation for the influence of this article is twofold. First of all, the translation technique described was the basis of Bauer and Samelson's proposal for IAL and later ALGOL. As a result, this technique was extremely suitable for translating ALGOL.

The second part of the explanation lies in the moment of publication. It was published some months before the publication of the ALGOL 60 report. This report generated much interest in ALGOL and as a result in the translation of ALGOL. People starting writing their own translators for ALGOL 60 had a suitable basis to start from: the technique described in Bauer and Samelson's article.

3.1.1 Sequental formula translation explained

In *Sequential Formula Translation*, Bauer and Samelson (1960) described the translation technique used by the ALCOR group for their IAL translators. The basis of the technique was still the same as in the design of the simple calculator (see above): incoming symbols are postponed till the first moment they can be evaluated, or in this case, translated. Incoming operators are compared with the topmost operator in the stack using a matrix defining the action to be taken given an incoming symbol and a topmost symbol. The main difference between the design of the calculator and the technique described in this article was that the calculator was to be a device and the translator was a computer program for a general purpose computer.

The technique is explained with an example. In Figure 3.4 the algebraic expression $(a \times b + c \times d)/(a - d) + b \times c$ is translated. The translator uses a symbol stack, initially it is empty. The program that is generated uses a number stack to calculate (subparts of) the expression. In this example, the generated program is an ALGOL 60 program, normally it would be an assembly program. **array** N is the number stack. Instead of using a stack pointer, the values of that stack pointer are used directly for explanatory purposes. Otherwise, operations on the stack would look like N[n] := b; and the stack pointer (here n) would be incremented and decremented when an element is pushed onto the stack or removed from the stack, respectively.

For simple arithmetic expressions this method was clear, however, IAL consisted of more than just simple arithmetic expressions. Other parts of IAL were translated in a similar way: postponing symbols till they can be

symbol stack	next symbol	resulting program
[]	(	
[(]	a	N[0] := a;
[(]	×	
[(, ×]	b	N[1] := b;
[(, ×]	+	N[0] := N[0] × N[1];
[(, +]	c	N[1] := c;
[(, +]	×	
[(, +, ×]	d	N[2] := d;
[(, +, ×]	)	N[1] := N[1] × N[2];
[(, +]		N[0] := N[0] + N[1];
[(]		
[]	/	
[/]	(	
[/, (]	a	N[1] := a;
[/, (]	−	
[/, (, −]	d	N[2] := d;
[/, (, −]	)	N[1] := N[1] − N[2];
[/, (]		N[0] := N[0] ÷ N[1];
[/]		
[]	+	
[+]	b	N[1] := b;
[+, ×]	×	
[+, ×]	c	N[2] := c;
[+]	⊥	N[1] := N[1] × N[2];
[+]		N[0] := N[0] + N[1];
[]		

Figure 3.4: The translation of $(a \times b + c \times d)/(a - d) + b \times c\perp$. This example is taken from Bauer and Samelson's article and is slightly adapted to make it more understandable.[28]

evaluated and translated. For example, the brackets delimiting a compound statement, **begin** and **end** were treated like arithmetic parenthesis. The **end** keyword takes care of evaluating all the remaining symbols on the stack till the **begin** symbol is reached.

More problematic were the **if** and **for** statements. Remember, Bauer and Samelson's article was about translating IAL: the **if** statement is like **if** B; S and the **for** statement like **for** v := L; S. The ; is the syntactical end of the statement, but it is not the semantical end: effectively, S belongs to the statement.

In the translation of **if** $B;_0S;_1$ the **if** symbol is removed from the stack on the encounter of $;_0$. The target address for the case B = **true** is known at this point in the translation, however, the target address for the case B = **false** is still unknown. To solve this problem, an auxiliary **if** symbol is introduced during the translation which will be removed on encounter of $;_1$ when the target address for the case B = **false** is known. In other words, during the translation $;_0$ is treated as a kind of **then** symbol.

for v := L^0, L^1, $\cdots$, L^n; S
is translated into:
v '[0] := L^0; v1[1] := L^1; $\cdots$; v'[n] := L^n;
for i := 1 (1) k; **begin** v := v'[i]; S **end**

Figure 3.5: Translation of a list-of-values variant **for** statement.

The list-of-values variant **for** statement was translated into equivalent substatements (see Figure 3.5). The translation of a start-step-end-variant **for** statement goes as in Figure 3.6. Of course, if the step is negative, the **if** statement would test the opposite. In case the sign of the step is unknown at compile time, it has to be decided at run time, and the code to decide that, is added too.

The translation of other elements in the IAL language, like declarations and procedures, were treated as straightforward. For example, procedures were translated as mere subroutines. One element of IAL, however, did get more attention: the **array**. The translation of subscripted variables was not simple. Although Bauer and Samelson treated only static arrays, the computing of the addresses for these subscripted variables needed much explanation. Actually, memory management was one of the hot items in the field of translator writing in these days. The translation of ALGOL 60, especially the implementation of recursion, procedures, and dynamic sized arrays, stimulated the development of dynamic memory management.

for $\mathsf{v} := \mathsf{E}^i$ (E^s) E^e; S
where E^s is positive, is translated into:
$\mathsf{v} := \mathsf{E}^i$; $\mathsf{s} := \mathsf{E}^s$; $\mathsf{e} := \mathsf{E}^e$;
L_S: S; $\mathsf{v} := \mathsf{v} + \mathsf{s}$; **if** $\mathsf{v} \leq \mathsf{e}$; **goto** L_S; $\mathsf{v} := \mathsf{v} - \mathsf{s}$;

Figure 3.6: Translation of a start-step-end variant **for** statement.

3.2 Implementing procedures and recursion

3.2.0 "Solving" by ignoring

The publication of the ALGOL 60 report was not the start of implementation of ALGOL languages. For example, in the USA different projects were started to implement algebraic languages, like MAD, NELIAC, and JOVIAL. All these languages were based on IAL, that is, it was not an implementation of IAL itself, but IAL was used as a set of guidelines for their own algebraic languages. In Europe, as said earlier, the ZMMD group was building translators for their proposals for IAL and later for the final draft of IAL too.

Although there was experience with translating IAL-like languages, the translation of ALGOL 60 was different from the translation of IAL in two important ways. First of all, there were some elements of ALGOL 60 which were new and these elements were not seen as useful or understood completely.[29] In particular recursion, the procedure concept, dynamic sized arrays and **own** variables were problematic and led to the development of dynamic storage allocation.

Second, the amount of interest generated by ALGOL 60 was much bigger than generated by IAL. Many more implementations were started after the publication of the ALGOL 60 report than were started after the publication of the report on IAL.

The ALCOR group started an implementation of ALGOL 60 based on their translator for IAL. Instead of implementing the new programming language concepts, however, they decided to ignore these difficult features. Conditional boolean expressions, conditional designational expressions, the **while** part in the **for** statement, **own** declarations and recursive procedures were not part of the early ALGOL 60 translators of the ALCOR group.

Instead of focussing on implementing the whole language the ALCOR group tried to maximise intercompatibility between the computers of the members of the ALCOR group. All members used the same hardware representation and most of them also used the same translation technique as described above. The translation matrix controlling the method was given to all the members to assure that the same semantics were implemented at all the different installations. As a result, programs written for one member's computer could run on all member's computers.[30] Later, their translators became more feature complete, but the pioneers of implementing these features were working outside the ALCOR group.

Ignoring difficult or useless features was not uncommon. According to a questionaire in the *ALGOL Bulletin* no. 9, the **own** declaration was almost always omitted.[31] Outside the ALCOR group, more than half of the compilers did not include variable size arrays.[32] Interestingly, all the non-ALCOR compilers in this questionaire did include procedures.[33] Either they did not mention recursion or it was fully implemented.

Although recursion was not implemented by the ALCOR group, it was one of the new interesting and controversial concepts in ALGOL 60. As said before, many people did not see why it would be useful at all. Others, however, did see the theoretical importance of the concept.[34] In 1960, two articles on the implementation of recursive procedures were published.

3.2.1 Dijkstra's *Recursive Programming*

The first and most influential article of the two was *Recursive Programming*[35] by E.W. Dijkstra. It was published in the second issue of *Numerische Mathematik* in 1960. As said before, his implementation of recursive procedures was developed during the development and implementation of the ALGOL 60 compiler for the Electrologica X1 at the Mathematical Centre in Amsterdam.

Dijkstra based his method of implementing recursion on the translation technique of Bauer and Samelson because it was 'well known (...) and so elegant that we could not refrain from trying to extend this technique by consistent application of its principles.'[36] Actually, he made the observation that it does not matter if an simple expression is evaluated or that an expression containing a procedure call is evaluated. Both evaluations will use the next free places on the stack to compute the resulting value. Hence, procedures can be treated like expressions with keeping some extra administration.

The translator uses the stack for computing intermediate results only. Values needed are pushed onto the stack and are used when a subexpression can be computed. Then, the values used are removed and replaced by the value of the expression. However, when a procedure is called the parameters and local variables of the procedure are pushed onto the stack. Because these elements must be accessible later, access to elements deeper down in the stack is needed. For this reason a more elaborated stack was needed instead of the simple stack used for calculating simple expressions.

The program text is stored in memory, instruction after instruction this program is executed, using the stack to compute intermediate results. When a procedure is called, the normal flow of instructions changes because the static program text of the procedure is located somewhere else in memory. Furthermore, the dynamic part of the procedure is pushed onto the stack. This dynamic part consists of the parameters, the local variables and the link data. This link data defines the state of the execution of the program in which the procedure is called, and is used to return back to the normal flow of the program where it was when the procedure was called.

The execution of the procedure program text also uses the stack in the same way the main program did: for computing intermediate results and for calling procedures. This execution starts on the first free place on the stack, directly after the places where the parameters and local variables of the procedure are stored. If another procedure is called, the same thing happens, the current situation is stored as part of the call and is pushed onto the stack with the dynamic data of the new procedure. This procedure is then executed. When the execution ends, the old situation of the previous procedure is restored, and execution of that older procedure goes on. Eventually, this procedure is also finished, the state of the main program is restored and execution of the program goes on. Of course, more than two procedures can be called, even recursively.

As part of the link data, a block number is stored to be able to access global variables with respect to this block. Initially, the block number is zero. In addition, two parameter pointers, a return address and the stack pointer are stored as part of the link as well. This approach was also used to execute blocks. These were treated as procedures without any parameters or return value.

3.2.2 The solution of Irons and Feurzeig

Another way of implementing procedures was published in 1960 by E.T. Irons and W. Feurzeig in *Comments on the Implementation of Recursive Procedures and Blocks in Algol-60*.[37] Where Dijkstra only used text to describe his method of implementing recursive procedures, Irons and Feurzeig were also using flow diagrams to exemplify things. They treated blocks, procedures, and the **goto** statement together because these concepts change the normal flow of execution of a program. Again, the translation of both procedures and blocks are more or less similar. The way of handling blocks is just somewhat simpler because less information has to be checked and stored.

The procedures and blocks are handled by special entrance (see Figure

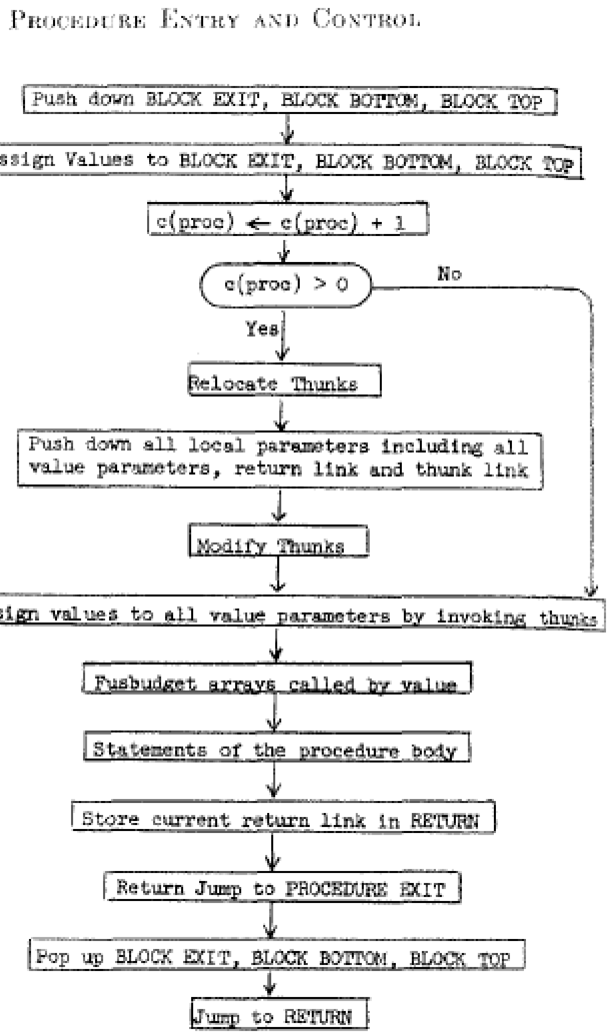

PROCEDURE ENTRY AND CONTROL
Push down BLOCK EXIT, BLOCK BOTTOM, BLOCK TOP
Assign Values to BLOCK EXIT, BLOCK BOTTOM, BLOCK TOP
c(proc) ← c(proc) + 1
c(proc) > 0
No
Yes
Relocate Thunks
Push down all local parameters including all value parameters, return link and thunk link
Modify Thunks
Assign values to all value parameters by invoking thunks
Fusbudget arrays called by value
Statements of the procedure body
Store current return link in RETURN
Return Jump to PROCEDURE EXIT
Pop up BLOCK EXIT, BLOCK BOTTOM, BLOCK TOP
Jump to RETURN

Figure 3.7: The flow diagram for procedure entry and control in the article of Feurzeig and Irons about recursive procedures.[39]

3.7) and exit (see Figure 3.8) routines which are executed by entering a block (or procedure) and by exiting them. Procedures are treated as blocks where the return link and thunk link are stored as local variables. Thunks are a way to get the address of a parameter into a standard location, regardless the parameter type.[40] Interestingly, J. Jensen and Peter Naur (1961) give in *An implementation of ALGOL 60 procedures*[41] a more or less similar implementation of handling different parameters in procedures.

Furthermore, three pointers are used to point to the current active block: block exit points to the exit routine, block bottom points to the lowest memory address of the active block and block top points to the highest memory address of the active block. These pointers are comparable to the procedure pointers in the link data used by Dijkstra.

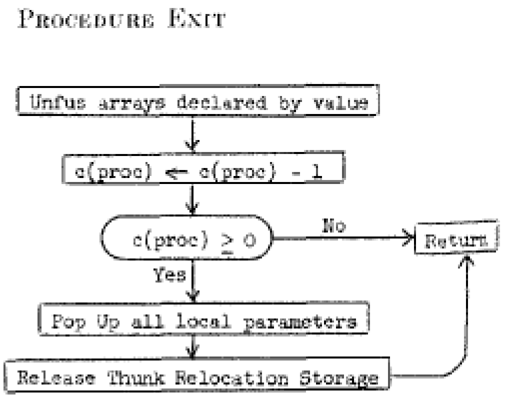

PROCEDURE EXIT
Unfus arrays declared by value
c(proc) ← c(proc) - 1
c(proc) ≥ 0
No
Return
Yes
Pop Up all local parameters
Release Thunk Relocation Storage

Figure 3.8: The flow diagram for procedure exit in the article of Feurzeig and Irons about recursive procedures.[43]

Instead of a block number, however, there is a recursion counter. For every procedure c(proc) denotes the recursion depth. Initially, the recursion counter is −1. Because procedures can be recursive and blocks can not, this counter is used for procedures only.

Both methods of implementing recursion were very similar. When a block is executed, the current situation is stored, local variables, including parameters, are pushed onto the stack and the exectution of the block starts. By the end of the execution, the old situation is restored. As for the handling of procedure parameters, here too, both Jensen and Naur's, and Ingerman's method are very similar. The extensions to the technique to implement the more difficult concepts in ALGOL 60 were more or less straightforward and fitted easily into the technique of sequential formula translation.

3.3 The influence of ALGOL's structure

Besides extending the common translation technique, two other approaches to the translation of ALGOL were undertaken. These approaches, improving the existing translation techniques and inventing new translation techniques, are now discussed.

Although most of the efforts in these approaches started differently the focus was the same: on the structure of the language or the structure of the notation used to describe the language. The highly structured definition of ALGOL 60 in the ALGOL 60 report combined with the use of the BNF caused a growth in interest in syntax of programming languages and the structure of translators. As a result, several syntax directed translators and translation techniques were developed.

The ideal was a program that, given a definition of a language in a metalanguage like BNF extended with some semantical information, could generate a translator. One such a translator was developed by Brooker and Morris for the ATLAS computer. This 'Assembly Program for a Phrase Structure Language'[44][45], however, was developed outside of the ALGOL effort. Nonetheless, Brooker and Morris may have influenced some of the scientists discussed below because of the many publications on their translation program.

In January 1961, the issue of the *Communications of the ACM* was dedicated to the translation of algorithmic languages. From the seven articles on the translation of ALGOL, two were on the translation of ALGOL in relation to the structure of ALGOL 60 itself: *Recursive Processes and ALGOL Translation*[46] by A.A. Grau and *A Syntax Directed Compiler for ALGOL 60* by E.T. Irons.

3.3.0 Grau's recursive translation technique

In his article, A.A. Grau improved the translation technique described by Bauer and Samelson by reducing the size of the translation matrix. This matrix, as said earlier, defined the action to be taken given an incoming symbol and the topmost symbol on the stack. Because ALGOL 60 consisted of many symbols this matrix was large. Given the small amount of memory computing machines were equipped with in those days, this was problematic, especially for the many small scale computers in use in Europe.

An important aspect of ALGOL's description was its recursive definition. Grau tried to reflect this recursive structure on the translator: if there is a statement, it should be translated by a statement procedure, a expression by an expression procedure, etc. However, because statements can be recursive, a procedure must be able to call itself. So the translator itself should be recursive. In fact, the translator only translates blocks because blocks are the basic elements of an ALGOL 60 program. All other elements are translated by dedicated procedures called by this block procedure.

These recursive procedures are called when the translator reaches a state when the with the procedure corresponding elements are expected. For example, if the translator is in a state where an arithmetic expression is expected, and the next input symbol is an **if** symbol, then the translator pushes, according to the definition of the ALGOL 60 arithmetic conditional expression:

⟨arithmetic expression⟩ ::= **if** ⟨boolean expression⟩ **then** ⟨simple ae⟩
else ⟨arithmetic expression⟩

the next state onto the stack, that is the then-part state and calls the boolean expression procedure. After translating the boolean expression, the top most element of the stack will be this then-part state.

The description of Grau's technique is difficult to understand because Grau (1961) used recursive procedures which were to be implemented using a stack. In his technique the recursive descent parsing technique can, however, already be recognised.

3.3.1 Irons's syntax directed compiler

Another approach to the problem of translating ALGOL, independent of the techniques used in the ALCOR group, was one developed by E.T. Irons and published as *A Syntax Directed Compiler for ALGOL 60*[47] in 1961. Later, in 1963, an extended version of the article was published as *The Structure and Use of the Syntax Directed Compiler*[48]. According to Irons, the problem of early compilers was that the function of translation of the language and the function of definition of the language were mixed into one program. He wanted to separate the two functions by using an extended metalanguage to define a programming language and a general translator program to translate programs written in that newly defined programming language using the definition of that language.

This metalanguage consists of a set of rules defining both syntax and semantics. These rules have the form syntax =:: S $\rightarrow$ {semantics}. S, a syntactic unit, is the subject of the rule. The syntactical part of the rule consists of syntactical units, similar to the BNF. The semantical part is a list of semantical definitions which can have three different forms: symbols in the output language, substitution of output symbols for other output symbols, and functions on output symbols.

The translator works by using definitions, as described above, to translate a program text into the target language. Starting at the bottom, from left to right, syntactic units are recognised and the semantic rules are applied till an topmost element is recognised and no further steps can be taken.

In March 1962, Robert S. Ledley and James B. Wilson published *Automatic-Programming Language Translation Through Syntactical Analysis*[49], describing an approach to the general problem of translation based on the work of Irons (1961). The metalanguage used by Irons was changed a bit: the syntactical element of the rule is put in front, followed by the syntactical part and then the semantical part. In addition, the semantical part was made more understandable. The underlying technique, however, was the same.

3.3.2 Lucas's structure of formula translators

In September 1961, P. Lucas published *The Structure of Formula - Translators*[50] in the *ALGOL Bulletin* supplement number 16. He, as did Grau (1961), based his work on Bauer and Samelson and tried also to improve the technique by reducing the translation matrix. Although the technique was developed while implementing an ALGOL 60 translator[51], the article had a more general claim: it presented a technique for algebraic translators in general.

Using three levels of meta language: the object language, the language used in the ALGOL 60 report, and a generalised version of the same language using Greek letters between double angular brackets, a new language was defined. This third level metalanguage is used to describe patterns of elements in the second level metalanguage. For example, the pattern underlying $< integer >::=< digit > \mid < digit >< integer >$ is $<< \beta >>::=<< \alpha >> \mid << \alpha >><< \beta >>$.

Lucas now distinguished three types of syntactical definitions: enumerations, juxtapositions and combined definitions. In all three groups there are two variations: explicit and recursive definitions. Enumerations have the form $<< \chi >>::=<< \alpha_0 >> \mid << \alpha_1 >> \mid << \alpha_2 >> \mid \cdots << \alpha_n >>$ and form a so called syntactic category. The lefthand side variable can be replaced by one of the variables on the right hand side.

The juxtaposing definition looks like $<< \chi >>::=<< \alpha_0 >><< \alpha_1 >><< \alpha_2 >> \cdots << \alpha_n >>$. The left hand side variable may be replaced by the whole right hand side. The combination group is, of course, a combination of the other two groups.

A definition is explicit if a left hand side variable does not occur in any (sub)part of a right hand side variable. In recursive definitions, however, left hand side variables does occur in the right hand side variables or subparts of

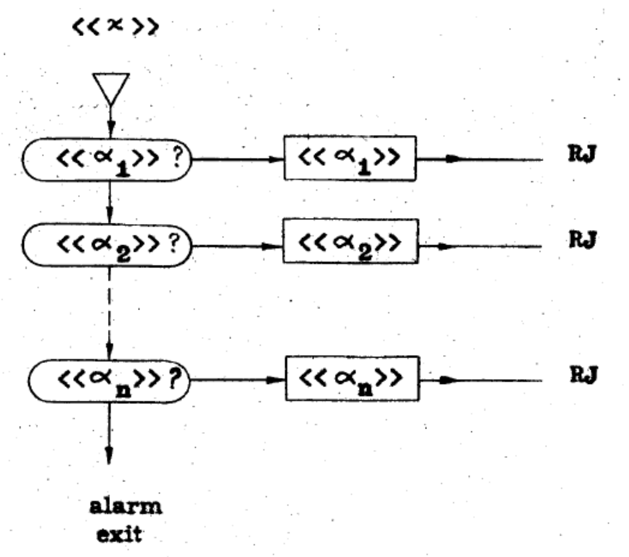


Figure 3.9: The structure of a translation procedure of an enumerating definition.[53]

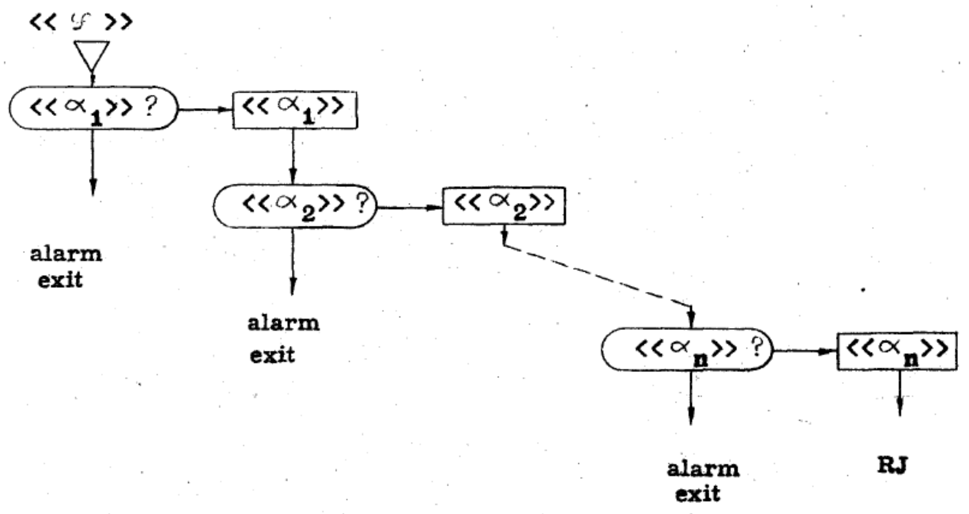


Figure 3.10: The structure of a translation procedure of a juxtaposing definition.[55]

it. Lucas does denote this with $\mathsf{cont}(<< \chi >>, << \alpha >>)$, that is, $<< \alpha >>$ does occur in $<< \chi >>$.

Lucas presented for every group of definitions the structure of the translation procedure. In Figures 3.9 and 3.10 the program structure of respectively enumerating and juxtaposing definitions are presented. A rounded box denotes the question: Is the incoming symbol the element inside the box? If so, that element is translated, indicated by a box with the name of the element in it.

In fact, first is tested if the expected element is indeed found. If so, the translation procedure of that element is called. These two Figures give the explicit structure. If a definition is recursive, the structure changes as expected: a loop is introduced. The whole translator consists of recursive procedures calling each other according to the structure of their definitions. Of course, some extra administration is added to the structure of the procedures. In essence, however, this is the whole technique. Here too, as in the case of Grau (1961), the later recursive descent parsers can already be recognised.

3.3.3 Connecting ALGOL-like languages with context-free language theory

Interestingly, both Grau (1961) and Lucas (1961) arrived at similar ideas when they both try to improve the translation technique of Bauer and Samelson. The same happened in the case of Dijkstra (1960), and Feurzeig and Irons (1960) with implementing procedures and also with Jensen and Naur (1961) and Ingerman (1961) with their implementation of the different types of parameters in procedures. Somehow, independent from each other, the same developments were made. This can be a sign of an immature scientific infrastructure. On the other hand, it can also be a sign of how well the translation technique, theory and language fitted together.

In addition, the terminology and notation used was not consistent throughout the different articles. That is another sign of the evolving field of translator writing: a common basis had to be found. One big step towards a common base was made by Ginsburg and Rice (1962). They wanted to get more knowledge about programming languages because these languages were becoming more and more important for instructing computers. To that end, they abstracted from ALGOL and developed two models of similar programming languages: definable languages and sequentially definably languages.[56] The former are languages defined by the BNF where all left-hand side elements are metalinguistic variables. This set of languages are equal to the type

2 languages defined by Noam Chomsky in 1956: the context-free languages.

As a result, Ginsburg and Rise (1962) not only tried to formalise ALGOL-like languages, they also made the connection between programming languages and theoretical linguistics. This connection turned out to be a fruitful one. The field of formal languages, the definition of programming languages, and translation became a very popular one. Soon after the publication of *Two Families of Languages Related to ALGOL* in 1962, publications in the field of translator writing became more and more theoretical.

With this formalisation the field of programming languages could be fit into the theory of computing and automata theory, developed in the 1940s and 1950s.[57] In other words, the practice gained with programming computers and with programming languages could be based on a theoretical foundation. The field of computing became a real science in the sense that it got its own theoretical component connecting practice and theory: theoretical computer science.

3.4 Conclusion

This chapter started with the question what part of the ALGOL effort was the most influential. The answer lies in the fact that the ALGOL effort was a catalyst for the field of translator writing: it transformed this field from a craftsmanship to a science. This transformation started with the development and publication of a translation technique for IAL by Bauer and Samelson, who initiated the ALGOL effort to construct a translation technique and matching language simultaneously.

The language was made to fit the technique, and vice versa. When Bauer and Samelson, some months before the publication of the ALGOL 60 report, published their technique, it was widely read. Because of the popularity of ALGOL 60 many implementations of ALGOL 60 translators were started, often based upon the technique described by Bauer and Samelson. During this process, the isolated efforts to create an algebraic language became part of a larger effort: the ALGOL effort.

Although there was a translation technique for IAL, some elements of ALGOL 60 requested the extension of the technique (notably the implementation of recursion and procedures by Dijkstra (1960), and by Feurzeig and Irons (1960)). Both groups came to similar solutions, fitting into the well-known translation technique.

Other approaches to the translation of ALGOL were improving the technique and inventing new techniques. Both Grau (1961) and Lucas (1961) improved the technique by tackling the size of the translation matrix. Both arrived to a solution with recursive procedures in which the later recursive descent parsing technique can be recognised.

Irons (1961) invented a new technique of translation based on the structure of ALGOL. From left to right, from bottom to top, his technique was based on a metalanguage in which both syntax and semantics were to be specified. A general translator was then able, with these specifications and a program text, to translate the program into a executable version.

Although these approaches to develop syntax directed translators were different, the lack of a common notation, terminology, or theory was striking. Clearly, the field missed a sound theory to be able to evolve from a craft into a science. Ginsburg and Rice (1962) supplied the field with this theory: they created the connection between Chomsky's (1956) context-free languages and ALGOL-like languages and hence connected the field of programming languages with other theories of computing and automata theory.

The further development of the field of translator writing, although related to ALGOL at first, was no longer ALGOL centred. It became an independent field of research, with an own vitality, evolving freely from the ALGOL effort. But what was the most influential part of the ALGOL effort for the transformation of the field of translator writing? Bauer and Samelson's translation technique or the structured definition of ALGOL 60? Or the whole ALGOL effort as it created a common notation, theory and technique?

3.5 Notes

[0]K. Samelson and F. L. Bauer, 'Sequential formula translation', *Commun. ACM* 3:2 (1960)

[1]E. W. Dijkstra, 'Recursive Programming', *Numerische Mathematik* 2 (oct 1960)

[2]Idem, 'An ALGOL 60 Translator for the X1', in: Richard Goodman, editor, *Annual review in automatic programming 3* (1963), p. 3

[3]F.E.J. Kruseman Aretz, *The Dijkstra-Zonneveld ALGOL 60 compiler for the Electrologica X1* (Amsterdam: CWI, 2003), ⟨URL: http://ftp.cwi.nl/CWIreports/SEN/SEN-N0301.pdf⟩, p. 1

[4]Dijkstra, 'An ALGOL 60 Translator for the X1'

[5]Idem, 'Making a Translator for ALGOL 60', in: Richard Goodman, editor, *Annual review in automatic programming 3* (1963)

[6]Seymour Ginsburg and H. Gordon Rice, 'Two Families of Languages Related to ALGOL', *J. ACM* 9:3 (1962)

[7]Donald E. Knuth, 'A History of Writing Compilers', in: Pollack and W. Bary, editors, *Compiler Techniques* (Princeton, 1972), p. 44

[8]ibid., p. 39

[9]J. H. Wegstein, 'From formulas to computer oriented language', *Commun. ACM* 2:3 (1959), p. 7

[10]ibid.

[11]ibid.

[12]Friedrich L. Bauer and K. Samelson, 'Sequentielle Formelübersetzung', *Elektronische Rechenanlagen* 1:1 (1959)
[13]Samelson and Bauer, 'Sequential formula translation'
[14]F. L. Bauer, 'The Cellar Principle of State Transition and Storage Allocation', *Annals of the History of Computing* 12:1 (1990), p. 43
[15]Samelson and Bauer, 'The ALCOR project', p. 207,217
[16]ibid., p. 208
[17]ibid.
[18]Samelson and Bauer, 'Sequential formula translation', p. 208
[19]Changed example from Samelson and Bauer, 'The ALCOR project', p. 209
[20]Changed example from ibid.
[21]Bauer, 'The Cellar Principle of State Transition and Storage Allocation', p. 46,47
[22]Bauer, 'From the Stack Principle to ALGOL', p. 34
[23]ibid., p. 35
[24]Bauer, 'The Cellar Principle of State Transition and Storage Allocation', p. 47
[25]ibid.
[26]Samelson and Bauer, 'Sequential formula translation', p. 210
[27]ibid., p. 78
[28]ibid.
[29]Samelson and Bauer, 'The ALCOR project', p. 215
[30]L. L. Bumgarner and M. Feliciano, 'ALCOR Group Representation of ALGOL Symbols', *Commun. ACM* 6:10 (1963), p. 597
[31]Peter Naur, 'ALGOL translator characteristics and the progress in translator construction', *ALGOL Bull.* 10 (1960)
[32]ibid.
[33]ibid.
[34]E. T. Irons and W. Feurzeig, 'Comments on the Implementation of Recursive Procedures and Blocks in Algol-60', *ALGOL Bull.* Sup 13.2 (1960), p. 1
[35]Dijkstra, 'Recursive Programming'
[36]ibid., p. 313
[37]Irons and Feurzeig, 'Comments on the Implementation of Recursive Procedures and Blocks in Algol-60'
[38]ibid., p. 4
[39]ibid.
[40]P. Z. Ingerman, 'Thunks: a way of compiling procedure statements with some comments on procedure declarations', *Commun. ACM* 4:1 (1961)
[41]J. Jensen and Peter Naur, 'An Implementation of Algol 60 Procedures. (pre-print from Nordisk Tidskrift for Informations-Behandling, Volume 1, No 1 -1961)', *ALGOL Bull.* Sup 11 (1961)
[42]Irons and Feurzeig, 'Comments on the Implementation of Recursive Procedures and Blocks in Algol-60', p. 5
[43]ibid.
[44]D. Brooker, R. A. and Morris, 'An Assembly Program for a Phrase Structure Language', *The Computer Journal* 3:3 (1960), ⟨URL: http://comjnl.oxfordjournals.org/cgi/content/abstract/3/3/168⟩
[45]Idem, 'Some Proposals for the Realization of a Certain Assembly Program', *The Computer Journal* 3:4 (1961), ⟨URL: http://comjnl.oxfordjournals.org/cgi/content/abstract/3/4/220⟩
[46]A. A. Grau, 'Recursive processes and ALGOL translation', *Commun. ACM* 4:1 (1961)

[47]Edgar T. Irons, ‘A syntax directed compiler for ALGOL 60’, *Commun. ACM* 4:1 (1961)
[48]Idem, ‘The Structure and Use of the Syntax Directed Compiler’, in: Richard Goodman, editor, *Annual review in automatic programming 3* (1963)
[49]Robert S. Ledley and James B. Wilson, ‘Automatic-programming-language translation through syntactical analysis’, *Commun. ACM* 5:3 (1962)
[50]P. Lucas, ‘The Structure of Formula-Translators’, *ALGOL Bull.* Sup 16 (1961)
[51]ibid., p. 1
[52]ibid., p. 14
[53]ibid.
[54]ibid., p. 15
[55]ibid.
[56]Ginsburg and Rice, ‘Two Families of Languages Related to ALGOL’, p. 350
[57]Michael S. Mahoney, ‘Computer Science. The Search for a Mathematical Theory’, in: John Krige and Dominique Pestre, editors, *Science in the 20th Century* (Amsterdam: Harwood Academic Publishers, 1997), ⟨URL: `http://www.princeton.edu/~mike/articles/20thcSci/20thcent.html`⟩

4 Succession

In Search of a Worthy Successor to ALGOL 60

Using ALGOL 60 • As the main communication language • Maintenance of ALGOL 60 needed for acceptance in industry • On ALGOL X and ALGOL Y • Creating a successor to ALGOL 60 • How ALGOL 68 was the end of the ALGOL effort

4.0 Use and maintenance of ALGOL 60

In the February 1960 issue of the *Communications of the ACM*, a new department, 'Algorithms', was announced and it was dedicated to algorithms written in the ALGOL language.[0] Until the publication of the ALGOL 60 report in May 1960, the ALGOL language used in this department was IAL, after that, it was ALGOL 60. Later, ALGOL 60 was also used in several books and other journals like the *Computer Journal*, the *Computer Bulletin*, *Numerische Mathematik*, *B.I.T.*, and *ALGORYTHMY*.[1]

For years, it was the main communication language for algorithms. Besides, it was also a popular language to teach programming at universities. As a result, whole generations of computer scientists grew up with ALGOL 60 and were influenced by ALGOL 60.

Although ALGOL 60 was the most used programming language for man-to-man communication, it was certainly not the most used algebraic programming language to instruct computers with. Unfortunately, there are no sources on the use of ALGOL 60 or, for that matter, on the use of any other programming language. Two observations can be made, however: ALGOL 60 was used more in Europe than in the USA and ALGOL 60 was more popular in academic circles than in industry.

In the early 1960s, the computing community in the USA was much more developed than in Europe. Consequently, there was already a large amount of programs written in certain languages. Switching to a new language without a trusted implementation and not backed by a computer manufacturer was almost impossible. Migration to a new language would mean retraining of experienced programmers and rewriting of working software.

In Europe, the computing community was strongly influenced by the American industry, especially by IBM. On the other hand, the development in computer science took place at universities and research centres and ALGOL was popular in these circles. Furthermore, the European computing industry was emerging and the legacy of already existing software was much smaller than in the USA. In Germany, the government stimulated the use of ALGOL by requiring an ALGOL implementation on every computer ordered by a university. Although the USA government did support COBOL for data-processing, it did not support a language for algebraic applications.[2] Consequently, a new language like ALGOL 60 was able to gain more support in Europe than in the USA.

The popularity of ALGOL in universities and research centres can be explained by the nature of the ALGOL effort and its results. The ALGOL effort introduced the BNF and some new programming language concepts. Furthermore, it stimulated research on formal languages, programming languages, and translation techniques. In other words, the ALGOL effort was for a part an academic effort.

As said before, switching to ALGOL 60 was in industry not economically sound. There were more reasons why ALGOL was not accepted by industry, however. According to a survey on programming languages and processors from late 1962 there were many different ALGOL translators from different groups and companies.[3] FORTRAN, on the other hand, had also many translators, almost all of them, however, were made by IBM.[4] Hence, FORTRAN was conceived as a standard and ALGOL was not.

FORTRAN was pushed by IBM and IBM became the biggest computer manufacturer in the 1960s. At the RAND Symposium in 1961, the future of ALGOL was discussed. Bemer started the discussion with: ‘No reasonable mechanism for maintenance seems to exist. No one seems to be able to answer the basic question, ”What is ALGOL?” I foresee a difficult time for ALGOL unless a revised maintenance procedure is devised. The language is too general and has too many ambiguities.’[5] Galler added that ‘it just isn’t readable and people don’t read it.’[6]

The problem of the difficult notation in which ALGOL 60 was described would gradually disappear when the BNF became common knowledge among computer scientists. It was, on the other hand, an obstacle for many programmers to consider ALGOL 60 at all.

The problem of maintenance, however, was a more serious one. Soon after the publication of the ALGOL 60 report, it became clear that the report contained errors and ambiguities. Already in June 1960, Peter Naur wrote a letter to the authors of the ALGOL 60 report to propose four minor improvements.[7] Unfortunately, only six authors of the report reacted and they did not agree with each other on the four changes.[8]

Meanwhile, the American ACM ALGOL subcommittee was transformed into the ALGOL Maintenance Group.[9] They proposed that in Europe a similar maintenance group would be formed or that some Europeans would become members of the American Maintenance Group.[10] Bauer and Samelson were advocates of the latter option because they feared that two different groups would lead to two different languages.[11] Others, especially the Russians, wanted a separate European maintenance group.[12] The Europeans held their discussions on the maintenance of ALGOL in the *ALGOL Bulletin*. The work of the American ALGOL Maintenance Group was also reported in that bulletin.

The difference between the European and American approach to the maintenance of ALGOL is striking. In the USA, a formal maintenance group was founded almost directly after the publication of the ALGOL 60 report. The members were coming from the universities, industry and governmental research centres.[13] Although not explicitly stated, the American ALGOL Maintenance Group tried also to be a substitute for a governing body behind ALGOL. One of the main problems for the acceptation of ALGOL in industry was the lack of one responsible and stable organisation behind ALGOL. Where FORTRAN had IBM, ALGOL had just a bunch of scientists from all over the world.

4.1 The move of responsibility for ALGOL to Working Group 2.1 of IFIP

During 1960 and 1961, many problems with ALGOL 60 were collected and discussed. In the *ALGOL Bulletin* number 14, Peter Naur included a questionaire on the status of maintenance.[14] Although rather lengthy, there were three main points: ambiguities and obscurities, recommended subsets, and proposed extensions. In the June 1962 bulletin the results were published.[15]

During March 26 – 31 1962, a symposium on Symbolic Languages in Data Processing was held in Rome. A week earlier, the Working Group 2.1 on ALGOL was established as part of Technical Committee 2 (programming languages) of the International Federation of Information Processing Societies (IFIP). After the Rome symposium, on 2 and 3 April, eight of the authors of the ALGOL 60 report, some advisors and the chairman of the new founded Working Group 2.1, W.L. van der Poel, held a meeting to clean up the ALGOL 60 report. As the basis of this meeting the answers to the questionaire of *ALGOL Bulletin* 14 were used. The result of the meeting was the *Revised Report on the Algorithmic Language ALGOL 60*[16] and the transfer of responsibility for ALGOL to this Working Group 2.1 of IFIP. ALGOL got 'a new home' as Daniel D. McCracken (1962) put it.[17] With this move to IFIP, it seemed that the problem of the lack of a governing body behind ALGOL was solved. However, IFIP was not a strong organisation like IBM, but an international organisation of national computing centres without any budget of its own. ALGOL got its home, but it was yet far from home.

The meeting in Rome was not completely successful. First of all, not all problems with ALGOL 60 were resolved, five problems were left to be solved by Working Group 2.1. These problems were: side effects of functions, the call-by-name concept, static or dynamic **own**-concept, static or dynamic **for**-statement, and the conflict between specification and declaration.[18] Second, other ambiguities and inconsistencies remained in the language. In 1965, Donald E. Knuth published *A list of the remaining trouble spots in ALGOL 60*[19] in the *ALGOL Bulletin*. This list contained ten ambiguities and twelve corrections. Later, in 1969, some extra corrections were added to the list.[20] Finally, not everyone agreed with the move to IFIP. Especially Peter Naur felt that he was wronged by this move.

Although Peter Naur was not present at Rome, he had sent the participants the answers to question 40 of the questionaire in *ALGOL Bulletin* 14. This question was to 'indicate the order of preference of your group of the following bodies as far as the official adoption of clarifications, subsets, and extensions of ALGOL 60 is concerned.'[21] Most people (17) wanted to stay with the current situation of the USA Maintenance Group and the ALGOL Bulletin as their first choice. Furthermore, only ten people wanted IFIP to take over and eight people wanted an ad hoc committee. As second choice, IFIP was even less popular (3 votes).[22] Based on these results, Naur could not support the move of ALGOL maintenance to IFIP.

Despite Naur's objection, the IFIP Technical Committee established nonetheless a working group to take over responsibility for the ALGOL effort.[23] The chairman of Working Group 2.1, van der Poel, offered Naur the job of secretary of Working Group 2.1. Naur would only accept it if the more influential members of the effort like Bauer and Samelson would support him. They did not and accused Naur instead of being biased in his presentation of questionaire 14.[24] Peter Naur interpreted this as that 'it is clear that the responsible bodies of IFIP, in establishing the Working Group, deliberately have chosen to ignore the existence of the ALGOL Bulletin and the information and opinions expressed in it.'[25] Peter Naur decided to stop with his work on the *ALGOL Bulletin* which would not be published again until 1964.

Unfortunately, Peter Naur was the only one reporting on these events, it is uncertain if his view corresponded with the view of the other authors of the ALGOL report or with members of the ALGOL effort. For sure, the six authors at the Rome meeting did not vote against a move to IFIP. In addition, six of the twelve authors of the ALGOL 60 report became member of Working Group 2.1, including Peter Naur.[26] It seemed that half of the authors were not interested in further involvement with ALGOL.

This early phase of the work of Working Group 2.1 was not very interesting. Not only opponents of IFIP and the ALGOL effort governed by the IFIP like Peter Naur[27] and Wirth[28], but also van der Poel[29], the chairman of Working Group 2.1 agreed on this. The Working Group produced only a subset of ALGOL 60 and a report on input-output procedures. These documents were another consequence of the maintenance started after the publication of the ALGOL 60 report. Recall that one of the goals of the ALGOL effort was to create a universal algorithmic language. Although the ALGOL language was designed to be machine independent, implementations were, of course, machine dependent. The problem was that almost no translator was complete. Furthermore, given the ambiguities in the report, not all implemented features could be guaranteed to work in the same way on every implementation. And, again, the lack of input-output procedures, which were needed in practical situations, enforced implementors to add extensions to the language.

To improve this situation, three approaches were taken: defining subsets, defining input-output procedures and standardisation of ALGOL 60. Unfortunately, there were different subsets defined: SMALGOL 61[30], the ALCOR group subset[31], the ECMA subset[32], and the IFIP subset[33]. Also different input-output procedures were defined: one by Knuth, also known as Knuthput,[34] and one by IFIP[35]. Finally, there were also different standardisation efforts going on: by ISO, by IFIP, by ECMA, and by the ASA. These standardisation efforts were another aspect of creating a stable language and to make the language more acceptable in industry.

4.2 Creating a successor to ALGOL 60: On ALGOL X and Y

During the first half of the 1960s, maintenance of ALGOL meant solving ambiguities, removing errors, creating subsets and standardisation. Besides these aspects, there was always the wish for certain extensions to and improvements of the language. In 1964, Working Group 2.1 started with a new project: developing a new ALGOL. For the short time, there was ALGOL X and for the long time ALGOL Y. In practice, this difference was not so clear. Simple extensions of ALGOL 60 were clearly for ALGOL X, but for the more ambitious and often disputable features, the predicate 'being for ALGOL Y' meant often that it was to be reconsidered sometime in the future again. Most discussions in Working Group 2.1 were on ALGOL X, but ALGOL Y was never forgotten.[36]

At the symposium on Symbolic Languages in Data Processing, March 1962 in Rome, a discussion was held on the necessity of extensions to ALGOL 60.[37] Most of the panel members wanted some kind of extension to ALGOL 60. These wishes ranged from input-output (I/O) procedures, symbol manipulation, cleaning up of the **for** statement, double precision numbers, more types, to the ability to define new types. Most of these extensions were requested because of experience gained in the two years of using ALGOL 60, that is, they were needed in practical situations. However, with these extensions, the language would become more a general purpose programming language than a language intended for algorithmic work only.

Even before this symposium, the need for extra types was expressed by R.W. Hockney in the June 1961 edition of the *ALGOL Bulletin*. He proposed the extension of ALGOL with the complex type and a more general array concept.[38] With these additions, ALGOL would be better suited to describe algorithms involving complex numbers, matrices and bit patterns.

Another, more interesting proposal was made by Niklaus Wirth in 1963. He proposed to generalise ALGOL by removing type declarations and by replacing procedure declarations by so called 'quoted expressions'.[39] New variables were declared with the **new** operator and the type of the variables would be deduced at run time. In cases the deduced type would result in meaningless operations, the value undefined was to be assigned.

Quotations were expressions or a list of statements enclosed by the '' symbol, hence the name quotation. Quotations could be assigned to variables even with parameters between parenthesises. Where a quotation variable was used the contents of the quotation were replaced. Normal calls of these procedure-like variables resulted in call-by-value calls of the parameters. However, when one of the parameters was a quotation, it became effectively a call-by-name call.

In May 1964, F.G. Duncan revived the *ALGOL Bulletin*. In this "first" issue the open ends from the last issue were closed and a new project started. In a meeting of the Working Group 2.1 in March 1964 in Tutzing, 'there was a considerable body of opinion in favour of developing a so-called "ALGOL X" by building extensions on to ALGOL 60. This extended language would provide both a long overdue short-term solution to existing difficulties and a useful tool in the development of the radically reconstructed future ALGOL (the so-called "ALGOL Y").'[40]

Till May 1965, when draft proposals for ALGOL X were requested, and

October 1965, when these drafts were presented, various proposals were made and discussed in the *ALGOL Bulletin*. Some proposals were simple or similar to other proposals, some were more interesting.

One of the interesting proposals was the **case** expression. C.A.R. Hoare made this proposal as a replacement for the **switch** in October 1964.[41] There was an earlier proposal to remove the **switch** concept by Duncan and van Wijngaarden. They introduced the type label to enable arrays of labels which the same function as the **switch**.[42] Hoare's proposal, however, was more flexible in the sense that no array of labels had to be initialised first, nor the problems attached to labels had to be taken in account.

An example of Hoare's case expression is **case** n -2 **of** (1−y/z **else** $1 + y/z$ **else** 1). Here the case-clause n−2 decides which expression of the list following the case-clause is executed. If this case expression is 1, then the first expression is executed. If it has value 2, the second expression is executed, etc. If the value of the case-clause is 0, negative, or exceeds the number of expressions, the whole case expression was undefined. In the example, only for n = 3, 4, or 5, the case expression has a non-undefined value, namely one of the expressions in the list following the case-clause.

Another proposal in the October 1964 issue of the *ALGOL Bulletin* was made by Peter Naur.[43] He focused on the problem of ALGOL 60 being too rigidly machine independent. Because of that, features of particular machines could not be used in the language. Naur wanted to introduce a special element, the Environment Enquiry, to give the programmer information about the characteristics of the particular implementation and machine the program is running on.

Programs can become machine independent by using machine dependent characteristics which influences the working of the program. For example, if there is not enough memory space, the program can decide to stop running. Another examples are the maximum and minimum integer values. In addition, Naur (1964) proposed the possibility to create new operators, or redefine operators, the string type, the problem of labels and switches, non-rectangular arrays, and the difference between operators and standard functions.

In the *ALGOL Bulletin* of November 1965, Gerhard Seegmüller published some of his proposals for ALGOL X. He, too, wanted more types: complex, character or string, bit and label. The **for** statement was cleaned up, that is, omit the step if it is 1. He introduced a new repetitive statement, the **all** statement. A variable in the all-clause would get assigned to all the consecutive values specified in the all-list, that is, a list with numbers, or expressions generating numbers.

He introduced (1965) also the reference variable as a generalisation of the call-by-name concept. To that end he invented the **ref** operator. The reference type can be used in combination with other types into a reference array, a reference integer, or a reference integer procedure. Procedures were adapted in such a way that the call-by-name parameter calling was replaced with the call-by-reference parameter calling. A special case of the last call-type was the call-by-procedure to enable the Jensen device.

In January 1966, Niklaus Wirth and Helmut Weber published their Euler language in the *Communication of the ACM*[44][45]. This language was already discussed at the Princeton meeting of Working Group 2.1 in May 1965.[46] The article on Euler is about the way how to define a programming language, about an algorithm for syntactical analysis of phrase structure languages, and about the programming language Euler itself. Although the method of definition described by Wirth and Weber is interesting, it played a minor role in the ALGOL effort: the defining method invented by van Wijngaarden was used to describe ALGOL X, and eventually ALGOL 68. The various programming languages defined later by Wirth all used the BNF as the defining method.

The language itself[47] builds upon Wirth's earlier generalisation of ALGOL[48] (1963). The rigid type concept of ALGOL 60 was replaced by a more general type concept. Variables did not have a special type, and the types of ALGOL 60 were supplemented with types reference, label, symbol, list, procedure and undefined. Types were assigned to variables and to procedures in a dynamic way.

The most interesting type introduced in Euler was the list type. It had to replace arrays, but it was more flexible than arrays: all kind of values can be assigned to list elements, even lists itself, so tree-like structures were possible. To manipulate lists, special operators were introduced and elements in the list could be accessed by an index number, starting at one.

Procedures were similar to the 'quotations' described earlier. Procedure texts could be assigned to variables and besides the call-by-value and call-by-name, the call-by-reference was also available. Every expression resulted in a value. In case of an assignment, it was the value of the expression on the right hand side. In case of an output-expression it was the value of the expression being outputted. There were only two kinds of declarations: a new variable was declared with the **new** keyword and labels with the **label** keyword.

In the November issue of the ALGOL Bulletin in 1965, C.A.R. Hoare made a proposal for record handling.[49] The possibility to create and use records improved a programming language enormously, it could now be applied to many more problem(area)s. Where records were initially invented for data processing, i.e. in COBOL, it was now proposed as a general programming language construct.

Every record belongs to one and only one record class defined by the programmer. Such a record class denotes a certain object existing in the "real world". Every object has some properties, denoted by fields. A field is like a variable, and has a name and a type and can contain a value. Fields were declared in the declaration of the record class.

Besides properties, records can also contain relationships. To denote relationships, the type reference is introduced. In a record class definition, a field may be a reference to a record belonging to a certain record class and denotes the relationship between two records in terms of record classes. Furthermore, records can be created dynamically, that is, at run time by statements in the program.

Record fields are accessible by "functions" with the same name, which, applied to a record belonging to the record class, return the value of the field. So, given a record class containing the fields A and B and a record of that class named R, the value of field A of R is A(R) and it can be set by A(R) := value. Fields which are references, return a record, and repetitive application of fields is possible. thus A(B(R)) denotes the A value of the B value, which is a reference to record R of the same class.

In addition, two operations were defined: the destruction operator and the record class identifier as the constructor operator. Besides that, the **null** value was added to denote a reference to no record at all.

Hoare (1965) also gave some possible extensions to his record concept. The idea of procedure fields is interesting in the light of the concept of an object in object oriented programming.[51] Another extension was the set concept. Instead of encoding a set with integers and write the program using these integers, Hoare proposed to give a set a name and list all the items. Thus **set** suit(clubs, diamonds, hearts, spades); would define the set suit with four items. Extra operators, like **is a**, should be added too.

In October 1965, Peter Naur proposed to change the way procedures were specified. Instead of **procedure** P(a, b, c); **value** b; **integer** a, b; **real** c; he wanted to write **procedure** P(**integer** a, **integer value** b, **real** c);.[52] The advantages were a better readability and less repetition of keywords.

The main line in these proposals is clear: ALGOL X would introduce some new types, type flexibility, records, a better iterative statement, no labels nor denotational expressions, a case expression, I/O, string handling, and for the rest it would be an improved version of ALGOL 60. Instead of a special-purpose programming language, the new language would be a general programming language suitable for a wide range of computational problems. The need for an algorithmic language in the late 1950s had evolved in the need of a general purpose programming language in the 1960s. Numerical

```
record class person;
   begin
       integer date of birth;
       Boolean male;
       reference father, mother, youngest offspring, elder sibling (person)
   end;

reference Jack, Jill (person)

begin
    reference John (person);

    John := person; comment creates the record John
    date of birth(John) := today;
    male(John) := true;
    father(John) := Jack;
    mother(John) := Jill;
    youngest offspring(John) := null;
    elder sibling(John) := youngest offspring(Jack);
   youngest offspring(Jack) := John
end
```

Figure 4.0: Example from C.A.R. Hoare's article on record handling.[50]

applications were clearly not the only relevant applications any more.

4.3 The end of the ALGOL effort: the creation of ALGOL 68

4.3.0 Orthogonality versus pragmatism

In the previous section, the proposed extensions and changes to ALGOL 60 were discussed. These proposals where the basis for the development of ALGOL X. At the meeting of Working Group 2.1 at Princeton, May 1965, only one more or less complete proposal was available.[53] The members of the group were then invited to create their own proposal for ALGOL X.[54] In October 1965, the next meeting of the Working Group was held at Saint Pierre de Chartreuse and three proposals were presented.[55] Wirth's (1966) Euler was combined with Hoare's (1966) proposal for records and was the most feature complete proposal. Seegmüller's proposal was, basically, an extension to Wirth's (1963) earlier proposal and was not seen as a serious candidate.[56]

The last proposal was *Orthogonal design and description of a formal language*[57] by van Wijngaarden. It was not really a proposal for a language, but a proposal for a better method to define programming languages. In this report, three new ideas were expressed.[58] First of all, a two-level grammar was used instead of the BNF. This grammar would later be known as vW-grammar or W-grammar. Second, van Wijngaarden proposed orthogonality, or in his own words: 'As to the design of a language I should like to see the definition of a language as a Cartesian product of its concepts.'[59] Finally, the language defined was an expression-oriented language: there was no distinction between expressions and statements.

The result of this meeting was that a subcommittee was founded consisting of the four authors of proposals for ALGOL X. Together they would decide upon one final proposal for ALGOL X. However, it was also decided that the resulting language would be described in the notation invented by van Wijngaarden.[60] The subcommittee decided further that van Wijngaarden would write the proposal and would send it to the others for discussion.[61] In addition, another subcommittee was created to discuss I/O.[62]

In April 1966, the subcommittee working on the new proposal held a meeting at Kootwijk where two proposals were presented. The proposal of van Wijngaarden was written using the accepted notation, but was not complete. The other proposal was written by Wirth and Hoare and would later be published in the *Communications of the ACM* as *A Contribution to the Development of ALGOL*[63]. Their proposal, although not written in the right notation, was felt to describe the right language.[64] Unfortunately, Hoare and Wirth on the one hand, and Seegmüller and van Wijngaarden on the other hand could not decide upon one language acceptable by all. The orthogonality pushed by van Wijngaarden conflicted with the more pragmatic approach taken by Wirth and Hoare.[65] They wanted to create a language which could become ALGOL 66 and they felt that the language proposed by van Wijngaarden was too ambitious.[66]

4.3.1 A contribution to the development of ALGOL

Wirth and Hoare decided to publish their proposal in the *Communications of the ACM* as *A Contribution to the Development of ALGOL*. The goals stated at the start of the article were: to give a presentation of where the ALGOL effort was heading for a broader public; be a document which could be used for experimental implementations; and describe problems for further extention.[67] The language was designed with four criteria in mind: it should be usable for programming computers; it should be usable usable as a communication language; it should be usable for teaching and research; and it should be as practical usable as possible.[68]

The language described contained, with respect to ALGOL 60, new data types: complex, a long variant for both real and complex, sequences in the form of a sequence of bits (called bits) and a sequence of characters (called string). For these new types, appropriate operators and functions were defined too, like the concatenate operator for strings, the logical operators for bits, and type conversions. In addition, the language was strong typed: types of expressions were known at compile time.

Switches were replaced by the case expression. Labels became just labels, that is, place markers for use with the **go to** statement. All other forms of designational expressions were removed. The iterative statements were simplified and contained only the while and the for-step-until variants. Parameters to procedures are either value-parameters, or result-parameters. Arrays were slightly simplified. the record type was added including references to records as proposed by Hoare.

After this short introduction into the main features of the language, the authors gave some extensions: extra string operations, more flexible types, initial values and constants, array constructors (initial values for arrays), and more flexible handling of records, for example, the union denoting that a record reference variable may refer to records of all record classes in the union, or a **is** operator deciding if a record is of a certain record class.

Then the formal definition of the language defined using the BNF followed. In part three, the 'proposed set of standard procedures' or standard library was discussed. This library contained I/O, environment enquiries like in Naur's (1964) proposal (making the programmer able to ask questions on the implementation and decide on certain features in the program, like the biggest integer), mathematical functions, like sin, pi, ln, exp, etc.

This document was the basis for the further development of languages by Wirth. In September 1966, Wirth reported in the *ALGOL Bulletin* that he was implementing the language on an IBM 360 at Stanford University.[69] While implementing he came across some problems and altered the language accordingly. Some syntactic sugar for arrays was introduced, like `x[i,j]` instead of `x[i][j]`. Strings became static, that is, they could not grow, and were declared being of a certain length. In the **for** statement, the step element may be omitted when one. Concurrency was added with the **also** keyword: S **also** T means that S and T are executed concurrently. Synchronisation of parallel processes was available via P and V operations invented by Dijkstra, Wirth (1966) called them **on** and **off** respectively.

4.3.2 The end and the ALGOL 68 report

Although the subcommittee could not agree upon one proposal, van Wijngaarden did create a new draft for the next meeting of Working Group 2.1 in Warshaw. Actually, the meeting was moved to October 1966 because Van Wijngaarden was unable to produce the draft in time.[70] At the meeting, the main question was if this draft could be accepted because it was written by only one member of the subcommittee. Wirth did not attend the meeting and resigned later form Working Group 2.1 alltogether. Hoare wanted to consider accepting the draft only if it was completed first; the results of the I/O subcommittee was not yet added to the draft of van Wijngaarden.[71] Besides these more formal discussions, some technical matters were also treated. Some people wanted to restrict references to records only, the diagonal approach, which conflicted with the orthogonal approach taken. In this orthogonal approach references should be applicable to all objects, not only records. McCarthy proposed operator overloading and Samelson anonymous subroutines.[72]

At the end of the meeting there was still no complete draft. It was decided that van Wijngaarden would edit alone the draft which would be complete at the next meeting in May. Van Wijngaarden thought that 'the delay in producing the final version may not be very long'[73], however, it would be February 1968 before the draft, known as MR 93[74], was ready. It 'was the cause of much shock, horror and dissent, even (perhaps especially) amongst the membership of WG2.1. It was said that the new notation for the grammar and the excessive size of the document made it unreadable.'[75]

Opposition agains the notation and the draft became louder, dropping the draft or adding a minority report became realistic options.[76] At the meeting in Mϋnich, December 1968, the ultimate decision had to be made: drop it or accept it as ALGOL 68. *Report on the Algorithmic Language ALGOL 68*[77] was accepted and a minority report was published too, signed by almost half of all the members of Working Group 2.1.[78] stating that:

> We regard the current Report on Algorithmic Language 68 as the fruit of an effort to apply a methodology for language definition to a newly designed programming language. We regard the effort as an experiment and professional honesty compels us to state that in our considered opinion we judge the experiment to be a failure in both respects.
>
> The failure of the description methodology is most readily demonstrated by the sheer size of the Report in which, as stated on many occasions by its authors, "every word and every symbol

> matters" and by the extreme difficulty of achieving correctness.
>
> (...)
>
> We fail to see how the language proposed here is an 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.[79]

The spirit with which the ALGOL effort started in the 1950s was broken. The ALGOL effort ended with a failure for some. Others, however valued the new language, especially after the publication of the revision in September 1973. The language was not used much in industry, although it was used in the academic world, but less than ALGOL 60 was. It is often regarded, however, that it influenced many languages which were developed later. The question is if the language itself was influential, or that the development of the language was influential.

In my definition, the ALGOL effort ends here with the publication of the ALGOL 68 report and the Minority Report. This does not mean that there were no further developments in Working Group 2.1 and the ALGOL Bulletin: as was the case with ALGOL 60, ALGOL 68 also had its maintenance period and the last ALGOL Bulletin was issued in 1988.

4.4 Conclusion

The question that should be asked is what was the importance of this period of maintenance and succession? Was it important, and if so, why? First of all, the maintenance of ALGOL 60 was necessary for ALGOL itself. It was necessary to become accepted in industry, although it did not work out very well.

Although ALGOL 60 was the most used programming language for man-to-man communication, it failed to truly become the universal algebraic programming language for man-to-machine communication due to the computational context of that time: the predominance of IBM pushing FORTRAN, the legacy of applications already created and used in the industry. ALGOL was a new untested language and, especially on the American market and outside universities, it could not win the fight with FORTRAN.

Second, the move to IFIP seemed a wise move for the ALGOL effort: the lack of a governing body was now solved. It was, however, too late to obtain a fundamental part of the market. As part of IFIP, Working Group 2.1 created some reports on subsets, I/O, and standardisation. These results were not very interesting and definitely not important for the field of programming languages or computer science.

The move to create a successor to ALGOL 60 in 1964, was much more important. The ALGOL effort became a platform where the best computer scientists of that day discussed programming languages. Many proposals were made and discussed, most of them boiling down to extra types, string handling, cleaning up the **for** statement, and other minor improvements. Major improvements were made by Hoare: the case expression which allowed the removal of designational expressions and part of the label concept. More important was his paper on record handling (1966).

With records came references and with references the call-by-name concept could be replaced with call-by-reference. Although records were not new, Hoare made them a general programming language concept. There were other proposals, on overloading and the ability to (re)define operators. Furthermore, the standard library became larger, containing not only mathematical functions, but also I/O, string handling, and casts.

There was clearly a move from a special-purpose language towards a general programming language. Unfortunately, Working Group 2.1 was not able to define the new language with agreement of all its members. The new language had to be defined in the notatation invented by van Wijngaarden, which turned out to result in a unreadable report. After much delay, the draft was presented and accepted by the Working Group, however, not without a Minority Report of almost half of the Working Group. In this report it was stated that they considered the new language a failure.

4.5 Notes

[0]J. H. Wegstein, ‘Algorithms: Anouncement’, *Commun. ACM* 3:2 (1960)

[1]R. W. Bemer, *The Programmer’s ALGOL: A Complete Reference* (London: McGraw-Hill, 1967), chap. Foreword, p. x

[2]ibid., p. viii

[3]Bro, ‘Survey of programming languages and processors’, *Commun. ACM* 6:3 (1963), p. 5

[4]ibid., p. 4

[5]J. H. Wegstein, ‘ALGOL: a critical profile. The RAND Symposium, part two’, *Datamation* 10 (1961), p. 41

[6]ibid.

[7]Peter Naur, ‘ALGOL 60 Maintenance’, *ALGOL Bull.* 10 (1960), pp. 1–2

[8]ibid., p. 1

[9]ibid., p. 7

[10]ibid., p. 4

[11]Idem, ‘ALGOL 60 Maintenance’, *ALGOL Bull.* 11 (1960), p. 1

[12]ibid., pp. 2–3

[13]Idem, ‘ALGOL 60 Maintenance’, p. 5

[14]Idem, ‘The Questionnaire’, *ALGOL Bull.* 14 (1962), pp. 1–14

[15]Idem, ‘The discontinuation of the ALGOL Bulletin’, *ALGOL Bull.* 15 (1962)

[16]J. W. Backus et al., ‘Revised report on the algorithm language ALGOL 60’, *Commun. ACM* 6:1 (1963)

[17]Daniel D. McCracken, ‘A New Home for ALGOL’, *Datamation* 5 (1962), p. 44

[18]Backus et al., ‘Revised report on ALGOL 60’, pp. 2–3

[19]D. E. Knuth, ‘A list of the remaining trouble spots in ALGOL60’, *ALGOL Bull.* 19 (1965)

[20]Derick Wood, ‘A few more trouble spots in ALGOL 60’, *Commun. ACM* 12:5 (1969)

[21]Naur, ‘The Questionnaire’, p. 13

[22]Idem, ‘The discontinuation of the ALGOL Bulletin’, p. 2

[23]ibid.

[24]ibid., p. 3

[25]ibid.

[26]C. H. Lindsey, *A history of ALGOL 68*, in: *HOPL-II: The second ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1993), p. 7

[27]Peter Naur, ‘Successes and failures of the ALGOL effort’, *ALGOL Bull.* 28 (1968), p. 60

[28]N. Wirth, ‘Closing word at Zurich Colloquium’, *ALGOL Bull.* 29 (1968), p. 19

[29]W. L. van der Poel, *Some Notes on the History of ALGOL*, in: *A Quarter Century of IFIP* (Amsterdam: Elsevier, 1986), p. 374

[30]G. A. Bachelor et al., ‘SMALGOL-61’, *Commun. ACM* 4:11 (1961)

[31]Bumgarner and Feliciano, ‘ALCOR ALGOL Symbols’

[32]Fraser G. Duncan, ‘ECMA Subset of ALGOL 60’, *Commun. ACM* 6:10 (1963)

[33]S. Gorn, ‘Report on SUBSET ALGOL 60 (IFIP)’, *Commun. ACM* 7:10 (1964)

[34]D. E. Knuth, ‘A proposal for input-output conventions in ALGOL 60’, *Commun. ACM* 7:5 (1964)

[35]Gor, ‘Report on Input-Output Procedures for ALGOL 60’, *Commun. ACM* 7:10 (1964)

[36]Van der Poel, ‘Some Notes on the History of ALGOL’, p. 375

[37]P. Dreyfus, *General Panel Discussion. Are Extensions to ALGOL 60 necessary and if so What Ones?*, in: *Symbolic Languages in Data Processing* (London: Gordon and Breach Science Publishers, 1962)

[38]R. W. Hockney, ‘A Proposed Extension to Algol 60’, *ALGOL Bull.* Sup 12 (1961)

[39]Niklaus Wirth, ‘A generalization of ALGOL’, *Commun. ACM* 6:9 (1963)

[40]F. G. Duncan and A. van Wijngaarden, ‘Cleaning up ALGOL60’, *ALGOL Bull.* 16 (1964), p. 24

[41]C. A. R. Hoare, ‘Case expressions’, *ALGOL Bull.* 18 (1964)

[42]Duncan and Van Wijngaarden, ‘Cleaning up ALGOL60’

[43]P. Naur, ‘Proposals for a new language’, *ALGOL Bull.* 18 (1964)

[44]Niklaus Wirth and Helmut Weber, ‘EULER: a generalization of ALGOL and it formal definition: Part 1’, *Commun. ACM* 9:1 (1966)

[45]Idem, ‘EULER: a generalization of ALGOL, and its formal definition: Part II’, *Commun. ACM* 9:2 (1966)

[46]Van der Poel, ‘Some Notes on the History of ALGOL’, p. 374

[47]Wirth and Weber, ‘EULER: a generalization of ALGOL, and its formal definition: Part II’

[48]Wirth, ‘A generalization of ALGOL’

[49]C. A. R. Hoare, ‘Record Handling’, *ALGOL Bull.* 21 (1965)

[50]ibid., pp. 43–44

[51]Actually, SIMULA 67 was developed in the late 1960s as an extension of ALGOL 60 and successor to SIMULA and influenced by Hoare’s article and it is often called the first object oriented language. There the concept of records and record classes was used for processes and activities, respectively.

[52]Peter Naur, ‘The form of specifications’, *ALGOL Bull.* 22 (1966)

[53]Van der Poel, ‘Some Notes on the History of ALGOL’, p. 374

[54]ibid., p. 375

[55]Lindsey, ‘A history of ALGOL 68’, p. 8

[56]ibid.

[57]A. Van Wijngaarden, ‘Orthogonal design and description of a formal language’ (October 1965), ⟨URL: http://www.fh-jena.de/~kleine/history/languages/VanWijngaarden-MR76.pdf⟩

[58]Lindsey, ‘A history of ALGOL 68’, p. 8

[59]Wijngaarden, ‘Orthogonal Design’, p. 3

[60]Van der Poel, ‘Some Notes on the History of ALGOL’, p. 377

[61]Lindsey, ‘A history of ALGOL 68’, p. 9

[62]ibid.

[63]Niklaus Wirth and C. A. R. Hoare, ‘A contribution to the development of ALGOL’, *Commun. ACM* 9:6 (1966)

[64]Lindsey, ‘A history of ALGOL 68’, p. 9

[65]Van der Poel, ‘Some Notes on the History of ALGOL’, p. 378

[66]ibid., p. 379

[67]Wirth and Hoare, ‘A contribution to the development of ALGOL’, p. 4113

[68]ibid., pp. 413–414

[69]N. Wirth, ‘Additional Notes on - Contribution to the Development of ALGOL’, *ALGOL Bull.* 24 (1966), p. 13

[70]Van der Poel, ‘Some Notes on the History of ALGOL’, p. 379

[71]Lindsey, ‘A history of ALGOL 68’, p. 10

[72]ibid.

[73]Van der Poel, ‘Some Notes on the History of ALGOL’, p. 379

[74]A. Van Wijngaarden et al., *Draft Report on the Algorithmic Language ALGOL 68*, edited by Idem (Amsterdam: Mathematisch Centrum, 1968)

[75]Lindsey, ‘A history of ALGOL 68’, pp. 11–12

[76]Van der Poel, ‘Some Notes on the History of ALGOL’, pp. 385–386

[77]A. Van Wijngaarden et al., *Report on the Algorithmic Language ALGOL 68*, edited by Idem (Amsterdam: Mathematisch Centrum, 1969)

[78]Lindsey, ‘A history of ALGOL 68’, pp. 14–15

[79]Dijkstra et al., ‘Minority Report’

5 Conclusion
Summary and Conclusions

Summary • Important contributions of the ALGOL effort to computer science • BNF, programming language concepts and syntax directed translation • Connected by the theory of context-free languages • And the contribution of the scientific field of translator writing • Success and failure of the ALGOL effort

In the introduction the question was asked: *What was the importance of the ALGOL effort for the development of computer science?* After having studied four phases of the ALGOL effort, Creation, Notation, Translation, and Succession, it is time to answer the question and draw conclusions. Before answering this question, however, a summary of the preceding chapters is given.

5.0 Summary

5.0.0 The start of the ALGOL effort

In Chapter 1 the history of the start of the ALGOL effort and the reasons for its existence were told. In the late 1950s, the need for a universal algorithmic programming language was felt. In Central Europe, a GAMM subcommittee was founded to create an algebraic language and during the development of this language the ACM in the USA was contacted. It was decided that both the GAMM subcommittee and the newly formed ACM subcommittee would create a proposal for an algorithmic language. At the Zürich meeting in 1958, both subcommittees met and they drew up a proposal for an international algebraic language: IAL.

Two questions were asked: why was there a need for a universal algorithmic language and why did it have to be IAL and not some of the already existing languages? First of all, in the USA, the field of computing became an industry in the 1950s. Computers were sold to and used by many corporations, research centres and the government. In this commercial atmosphere, using problem-oriented programming languages, like algorithmic languages or data-processing languages was needed to make profitable the use of the expensive computing machines.

These programming languages, or automatic coding systems, did have a bad name: it was believed that they were fundamentally inefficient. The FORTRAN programming system, however, proved the opposite as FORTRAN programs were efficient, although not as efficient as hand-coded programs. Problem-oriented languages were seen as a necessity and many research groups and computer companies started creating these languages for both internal use and as part of the commercial computer systems that they were selling. Unfortunately, this development resulted in the creation of many similar, but different algorithmic programming languages. In this context the cry for a universal algorithmic programming language became louder.

In Europe, the situation was totally different. The field of computing was emerging and the first larger computers were being built. The main application for these machines was scientific computing. Instructing these computers was a difficult and error prone task. To solve this problem, work was started on formula translation and eventually on an algorithmic language in the GAMM subcommittee. Instead of creating yet another algorithmic language, the members of the subcommittee proposed to jointly create one international algebraic language to the ACM.

The Americans did not base their proposal on an already existing language. IT and MATH-MATIC were not sufficient or well known. FORTRAN, on the other hand, was well known, but the American members of the committee did not want to increase the dominance of IBM. As a result, the ACM subcommittee proposed a new language. The combination of the two proposals resulted in IAL: a new algorithmic language like other algorithmic languages of that time.

This start of the ALGOL effort was important because it created a common goal among computer scientists from the USA and Europe: developing one algorithmic language. Creating one universal algorithmic language became part of the agenda of computer science.[0] This may not sound important, but in an emerging field, as was the case with computer science, it meant more coherence. It meant that computer scientists could identify themselves, both to each other and to the outer world, as workers in computer science.

To enable this international development, however, communication between scientists was necessary. Although they had meetings dedicated to their common goal, communication on programming languages between different groups and people with different backgrounds appeared difficult: a sufficient notation to describe and discuss programming languages was lacking.

5.0.1 From IAL to ALGOL 60: notation and language

A sufficient notation was developed during the two years between the publication of the draft report on IAL and the publication of the ALGOL 60 report. In Chapter 2 this development of notation is discussed. Notation is important because it enables one to formally define a language in such a way that everyone can read and interpret the definition in the same way.

The notation to describe the early programming languages, like FORTRAN and IAL, was mostly natural language combined with some patterns denoting the form of the various language elements. The problem with this notation was ambiguity. Even for simple language elements, like numbers, expressions and simple control structures, this was already a problem. For complex structures like the procedure statement and declarations it was more problematic.

To give a more formal and complete description of the syntax of IAL a new notation was invented by Backus: the Backus Normal Form. Using this simple notation, complex structures in the language could be described formally. Unfortunately, the procedure concept of IAL was too complex to be described using this notation.

The procedure concept was also the most controversial topic in the various discussions on the development of ALGOL 60. Eventually, in the ALGOL 60 report, the procedure concept was strongly simplified. Input and output parameters were removed and call-by-name and call-by-value parameters introduced. Another important aspect of the new language was the notion of a block with its own scope. This block was an extension of the compound statement from IAL. What was new was recursion, added implicitly to the language because it was not explicitly forbidden.

The ALGOL 60 report was edited by Peter Naur. He wrote the draft version and used a slightly modified version of Backus's notation to describe the language. This draft was used as the basis for the ALGOL meeting. The final report would become the standard method of defining programming languages and the BNF became the standard method to describe the syntax of programming languages.

The importance of this phase of the ALGOL effort was twofold. First of all, it resulted in a new notation which would become the standard to define and describe programming languages. Although the goal of the ALGOL effort was to create a universal algorithmic language, it also created a universal metalanguage to describe all programming languages. The scope of the effort became larger than only the development of ALGOL 60: the whole field of programming languages became the scope of the ALGOL effort.

Second, the ALGOL effort became truly international and the result, ALGOL 60, was a new programming language, substantially different from the earlier algorithmic languages. It had a clear syntax and it introduced some interesting concepts to a broader public, like blocks, procedures and recursion.

In 1958, the common goal of the ALGOL effort was shared by a small number of computer scientists from the USA and Central Europe. With the publication of the ALGOL 60 report, the common spirit was shared by many computer scientists from all over the world. Computer science had produced a potent result: ALGOL 60.

5.0.2 From craftsmanship to science: translating ALGOL 60

The potential of ALGOL 60 became clear during the implementation phase in the early 1960s: the ALGOL effort was a catalyst, transforming the field of translator writing from craftsmanship into a science. This transformation started with the development and publication of a translation technique for IAL by Bauer and Samelson. They were the initiators of the ALGOL effort and started with the construction of both a language and translation technique. Although their technique was known and developed by many other computer scientists, it was the publication of their technique that made it so important.

The language was made to fit the technique, and vice versa. When Bauer and Samelson, some months before the ALGOL 60 was created, published their technique, their article became widely read. And, because of the popularity of ALGOL 60, the implementation of many translators was being started, many based upon the technique described by Bauer and Samelson. During this process, the isolated nature of efforts to create an algebraic language was finally broken: publication and referring to publications became the norm.

Although there was a translation technique for IAL, to translate ALGOL 60, the technique had to be extended. These extensions fitted into the well known translation technique. Other approaches to the translation of ALGOL were improving the technique and inventing new techniques: syntax directed translation techniques. A new technique was invented based on the structure of ALGOL. From left to right, from bottom to top, this technique was based on a metalanguage in which both syntax and semantics were to be specified. A general translator was then able, with these specifications and a program text, to translate the program into a executable version.

Although these approaches were different, the lack of a common notation, terminology, or theory was striking. Clearly, the field missed a sound theory to be able to evolve from a craft into a science. Ginsburg and Rice supplied the field with a theory: they created the connection between Chomsky's context-free languages and ALGOL-like languages and hence connected the field of programming languages with other theories of computing and automata theory. With that, computer science received its own theoretical component.

The further development of the field of translator writing, although related to ALGOL at first, was no longer ALGOL centred. It became an active field of research, evolving independently from the ALGOL effort.

5.0.3 In search of a worthy successor to ALGOL 60

The ALGOL effort continued with a period of maintenance. A formal maintenance body was necessary to become accepted, although it did not work out very well. Although ALGOL 60 was the most used programming language for man-to-man communication, it failed to truly become the universal language due to the computational context of that time: the predominance of IBM pushing FORTRAN, the legacy of applications already created and used in industry. ALGOL was a new untested language, and, especially on the American market and outside universities, it could not win the fight with FORTRAN.

Second, the ALGOL effort was moved to IFIP, solving the lack of a governing body. It was, however, too late to gain a fundamental part of the market. As part of IFIP, Working Group 2.1 created some reports on subsets, I/O, and standardisation. These results were not very interesting, and, not important for the field of programming languages or computer science.

The move to create a successor to ALGOL 60 in 1964, however, was much more important. The ALGOL effort became a platform for the best computer scientists of that period to discuss programming languages and their underlying concepts. Many proposals were made and discussed, most of them boiling down to extra types, string handling, cleaning up the **for** statement, and other minor improvements. Major improvements were made by Hoare: the case expression which allowed the removal of designational expressions and part of the label concept.

More important was his paper on record handling. With records came references and with references, the controversial call-by-name parameter concept could be replaced with the call-by-reference parameter concept. Although records were not new, Hoare made them a general programming language concept.

There were other proposals, on overloading and the ability to (re)define operators. Furthermore, van Wijngaarden proposed orthogonality, one of the most important contributions to programming. In addition, the standard library became larger, containing not only mathematical functions, but also I/O, string handling, and type casts.

There was clearly a move from a special-purpose language towards a general programming language. Unfortunately, Working Group 2.1 was not able to define the new language with agreement of all members. The new language had to be defined in a new notatation, the van Wijngaarden grammar, which turned out to result in an unreadable report. After much delay, the draft was presented and accepted by the Working Group as the ALGOL 68 report, however, not without a Minority Report signed by almost half of Working Group 2.1. In this minority report it was stated that they considered the new language a failure. And so the ALGOL effort came to a sad end.

5.1 Conclusions: The important contributions by the ALGOL effort

5.1.0 The Backus Naur Form

One of the most important contributions to computer science by the ALGOL effort was the notation to define the syntax of programming languages: the BNF. Besides the BNF the ALGOL 60 report as a whole was also important because it would set the example for the definition of programming languages. By using the BNF to define the syntax of ALGOL 60 the report itself became a clear and structured document.

The BNF was developed by Backus to be able to define the syntax of IAL in a formal way. It was an huge improvement over earlier notations to describe programming languages. These earlier notations used natural language mixed with patterns. As a result, these notations were ambiguous and unable to formally define a programming language completely.

Although Backus's notation was an improvement, Backus was not able to define the complex procedure concept with his notation. Furthermore, the notation did not gain much attention until Naur used it for the draft ALGOL 60 report. In this report, he changed Backus's notation by replacing some symbols and using complete words instead of abbreviations. With the interest generated by the ALGOL 60 report the BNF also became well known.

The use of the BNF to define the syntax of ALGOL 60 resulted in a highly structured language. Programming language concepts known from IAL were more clearly defined. This clarification of programming language concepts was most visible by the procedure concept: the complex procedure statement from IAL was transformed into an elegant, simple and powerful procedure statement in ALGOL 60.

Although the BNF was important for the definition and development of ALGOL 60, the greatest importance of the BNF lie in its general applicability: the BNF could be used to define all ALGOL-like programming languages. After the publication of the ALGOL 60 report the syntax of almost all programming languages would be defined using the BNF. The ALGOL effort had supplied computer science with a notation to define programming languages.

5.1.1 Programming language concepts

Another important contribution by the ALGOL effort was the introduction or popularisation of programming language concepts. Although IAL was just another algebraic programming language like other programming languages of that time, it introduced the compound statement: one or more statements enclosed by the **begin** and **end** keywords were treated as one single "compound" statement.

The power of this compound statement became clear in the ALGOL 60 report: the compound statement was transformed into a special case of the block concept. A block consisted of a number of declarations followed by a number of statements and the whole was enclosed by the **begin** and **end** keywords. A compound statement was a block without declarations. Through these declarations a block got its own local scope known only inside the block.

The block was the building block of ALGOL 60: every ALGOL 60 program was a block containing declarations and statements wherein blocks could occur. As said earlier, the procedure concept was simplified in ALGOL 60. It now consisted of a heading and a body. The body was a block. In the heading the name, a list of arguments, and declarations of the arguments could be specified.

In the definition of the procedures of ALGOL 60 the distinction between call-by-name and call-by-value parameters was made. The call-by-value parameter is straightforward: the value of an argument was used as the value of the parameter in the body of the procedure. The call-by-name concept was different. Instead of assigning the value of the argument to the parameter in the body, the parameter in the body was substituted with the text of the argument.

The procedure concept in ALGOL 60 was also important because it made recursion possible. Although recursion was already known, ALGOL 60 popularised the concept.

During the development of a successor to ALGOL 60 other programming language concepts were introduced or popularised as well. Most notably were the **case** statement, the environment enquiries, orthogonality and the record concept including the reference type and call-by-reference parameter concept.

The importance of the ALGOL effort was that it made these programming language concepts mainstream. Since ALGOL 60 blocks and recursion are included in almost all programming languages, since the record proposal was made by Hoare (1965), records are common in general programming languages.

In addition, the various programming language concepts introduced in the ALGOL effort and the discussions about these concepts influenced other programming languages and thinking about programming languages in general.

5.1.2 Syntax directed translation and dynamic memory management

Besides the BNF and new programming language concepts the ALGOL effort contributed also translation techniques. First of all, new programming language concepts like procedures, recursion, and variable sized arrays did not directly fit in the well known sequential translation technique. An execution of an IAL program was, with respect to memory usage, static in nature: at compile time all memory management was done.

ALGOL 60, however, was another matter as a result of its possible dynamic behaviour: the bounds of arrays could be unknown at compile time and recursion could be of any unknown depth (as long as there was enough memory, of course). These problems were already solved in 1960 by different people by introducing an extra runtime stack to allocate memory for those dynamic elements. This dynamic memory management was general applicable to all translators for all languages where dynamic memory management was needed.

Besides dynamic memory management the ALGOL effort started an another approach to translation of programming languages: syntax directed translation. Influenced by the structure of ALGOL 60 and the BNF research was started to exploit this structure to create ALGOL 60 translators. It appeared that similar patterns of BNF definitions could be translated by similar translation schemes. This resulted in general translator schemes applicable to all ALGOL-like languages.

5.1.3 The contribution of the field of translator writing

All contributions by the ALGOL effort to computer science mentioned above were in itself important. The main importance of the ALGOL effort was, however, the combined contribution. The BNF, the programming language concepts, and the syntax directed translation techniques were connected by Ginsburg and Rice (1962) by proving the equivalence of languages defined by the BNF (notably ALGOL 60) and context-free languages.

With this connection a scientific field of translation was created: the practice of ALGOL became founded on a theoretical basis of formal languages. Soon research on formal languages, including programming languages, and on the connection with other mathematical theories like automata theory was started. In addition, the syntax directed translation techniques evolved rapidly by applying the new established theory. During the 1960s and 1970s, the field of translation became the best known field of computer science.

The importance of the ALGOL effort lies in its scope. Although the ALGOL effort produced several programming languages, it was not bound to these languages only. The contributions of the ALGOL effort were applicable to programming languages in general.

5.2 Success and failure

In the previous section the importance of the ALGOL effort for computer science is made clear. The ALGOL effort was a success. Furtermore, ALGOL 60 did become the most used programming language for man-to-man communication. On the other hand, the ALGOL effort was also a failure: the languages it produced did not become the universal algorithmic programming language in industry. Although it was used in Europe more than in the USA, it could not beat FORTRAN. FORTRAN became the de facto standard programming language for numerical work. This contradictory nature of the ALGOL effort is striking. How can something both be such a success and such a failure?

Of course, the answer on the question about success depends on the definition of success. Nonetheless, ALGOL was intended to be used as a programming language to instruct computers with and it failed in that respect. Despite its quality ALGOL was not received well in industry. From the perspective of the industry ALGOL was not a trustworthy language: it lacked a governing body. Where FORTRAN had IBM, ALGOL 60 had nothing but a bunch of scientists. The dominance of IBM and FORTRAN was also a part of the explanation of the failure of ALGOL; there were many computer manufacturers willing to implement and ship the language with its computers but they were not able to break the dominance of IBM.

The ALGOL languages, however, did become the main communication language for algorithms in many journals and publications. Furthermore, it was also often used to teach programming at universities. As a result, ALGOL became well known among computer scientists and influenced their thinking on programming languages. In addition, the ALGOL effort was the platform to discuss programming languages, and many features discussed during the development of ALGOL 68 would appear later in other programming languages. With respect to communication and influence, ALGOL was a success.

That the ALGOL languages could not become major players among the programming languages in industry did not affect its influence or its importance. On the contrary, by being a minor language there was less need for stabilisation or backwards compatability. By being an "academic language" ALGOL stayed object of research and development. Consequently, the ALGOL effort was able to prepare the way for a new generation programming languages, like Simula, Pascal, C, Smalltalk, etc., without the ballast of legacy.

5.3 Notes

[0]As believed necessary for the establishment of a scientific field by Mahoney in: Mahoney, 'Computer Science. The Search for a Mathematical Theory'

Bibliography

'Survey of programming languages and processors', *Commun. ACM* 6:3 (1963), pp. 93–98.

'Report on Input-Output Procedures for ALGOL 60', *Commun. ACM* 7:10 (1964), pp. 628–630.

ACM, 'A. M. Turing Award' (2006), ⟨URL: http://awards.acm.org/turing/⟩.

Bachelor, G. A. et al., 'SMALGOL-61', *Commun. ACM* 4:11 (1961), pp. 499–502.

Backus, J. W. et al., 'Report on the algorithmic language ALGOL 60', *Commun. ACM* 3:5 (1960), pp. 299–314.

Backus, J. W. et al., 'Report on the algorithmic language ALGOL 60', *Numerische Mathematik* 2:1 (1960), pp. 106–136.

Backus, J. W. et al., 'Revised report on the algorithm language ALGOL 60', *Commun. ACM* 6:1 (1963), pp. 1–17.

Backus, John, *The history of FORTRAN I, II, and III*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), pp. 165–180.

Backus, John, 'Programming in America in the 1950s – Some Personal Impressions', in: Metropolis, N., Howlett, J. and Rota, Gian-Carlo, editors, *A History of Computing in the twentieth century* (Academic Press, 1980), pp. 125–135.

Backus, John W., *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.*, in: *IFIP Congress* (1959), pp. 125–131.

Backus, J.W. et al., 'The FORTRAN Automatic Coding System for the IBM 704 EDPM : Programmer's Reference Manual', Technical report (Applied Science Division and Programming Research Department, International Business Machines Corporation, 1956), ⟨URL: http://community.computerhistory.org/scc/projects/FORTRAN/704_FortranProgRefMan_Oct56.pdf⟩.

Bauer, F. L., ‘The Cellar Principle of State Transition and Storage Allocation’, *Annals of the History of Computing* 12:1 (1990), pp. 41–49.

Bauer, F. L. and Wössner, H., ‘The ‘Plankalkül’ of Konrad Zuse: a forerunner of today’s programming languages’, *Commun. ACM* 15:7 (1972), pp. 678–685.

Bauer, Friedrich L., ‘Between Zuse and Rutishauser – The Early Development of Digital Computing in Central Europe’, in: Metropolis, N., Howlett, J. and Rota, Gian-Carlo, editors, *A History of Computing in the twentieth century* (Academic Press, 1980), pp. 505–524.

Bauer, Friedrich L., *From the Stack Principle to ALGOL*, in: Broy, Manfred and Denert, Ernst, editors, *Software pioneers : contributions to software engineering* (Berlin: Springer, 2002), pp. 26–42.

Bauer, Friedrich L. and Samelson, K., ‘Sequentielle Formelübersetzung’, *Elektronische Rechenanlagen* 1:1 (1959), pp. 176–182.

Bemer, R. W., *The Programmer’s ALGOL: A Complete Reference* (London: McGraw-Hill, 1967), chap. Foreword, pp. vii–xiii.

Bemer, R. W., ‘A Politico-Social History of Algol’, in: Halpern, Mark I. and Shaw, Christopher J., editors, *Annual review in automatic programming*, volume 5 (London: Pergamon, 1969), pp. 151–237.

Brooker, R. A. and Morris, D., ‘An Assembly Program for a Phrase Structure Language’, *The Computer Journal* 3:3 (1960), ⟨URL: http://comjnl.oxfordjournals.org/cgi/content/abstract/3/3/168⟩, pp. 168–174.

Brooker, R. A. and Morris, D., ‘Some Proposals for the Realization of a Certain Assembly Program’, *The Computer Journal* 3:4 (1961), ⟨URL: http://comjnl.oxfordjournals.org/cgi/content/abstract/3/4/220⟩, pp. 220–231.

Bumgarner, L. L. and Feliciano, M., ‘ALCOR Group Representation of ALGOL Symbols’, *Commun. ACM* 6:10 (1963), pp. 597–599.

Campbell-Kelly, Martin, *Computer: a history of the information machine* (BasicBooks, 1996).

Dijkstra et al., ‘Minority Report’, *ALGOL Bull.* 31 (1970), p. 7.

Dijkstra, E. W., ‘Recursive Programming’, *Numerische Mathematik* 2 (oct 1960), pp. 312–318.

Dijkstra, E. W., ‘An ALGOL 60 Translator for the X1’, in: Goodman, Richard, editor, *Annual review in automatic programming 3* (1963), pp. 329–345.

Dijkstra, E. W., ‘Making a Translator for ALGOL 60’, in: Goodman, Richard, editor, *Annual review in automatic programming 3* (1963), pp. 347–356.

Dreyfus, P., *General Panel Discussion. Are Extensions to ALGOL 60 necessary and if so What Ones?*, in: *Symbolic Languages in Data Processing* (London: Gordon and Breach Science Publishers, 1962), pp. 811–832.

Duncan, F. G. and Wijngaarden, A. van, ‘Cleaning up ALGOL60’, *ALGOL Bull.* 16 (1964), pp. 24–32.

Duncan, Fraser G., ‘ECMA Subset of ALGOL 60’, *Commun. ACM* 6:10 (1963), pp. 595–597.

Flamm, Kenneth, *Creating the Computer* (The Brookings Institution, 1988).

Ginsburg, Seymour and Rice, H. Gordon, ‘Two Families of Languages Related to ALGOL’, *J. ACM* 9:3 (1962), pp. 350–371.

Gorn, S., ‘Report on SUBSET ALGOL 60 (IFIP)’, *Commun. ACM* 7:10 (1964), pp. 626–628.

Grau, A. A., ‘Recursive processes and ALGOL translation’, *Commun. ACM* 4:1 (1961), pp. 10–15.

Hoare, C. A. R., ‘Case expressions’, *ALGOL Bull.* 18 (1964), pp. 20–22.

Hoare, C. A. R., ‘Record Handling’, *ALGOL Bull.* 21 (1965), pp. 39–69.

Hockney, R. W., ‘A Proposed Extension to Algol 60’, *ALGOL Bull.* Sup 12 (1961), pp. 1–12.

IBM, Programming Research Group, ‘Preliminary Report – Specifications for the IBM Mathematical FORmula TRANslating System FORTRAN’, Technical report (New York: IBM, 1954), ⟨URL: http://community.computerhistory.org/scc/projects/FORTRAN/BackusEtAl-PreliminaryReport-1954.pdf⟩.

Ingerman, P. Z., 'Thunks: a way of compiling procedure statements with some comments on procedure declarations', *Commun. ACM* 4:1 (1961), pp. 55–58.

Irons, E. T. and Acton, F. S., 'A proposed interpretation in ALGOL', *Commun. ACM* 2:12 (1959), pp. 14–15.

Irons, E. T. and Feurzeig, W., 'Comments on the Implementation of Recursive Procedures and Blocks in Algol-60', *ALGOL Bull.* Sup 13.2 (1960), pp. 1–15.

Irons, Edgar T., 'A syntax directed compiler for ALGOL 60', *Commun. ACM* 4:1 (1961), pp. 51–55.

Irons, Edgar T., 'The Structure and Use of the Syntax Directed Compiler', in: Goodman, Richard, editor, *Annual review in automatic programming 3* (1963), pp. 207–227.

Jensen, J. and Naur, Peter, 'An Implementation of Algol 60 Procedures. (preprint from Nordisk Tidskrift for Informations-Behandling, Volume 1, No 1 -1961)', *ALGOL Bull.* Sup 11 (1961), pp. 38–47.

Knuth, D. E., 'A proposal for input-output conventions in ALGOL 60', *Commun. ACM* 7:5 (1964), pp. 273–283.

Knuth, D. E., 'A list of the remaining trouble spots in ALGOL60', *ALGOL Bull.* 19 (1965), pp. 29–38.

Knuth, Donald E., 'backus normal form vs. Backus Naur form', *Commun. ACM* 7:12 (1964), pp. 735–736.

Knuth, Donald E., 'A History of Writing Compilers', in: Pollack and Bary, W., editors, *Compiler Techniques* (Princeton, 1972), pp. 38–56.

Knuth, Donald E. and Pardo, Luis Trabb, 'Early Development of Programming Languages', in: Belzer, Jack, Holzman, Albert G. and Kent, Allen, editors, *Encyclopedia of Computer Science and Technology*, volume 7 (Marcel Dekker INC., 1975), pp. 419–493.

Kruseman Aretz, F.E.J., *The Dijkstra-Zonneveld ALGOL 60 compiler for the Electrologica X1* (Amsterdam: CWI, 2003), ⟨URL: http://ftp.cwi.nl/CWIreports/SEN/SEN-N0301.pdf⟩, historical note SEN, 2.

Ledley, Robert S. and Wilson, James B., ‘Automatic-programming-language translation through syntactical analysis’, *Commun. ACM* 5:3 (1962), pp. 145–155.

Lindsey, C. H., *A history of ALGOL 68*, in: *HOPL-II: The second ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1993), pp. 97–132.

Lucas, P., ‘The Structure of Formula-Translators’, *ALGOL Bull.* Sup 16 (1961), pp. 1–27.

Mahoney, Michael S., ‘Computer Science. The Search for a Mathematical Theory’, in: Krige, John and Pestre, Dominique, editors, *Science in the 20th Century* (Amsterdam: Harwood Academic Publishers, 1997), ⟨URL: http://www.princeton.edu/~mike/articles/20thcSci/20thcent.html⟩, An electronic version at http://www.princeton.edu/ mike/articles/20thcSci/20thcent.html is used. Last visited 19 April 2006..

McCracken, Daniel D., ‘A New Home for ALGOL’, *Datamation* 5 (1962), pp. 44–46.

Naur, P., ‘Proposals for a new language’, *ALGOL Bull.* 18 (1964), pp. 26–43.

Naur, Peter, ‘ALGOL 60 Maintenance’, *ALGOL Bull.* 10 (1960), pp. 1–10.

Naur, Peter, ‘ALGOL 60 Maintenance’, *ALGOL Bull.* 11 (1960), pp. 1–4.

Naur, Peter, ‘ALGOL translator characteristics and the progress in translator construction’, *ALGOL Bull.* 10 (1960), pp. 14–16.

Naur, Peter, ‘The discontinuation of the ALGOL Bulletin’, *ALGOL Bull.* 15 (1962), pp. 2–3.

Naur, Peter, ‘The Questionnaire’, *ALGOL Bull.* 14 (1962), pp. 1–14.

Naur, Peter, ‘The form of specifications’, *ALGOL Bull.* 22 (1966), pp. 14–14.

Naur, Peter, ‘Successes and failures of the ALGOL effort’, *ALGOL Bull.* 28 (1968), pp. 58–62.

Naur, Peter, *The European side of the last phase of the development of ALGOL 60*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), pp. 15–44.

Naur, Peter, *Transcripts of Presentations*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), pp. 147–161.

Perlis, A. J. and Samelson, K., 'Preliminary Report: International Algebraic Language', *Commun. ACM* 1:12 (1958), pp. 8–22.

Perlis, Alan J., *The American side of the development of Algol*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), pp. 3–14.

Perlis, Alan J., *Transcripts of Presentations*, in: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages* (New York, NY, USA: ACM Press, 1978), pp. 139–147.

Poel, W. L. van der, *Some Notes on the History of ALGOL*, in: *A Quarter Century of IFIP* (Amsterdam: Elsevier, 1986), pp. 373–392.

Pressroom, ACM, 'Software Pioneer Peter Naur Wins ACM's Turing Award. Dane's Creative Genius Revolutionized Computer Language Design' (2006), ⟨URL: `http://campus.acm.org/public/pressroom/press_releases/3_2006/turing_3_01_2006.cfm`⟩.

Rosen, Saul, 'Programming Systems and Languages. A Historical Survey', in: Idem, editor, *Programming systems and languages* (London: McGraw-Hill, 1967), pp. 3–22.

Rutishauser, H., 'Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen', *Z. Angew. Math. Mech.* 32:3 (1952), pp. 312–313.

Rutishauser, Heinz, *Description of ALGOL 60*, volume 1, edited by Bauer, F. L. (Springer-Verlag, 1967).

Samelson, K. and Bauer, F., *The ALCOR project*, in: Gordon and Breach, editors, *Symbolic languages in data processing: Proc. of the Symp. organized and edited by the Int. Computation Center, Rome, 26-31 March 1962* (New York, 1962), pp. 207–217.

Samelson, K. and Bauer, F. L., 'Sequential formula translation', *Commun. ACM* 3:2 (1960), pp. 76–83.

Sammet, Jean E., *Programming languages : history and fundamentals*, Series in Automatic Computation (Englewood Cliffs, N. J.: Prentice-Hall, 1969).

Schwarz, H. R., 'The Early Years of Computing in Switzerland', *Annals of the History of Computing* 3:2 (1981), pp. 121–132.

Wegstein, J. H., 'From formulas to computer oriented language', *Commun. ACM* 2:3 (1959), pp. 6–8.

Wegstein, J. H., 'Algorithms: Anouncement', *Commun. ACM* 3:2 (1960), p. 74.

Wegstein, J. H., 'ALGOL: a critical profile. The RAND Symposium, part two', *Datamation* 10 (1961), pp. 41–45.

Wijngaarden, A. Van, 'Orthogonal design and description of a formal language' (October 1965), ⟨URL: http://www.fh-jena.de/~kleine/history/languages/VanWijngaarden-MR76.pdf⟩, MR 76.

Wijngaarden, A. Van, *Draft Report on the Algorithmic Language ALGOL 68*, edited by Idem (Amsterdam: Mathematisch Centrum, 1968), MR 93.

Wijngaarden, A. Van, *Report on the Algorithmic Language ALGOL 68*, edited by Idem (Amsterdam: Mathematisch Centrum, 1969), MR 101.

Wirth, N., 'Additional Notes on - Contribution to the Development of ALGOL', *ALGOL Bull.* 24 (1966), pp. 13–17.

Wirth, N., 'Closing word at Zurich Colloquium', *ALGOL Bull.* 29 (1968), pp. 16–19.

Wirth, Niklaus, 'A generalization of ALGOL', *Commun. ACM* 6:9 (1963), pp. 547–554.

Wirth, Niklaus and Hoare, C. A. R., 'A contribution to the development of ALGOL', *Commun. ACM* 9:6 (1966), pp. 413–432.

Wirth, Niklaus and Weber, Helmut, 'EULER: a generalization of ALGOL and it formal definition: Part 1', *Commun. ACM* 9:1 (1966), pp. 13–25.

Wirth, Niklaus and Weber, Helmut, 'EULER: a generalization of ALGOL, and its formal definition: Part II', *Commun. ACM* 9:2 (1966), pp. 89–99.

Wood, Derick, 'A few more trouble spots in ALGOL 60', *Commun. ACM* 12:5 (1969), pp. 247–248.

Zuse, Konrad, 'Some Remarks on the History of Computing in Germany', in: Metropolis, N., Howlett, J. and Rota, Gian-Carlo, editors, *A History of Computing in the twentieth century* (Academic Press, 1980), pp. 611–627.