Dijkstra's Rallying Cry for Generalization: The Advent of the Recursive Procedure, late 1950s — early 1960s

Edgar G. Daylight*

Autumn, 2010

Abstract

According to J.A.N. Lee in 1996, computer scientists "are reaching the stage of development where each new generation of participants is unaware both of their overall technological ancestry and the history of the development of their speciality, and have no past to build upon" [1, p.54]. A technically and historically accurate account, as attempted here, can help us, computer scientists, grasp some of the fundamental ideas underlying our discipline. This paper describes some early contributions of E.W. Dijkstra by elaborating on his involvement in putting forward and implementing the recursive procedure as an `ALGOL60` language construct. Particular attention is paid to Dijkstra's generalizing style of solving problems.

Keywords: recursive procedure, machine independence, `ALGOL`, Edsger Dijkstra

1 Introduction

Half a century ago, the 31-year old Dutchman, Edsger W. Dijkstra (1930–2002), was sitting in Rome's "Palazzo dei Congressi" attending the Panel Discussion on 'Philosophies for Efficient Processor Construction' at the *International Symposium of Symbolic Languages in Data Processing* (March, 1962). Together with Naur, Duncan, and Garwick, he was one of the few strong proponents of the recursive procedure in the `ALGOL60` programming language. Even though he had become famous more than a year before the symposium by being one of the first to build an `ALGOL60` compiler that could handle recursive procedures, a large group of panel members remained sceptical about its usefulness.

*He can be contacted by email (*egdaylight@dijkstrascry.com*) or by postal mail (Sint Lambertusstraat 3, B-3001 Heverlee, Belgium). He has a PhD in computer science, was recently a post-doc in History of Computing at the FNWI Institute voor Informatica (University of Amsterdam), and has the Belgian nationality.

Inspection of the proceedings [2] shows that almost every panel member had a slightly different view towards why the recursive procedure should or should not belong to a machine-independent programming language, such as `ALGOL60`. For instance, and as is explained in greater detail later, Dijkstra heavily supported the recursive procedure due to linguistic reasons. In contrast, Strachey and Samelson claimed that general language constructs, such as the recursive procedure, typically led to inefficient object programs. Strachey wanted to restrict (but not necessarily discard) the use of recursive procedures [2, p.368,373]. Samelson, on the other hand, was primarily concerned with the immediate economical considerations: "the final judge in matters of efficiency is money". Samelson wanted to minimize the financial cost of a complete project: designing a programming language, building a compiler, compiling programs, and executing those programs. In his opinion, the efficiency of the running program influenced the total cost the most and, therefore, he preferred to avoid the recursive procedure [2, p.364,372].

The tension between several panel members was apparent [2, p.373]. For instance, Naur's views, which were very similar to those of Dijkstra, were in sharp contrast to Samelson's economic considerations. And, Seegmüller's nasty but loudly applauded comment certainly did not help ease the tension:

> And the question is –to state it once more– that we want to work with this language, really to work and not to play with it, and I hope we don't become a kind of Algol play-boys. [2, p.375]

The comment was directed towards Dijkstra, Naur, and other linguists; i.e., researchers who favored general language constructs.

In McCarthy's absence, Garwick seems to have been the only person at the conference to have openly claimed the potential usefulness of the recursive procedure as a programming construct [2, p.369]. In hindsight unsurprisingly, given that `ALGOL60`'s main application domain was numerical analysis[1] and that, even during the early 1960s, the recursive procedure was not used by numerical analysts. For instance, only by 1963 did the Swiss mathematician Rutishauser find two examples of recursion for numerical computations which he himself found convincing. He also contrasted his two examples with others[2] in which recursion could, and in his opinion should, be replaced by iteration [5].

The `ALGOL60` report [6], published in May 1960, quickly led to non-numerical applications of the recursive procedure. It for instance enabled Hoare to express his intuitive ideas on sorting, and resulted in his now-famous `QuickSort` [7], which he published as a recursive `ALGOL60` program [8, p.145]. Besides the recursive procedure, `ALGOL60` was also innovative due to its recursive syntax, formalized in Backus Naur Form (`BNF`) notation. The formalized recursive syntax, in turn, led several researchers to design their `ALGOL60` compilers as a collection of mutually recursive procedures. Grau, for instance, described in a

[1] See [3, p.121-122] and [4, p.100-101].

[2] An example Rutishauser gave was calculating the factorial of a positive number n. It is more economical (both in space and time) to calculate it by iteration by means of a *for* loop than by recursive-procedure activations.

pseudo ALGOL60 language what we would today call a recursive-descent compiler[3]. A similar but later example is Hoare's compiler for the Elliott machine (cf. [8, p.146]).

Contributions of this Paper

If numerical analysts did not use recursive procedures in their work, and if ALGOL60 was primarily designed for numerical computations, why then did ALGOL60's official definition [6] include the recursive procedure as a language construct? And, why was it such a controversial topic in Rome 1962? As is elaborated on later, McCarthy had, based on his experience with recursion in LISP, unsuccessfully attempted to put forward the recursive procedure as an ALGOL60 language construct. Instead, it were the linguistically-minded Amsterdammers, Van Wijngaarden and Dijkstra, who had convinced the editor of the ALGOL60 report, Naur, to include recursive procedures in ALGOL60. However, these three men knew that recursive procedures were heavily opposed by other influential ALGOL60 researchers. Why then did the Dutch persevere, given that they did not need it in their programming?

ALGOL60's definition was discussed on the international scene during the late 1950s and it was officially defined in 1960. Its definition [6] strongly suggests that it had been defined without having a particular machine in mind: its syntax was presented with the help of a formal language (BNF, discussed later) and its semantics were stated without mentioning specific machine features. Placing ALGOL60 in an historical context, however, suggests otherwise. The programming systems prior to ALGOL60 had typically been defined in terms of specific machine features and by several researchers who would later become ALGOL60 actors. It therefore seems rather unlikely that all ALGOL60 actors were indeed reasoning machine independently during the late 1950s and early 1960s. In this paper, several people (Bauer, Samelson, Strachey, and Wilkes) will be mentioned in this regard as researchers who primarily reasoned in terms of machine efficiency and less so as linguists (à la Van Wijngaarden and Dijkstra) who focused on general language constructs prior to dealing with machine-specific problems.

The shared ambition, among all ALGOL60 actors, was to implement a universal, machine-independent programming language in conformance with the ALGOL60 report [6]. But it was a priori not clear to many ALGOL60 actors whether all of its language constructs were indeed implementable [11, p.12-13]. In practice, various ALGOL60 dialects were implemented, dialects which were influenced by local programming habits and specific machine features. For example, the British researchers Strachey and Wilkes wanted to restrict the official ALGOL60 definition –and the use of recursion in particular– due to specific features of contemporary machines, as explained in [12] and as heavily criticized by Dijkstra [13, 14, p.16-18, p.40-42].

[3] He published his recursive pseudo-ALGOL60 program in [9]; i.e., before the publication of Hoare's QuickSort. And, also Irons [10] should be mentioned in this regard.

Most early ALGOL60 compilers were not able to handle all language constructs and the recursive procedure in particular[4]. Dijkstra and Zonneveld, by contrast, did succeed in building one of the first ALGOL60 compilers which could handle almost all of the language, including the recursive procedure. It was completed in August 1960 and impressed several researchers. In Naur's words:

> The first news of the success of the Dutch project, in June 1960, fell like a bomb in our group. [18, p.119]

Essential to the Dijkstra-Zonneveld compiler was its run-time stack. Yet, the stack had long been invented by several independent researchers and its run-time usage was not novel either, as Dijkstra also explicitly acknowledged when he described some of the key ideas underlying the Dijkstra-Zonneveld compiler [19, p.313]. On the other hand, it is equally clear that the Dijkstra-Zonneveld compiler was indeed a technical innovation, as Rosen's comment illustrates:

> Almost everyone involved in writing an Algol compiler has used some of the ideas developed in connection with the Algol Compiler written by professor Dijkstra and his colleagues at the Mathematisch Centre in Amsterdam. [20, p.181]

What, then, made the Dijkstra-Zonneveld compiler so special?

Answers to the aforementioned questions are presented in the sequel by describing the work of some key players who were involved in the ALGOL60 effort. By doing so, two main messages are conveyed in this paper.

The first message is that the early history of programming languages can be viewed as a dichotomy between specialization and generalization. Specialization refers to language restrictions, static (compile-time) solutions, and the exploitation of machine-specific facilities —in the interest of efficiency. Generalization refers to general language constructs, dynamic (run-time) solutions, and machine-independent language design —in the interest of correctness and reliability. The dichotomy becomes effective if we keep in mind, throughout the sequel, that most ALGOL60 researchers were neither completely specialists nor generalists, and that they initially did *not* characterize each other as such. Only after 1960 did the dichotomy become increasingly apparent.

The second message conveyed in this paper is that Dijkstra's continual appeal for generalization led to practical breakthroughs in compiler technology, while some prominent ALGOL60 researchers who favored language restrictions in the interest of obtaining immediate practical results failed in their endeavours. On the other hand, however, Dijkstra's successes will be put into perspective as well, by showing that the dichotomy outlived the ALGOL60 effort, contrary to the claims Dijkstra made in Munich, half a year after the Rome conference.

Three disclaimers conclude this introduction. First, an historical-accurate narrative, as attempted here, often implies mathematical inaccuracy with respect to the current state of the art. In this paper, the recursive procedure is

[4]E.g., the ALCOR compilers described in [15], the Danish DASK ALGOL compiler [16, p.441], and SMALGOL61 [17, p.502].

described in terms of what computer practitioners of the late 1950s and early 1960s understood by it. Therefore, the recursive procedure is presented informally and without any mention of termination proofs. Likewise, noting, in hindsight, that Dijkstra, McCarthy, and others had made mistakes in their pioneering work has practical relevance [21, 22], but lies outside the scope of this paper. Second, the history of ideas is much less concerned with "firsts" than it is with the contextual development of ideas [1]. For example, the fact that Grau and Hoare, mentioned above, used `ALGOL60`'s recursive procedure in unanticipated ways is far more important than noting that Grau did so before Hoare. Third, this paper presents a *synthesis* of Dijkstra's work on `ALGOL60`. It therefore most definitely does not address all of his contributions. When discussing some of his papers, I will often only discuss parts of their contents.

2 Specialization versus Generalization

The end of World War II coincided with the beginning of the computer era. As two victors of the war, the USA and England were among the first to build computers. Continental Europe, by contrast, was in turmoil. Relying heavily on the Marshall Plan for economic revival, several continental-European researchers traveled to the USA and England to acquire knowledge in computing. In 1947, for example, the Dutch mathematician Van Wijngaarden visited England and the USA [23, p.102], and in 1949 Rutishauser from Switzerland visited the computer pioneers Aiken at Harvard and Von Neumann at Princeton [24, p.2].

2.1 USA

Besides building computing machines, American researchers were also quick in seeking easier ways to instruct their machines. In May 1954, the American Navy organized a conference in Washington D.C. entitled *Automatic Programming for Digital Computers* [25]. Instead of having a programmer tediously write down machine code, the conference attendees wanted to be able to provide the programmer with a more mathematical notation in which he could express himself more easily. The research challenge was to design a computer program that could automatically translate the mathematical expressions of the programmer into the instructions of the machine. Various automatic-translation programs were presented and discussed at the conference.

Most presentations at the 1954 conference covered mathematical notations and automatic-translation programs that only worked for a specific kind of machine. Two exceptions, however, were the presentations of Gorn [26] and Brown & Carr [27]. These three researchers discussed translation techniques that were applicable for any type of machine. And, to obtain such a general technique, they realized that the mathematical notation (intended for the programmer) had to be independent of any computing machine. That is, the mathematical notation had to be a *machine-independent* language. Furthermore, Gorn, Brown, and Carr sought a *universal* machine-independent language; i.e., a language

that was close to the universal language of mathematics and, hence, applicable to a large class of mathematical problems.

To appreciate the extreme stance taken by Gorn, Brown, and Carr, two observations are in order. First, the aspired universal, machine-independent language embodied generality in two ways: it was intended for various mathematical problems and a variety of computing machines. Second, given the limited storage sizes and execution speeds of contemporary computing machines, programmers in the 1950s did their utmost best to optimize the efficiency of their programs. That is, they applied special programming tricks in order to obtain programs that were economical in terms of program size and computation time. To apply such tricks, they exploited specific details of the mathematical problem that their computer program was intended to solve and they also exploited specific details of their computing machine. Hence, specialization –in contrast to generalization– was the prime occupation in computing during the 1950s. A general language, such as a universal machine-independent language, was viewed by many as unrealistic, because of the inefficiencies it would incur.

Due to its generality, Brown and Carr acknowledged that their aspired language would, indeed, incur a run-time penalty in efficiency, compared to existing machine-specific programming techniques. But, according to them, this penalty would be outweighed by a decrease in programming time and programming errors. For, by being completely ignorant of what machine would execute his mathematical expressions, the programmer only had to convert his mathematical problem into the mathematical notation of the universal language. He did not have to incorporate machine-specific characteristics in his manual labor. To get this message across, Brown and Carr advocated for an over-all measure of effectiveness, which included the new criteria of programming time and program correctness, along with the more traditional criteria of program size and computation time [27, p.89].

2.1.1 Speedcoding & FORTRAN

Also present at the 1954 conference were Backus and Herrick. They described a high-level *and* machine-dependent system, called Speedcoding. Instead of having to directly program in machine code, the aspiration was that a programmer could solely write down the formulas for the numerical problem at hand (i.e., declaratively). Yet, in order to obtain fast executables, the programmer would also be able to express how the data should be transferred from one storage hierarchy to another [28, p.111-112]. In their own words:

> [T]he question is, can a machine translate a sufficiently rich mathematical language into a sufficiently economical machine program at a sufficiently low cost to make the whole affair feasible? [28, p.112]

To obtain an affirmative answer, Speedcoding and the later `FORTRAN` were designed for *specific* machines and, hence, at the price of machine portability [20, 29, p.11, p.151]. The tendency to specialize also reflects in the program constructs: *do* statements (i.e., *for* loops) had to have static bounds and it was

<digit>	:=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<integer>	:=	<digit> | <integer><digit>
<realPart>	:=	. <integer> | <integer>. |
		<integer>.<integer>
<real>	:=	<realPart> | +<realPart> |
		−<realPart>

Table 1: An example in Backus Naur Form.

not possible to express potentially unbounded *while* loops, nor recursive procedures [29, p.145,159-160].

Backus: from FORTRAN to ALGOL58

After having worked on `FORTRAN`, Backus joined the `ALGOL` effort and made a significant contribution in [30], by devising a formal notation to describe the syntax of `ALGOL58`. Backus's notation almost went unnoticed however; it was Naur who grasped its potential and, who, after making some small but important modifications, used it to define `ALGOL60`'s syntax. The notation is therefore called Backus Naur Form (cf. [31, 32] and [4, p.99]).

To appreciate the conceptual leap that Backus took from `FORTRAN` to `ALGOL58`, it is important to note that, prior to his contribution, computer practitioners described syntax informally (e.g., in verbose English). For example, consider the definition: A real number is

> 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 previous passage is similar to how a real number was defined in `ALGOL58` [33, p.11] in that no mention is made of the finiteness of the machine. By including machine-specific constants, which do express the finite-storage limitations of the machine, the previous definition can be extended to:

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

The previous two passages, together, constitute the original definition of a real number in `FORTRAN` [34].

In short, both `FORTRAN`'s and `ALGOL58`'s syntax were defined informally. The syntax of `FORTRAN` was defined with and that of `ALGOL58` was defined without finite-storage limitations in mind. The informal definitions were ambiguous, incomplete, and often lengthy: `FORTRAN`'s and `ALGOL58`'s syntax were very cumbersome to use in practice [35, p.26-27].

Continuing with the real-number example, the `BNF` equivalent of the first passage, presented above, is depicted in Table 1. With | denoting *or*, Line 1

expresses that a digit is either 0 or 1 or 2 or . . . or 9. Line 2, in turn, *recursively* defines a sequence of digits to be either a digit or an integer concatenated with a digit. Line 3 defines the real part of a real number to either be an integer preceded by or followed by a decimal point or an integer followed by a decimal point and an integer. Finally, Line 4 defines a real number to have no sign, a plus sign, or a minus sign.

Note that Table 1 is only the BNF equivalent of the first passage, presented above. The finite-storage limitations of the machine (cf. the second passage) can not be expressed concisely in BNF notation. Indeed, the syntactic recursion, exemplified in Line 2 in Table 1, is what made BNF notation so concise: Line 2 allows an arbitrarily long but finite integer to be written down in ALGOL60 and, hence, also integers that simply could not fit in every computer's memory!

Backus's conceptual leap of abstracting away the computing machine's finite limitations cannot be stressed enough. Unlike his work during the FORTRAN years, where he focused on the design of the translator to obtain efficient machine code, Backus's abstraction allowed him to *solely* focus on the language. On the one hand, Backus was of course aided by ALGOL58's abstraction of finite storage. On the other hand, as we shall see, many computer practitioners did not let go of machine-specific features while designing and implementing a machine-independent programming language.

2.1.2 List Processing

Not present at the 1954 conference but equally important to mention are Newell, Shaw, and Simon. By 1957, these three men had implemented a list-processing system for automatic theorem proving [36, 37]. Their system was called the Logic Theory Machine (LT) and it served the purpose of trying to better understand how effective human-problem solving works in reality, such as finding a proof of a mathematical theorem, playing chess, or discovering scientific laws from data [37, p.218-219]. They used their Information Processing Language (IPL) to implement LT [36, p.232].

In contrast to the many numerical programs implemented during the 1950s, LT was symbolic in nature. While numerical programs were primarily static in the sense that e.g. the set of variables and constants (to be used at run time) could be determined in advance (i.e., prior to program execution), LT was primarily dynamic. For instance, the number, kind, and order of logical expressions used in LT were completely variable. Therefore, run-time translation was needed and carried out by an interpreter [36, p.230,231,235].

IPL was a very flexible programming language. A user could express the creation of a list during the course of computation. In addition, a user could create lists that consisted of other lists or lists of lists, etc. Adding, deleting, inserting, and rearranging items in a list at any time was possible. Finally, it was also feasible for an item to appear in any number of lists simultaneously [36, p.231]. In modern terms, dynamic memory management and aliasing characterized the work of Newell, Shaw, and Simon.

IPL was not only flexible in terms of memory assignments, but also in terms

of defining processes. There was no limitation on the size and complexity of hierarchical definitions. Likewise, no restriction was enforced on the number of references in the instructions or on what was referenced. Of particular interest is that processes could be defined implicitly, e.g. by *recursion* [36, p.231]. More generally:

> [T]he programmer should be able to specify any process in whatever way occurs naturally to him in the context of the problem. If the programmer has to 'translate' the specification into a fixed and rigid form, he is doing a preliminary processing of the specifications that could be avoided. [36, p.231]

In short, Newell, Shaw, and Simon focused more on the flexibility of their `IPL` language than on machine efficiency. Their first pseudo code was developed in a machine-independent way with the purpose of precisely specifying a logic theory machine. Only *afterwards* did they define `IPL` and in accordance with the `RAND JOHNNIAC` machine [36, p.232]. Not surprisingly, `IPL` had some shortcomings in terms of memory space and computation time, shortcomings which were not considered too problematic:

> [F]or it seemed to us that these costs could be brought down by later improvements, after we had learned how to obtain the flexibility we required. [36, p.232]

Hence, the prime concern was the language and the ease of being able to express oneself in that language for the problem at hand.

McCarthy: from FORTRAN to LISP

Likewise, John McCarthy was, contrary to the many numerical analysts of the 1950s, trying to use recursion in his programming. For, while working at IBM in the summer of 1958, he tried to write a `FORTRAN` program that would differentiate algebraic expressions, such as the expression y^2. To calculate the derivative of y^2 –which is equal to $2y$ times the derivative of y– McCarthy realized that he needed recursive conditional expressions. Since `FORTRAN` did not contain recursion, he tried to add it to the language, but without success. This led him to develop his own list-processing, recursive, command language `LISP` [38, p.27], which was greatly inspired by the work of Newell, Shaw, and Simon [39, p.187].

2.2 Western Europe

Compared to the USA, Western Europe had few computing groups during the 1950s [40, p.352]. One such group was the Swiss-German `ZMD`, which consisted of researchers from Zurich (e.g., Rutishauser), Munich (e.g., Bauer and Samelson), and Darmstadt (e.g., Bottenbruch) [41, p.61]. These three cities collaborated closely in scientific computing but were hindered by the diversity of computing machines: different machines were being built in each of these cities. To overcome this diversity, Rutishauser appealed for a "unified algorithmic notation" in

the 1955 GaMM[5] meeting and the ZMD group subsequently became more involved in researching automatic-translation techniques [35, 41, p.5, p.61].

In May 1958, a one-week ACM-GaMM meeting was held in Zurich, signifying the collaboration between an American delegation, led by Carr, and the ZMD group [33]. The collective focus was to define a universal, machine-independent programming language. The chosen name for the language was initially IAL (International Algorithmic Language), later denoted as ALGOL58, but would by January 1960 change into ALGOL (Algorithmic Language), and is denoted as ALGOL60 in this text [35, 42, p.35, p.79].

The intention of ALGOL58 was threefold [33, p.9].

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.

Besides the Swiss and Germans, also others such as the Dutch, Danes, and British were actively involved in computing during the 1950s. In Amsterdam, for instance, Van Wijngaarden led a team at the 'Mathematisch Centrum' (Mathematical Center). Contrary to Rutishauser, who had a rented computing machine at his disposal as early as 1950 [43, 44, p.24-25, p.41-50], the Amsterdammers had to build their own machine before being able to experiment with programming techniques. It took until January 1954, with the advent of their ARRAII machine, for the Amsterdammers to possess a working computer [23, p.124]. And, only in 1959, with their X1 machine, did the Amsterdammers actively participate in advancing the art of automatic translation; i.e., by joining the ALGOL60 effort. Surprisingly however, the Amsterdammers Dijkstra and Zonneveld succeeded very quickly in building an ALGOL60 compiler (August, 1960). By doing so, Amsterdam instantly became an internationally renown city for those involved in the ALGOL60 effort [23, p.125].

During the second half of the 1950s, the Swiss-German ZMD group extended and changed its name into ALCOR (ALgol COnverteR). Research teams from Copenhagen and Vienna had for instance joined the ALCOR initiative [15, p.210]. Each ALCOR team had its own unique computing machine, but shared the same aspiration: an easy mathematically-oriented programming language (initially ALGOL58, later ALGOL60). Similar to the Gorn-Brown-Carr philosophy, the ALCOR members wanted to be able to write a program once in ALGOL58 and then be able to automatically translate it into the instructions of any ALCOR machine. Furthermore, the ALCOR members wanted efficient translators *and* efficient machine programs [15, p.210]. The latter ambition, however, conflicted with their quest for a universal machine-independent language. For, recall from

[5] Gesellschaft für Angewandte Mathematik und Mechanik (Association of Applied Mathematics and Mechanics).

Brown and Carr's work that efficient machine programs and machine independence did not mix well, as Bauer and Samelson's own words from 1962 also seem to indicate:

> The [ALCOR] group was in agreement from the beginning, partly as a result of the experiment in machine design, to devise a [universal, machine independent] language and translator as an efficient programming tool *using available machine facilities to the fullest extent possible*. [15, p.210, my italics]

As we shall see later, Dijkstra, by contrast, designed a language after having first abstracted away from the machine. A similar remark holds concerning the design of the Dijkstra-Zonneveld compiler.

2.3 The Recursive Procedure Enters ALGOL60

As part of the ALGOL60 effort, a subcommittee in November 1959 (consisting of Rutishauser, Ehrling, Woodger, and Paul) recommended that certain restrictions be put in place with respect to the parameters of a procedure, thereby automatically preventing recursive activation through a parameter. According to Perlis, recursion in general was therefore not explicitly forbidden [42, p.86]. According to Naur, however, it was:

> We should also mention that this procedure concept of the old language introduced for every procedure body a completely closed universe of names. There was no communication to the outer world except through parameters. In this way of course, recursive activations were ruled out by the language itself. [4, p.151,154]

In August 1959, McCarthy wrote a letter in which he openly advocated for recursive procedures [45], and, in January 1960, at the final ALGOL60 Paris conference, McCarthy suggested to add recursive procedures to the ALGOL60 language [42, 38, p.86, p.30]. With regards to McCarthy's proposal to add recursive procedures, an American representative to the ALGOL60 conference proposed to add the delimiter **recursive** to the language, to be used in the context **recursive procedure**. The American's proposal was, by voting, turned down by a narrow margin [4, p.112]. According to some, this rejection was interpreted to mean that recursive procedures should not be added to the ALGOL60 language; others, however, interpreted it to mean that recursive procedures should not be distinguished syntactically from nonrecursive procedures by means of the proposed delimiter [4, p.160]. The latter category of people, therefore, did assume that recursive procedures (introduced by McCarthy) belonged to the ALGOL60 language, while the former category of people –including Naur and McCarthy [4, p.159-160]– assumed that recursive procedures did not belong to the language. In short, and in Perlis's words: "it is not clear what the votes meant!" [4, p.160].

The voting took place before other issues, concerning the (informal) semantics of procedures, had been clarified [4, p.160-161]. After the voting, and after several modifications were made to the semantics of procedures, the defined

language, from Naur's perspective, contradicted the November 1959 proposal to prohibit recursion [4, p.112]. For, Van Wijngaarden and Dijkstra had stumbled upon the possibility to syntactically express recursive-procedure activations. Not sure whether recursion was indeed intended semantically, they contacted the `ALGOL` editor, Naur, by telephone on approximately 10 February 1960 to convey this ambiguity [4, p.112]. After consulting Dijkstra, Van Wijngaarden suggested to Naur to add the following clarification to the `ALGOL` report:

> Any other occurrence of the procedure identifier within the procedure body denotes activation of the procedure. [6, p.311]

Hence, this made explicitly clear that recursive procedures could be expressed in `ALGOL60`. The fact that the alternative, of preventing recursive-procedure activations by means of several language restrictions, would be cumbersome, was also mentioned by Van Wijngaarden[6]. It is, in hindsight, clear that Van Wijngaarden and Dijkstra were primarily reasoning along linguistic lines and *not* in terms of specific machine features. As we shall see, simplicity for them meant less language restrictions.

Naur had initially been in favor of `ALCOR`'s efficiency-driven philosophy. But he became charmed by the one-sentence clarification of the Dutch and added it to the `ALGOL60` report. According to Naur's 1978 recollections:

> I got charmed with the boldness and simplicity of this [one-sentence] suggestion and decided to follow it in spite of the risk of subsequent trouble over the question (cf. Appendix 5, Bauer's point 2.8 and the oral presentation). [4, p.112-113]

Naur's reference to Bauer shows that he was well aware of `ALCOR`'s strong will to prohibit introducing recursive procedures in the language, in accordance with the November 1959 meeting[7].

3 Dijkstra's Continual Appeal for Generalization

Van Wijngaarden's team in Amsterdam spent most of the 1950s building computing machines and corresponding machine languages. The programmer Dijkstra typically had to construct the machine's software on paper, awaiting for the machine to be built by his colleagues Loopstra, Scholten, and Blaauw [23, p.111]. Only in 1959, did the Amsterdammers join the `ALGOL60` effort by actively pursuing a definition of the `ALGOL60` language and a corresponding translation technique. In hindsight, the Amsterdammers' lack of expertise in translation technology and machine-independent languages may have been a blessing in disguise, as Dijkstra's words from 1980 indicate:

[6]Cf. [4, p.112]. Today we know that it is almost impossible to write recursive grammar rules preventing the use of recursion. This was, however, not obvious to many `ALGOL60` actors at the time.

[7]Further support for this claim, involving also Van Wijngaarden and Dijkstra, can be found by reading Bauer's remarks in [4, Appendix 5] and, in particular, Bauer's words: "the Amsterdam plot on introducing recursivity" [4, p.130].

> The combination of no prior experience in compiler writing and a new machine [the X1] without established ways of use greatly assisted us in approaching the problem of implementing ALGOL60 with a fresh mind. [46, p.572]

Van Wijngaarden and Dijkstra viewed a programming language, such as ALGOL60, as a mathematical object. If certain aesthetic criteria were met, they would consider the language to be elegant and, hence, relevant. Only after such a language was defined, did they seek an automatic-translation technique. Their working style differed radically from the efficiency-driven approach followed by Bauer, Samelson, Strachey, Wilkes, and others.

3.1 In Search for a Simple Language

An important aesthetic criterion for the Dutch was generality: any unnecessary language restriction had to be avoided at all costs, in the interest of obtaining a simple language. Dijkstra clarified this point in a later publication [14] by making an analogy between ALGOL60 and the English language. He suggested considering any English text that respects five restrictions:

1. Words of more than 15 letters are forbidden.
2. The total number of letters of three consecutive words may not be greater than 40.
3. Sentences of more than 60 words are not allowed.
4. In one and the same sentence, the same word may not be used twice as a subject.
5. A list of 2000 words is given and each word in that list may not be used.

Dijkstra remarked that the readability of any text respecting restrictions 1-5 is not necessarily hindered and one can read such a text while being completely ignorant of the restrictions. But, constructing a correct English text in conformance with restrictions 1-5 is problematic, due to a lack of intuition when trying to comprehend the restrictions. In the interest of clarity and program correctness, Dijkstra did not want a language such as ALGOL60 to contain restrictions similar to 1-5 either. For instance, a procedure that can call another procedure but not itself was an unnecessary language restriction for Dijkstra. By discarding it, a more general and hence simpler language was obtained; i.e., a language which could handle recursive procedures.

In his paper [14], Dijkstra applied his ideology to ALGOL60 by advocating for *dynamic* instead of static constructions, since they make the language more systematic and powerful. One of Dijkstra's examples was based on the procedure and switch declarations in ALGOL60. The switch is a vector of statement labels, declared at the beginning of a block. Its declaration looks syntactically like an assignment statement. Both the procedure and switch declaration have a hybrid nature; i.e., an undesirable property according to Dijkstra. On the one hand, the procedure and switch declarations both reserve an identifier for a

special sort of object and that object is defined statically; i.e., immediately. In this sense, both the procedure and switch declarations are similar to the 'constant' declaration. On the other hand, however, while a constant number can be used in an assignment statement which dynamically assigns a value, a procedure or switch declaration can not be used in such a dynamic manner. Dijkstra therefore suggested to extend the concept of 'assignment of a value' so that lists, statements, etc. can also act as assigned values. This, in turn, would allow one to remove the value-defining function of the procedure and switch declarations. The result would then be that the declarators *procedure* and *switch* would only be followed by a list of identifiers, to which suitable assignments would eventually be made at run time [14, p.35-36]. According to Dijkstra:

> [Such a modification] is an improvement: the language then becomes more systematic and more powerful at the same time, as *all value-relations* have now become *dynamic*. [14, p.36, my italics]

Dijkstra's emphasis on general language constructs and corresponding dynamic implementations would, as many observed, have a negative effect on computation time. In Strachey's words:

> I think the question of simplifying or reducing a language in order to make the object program more efficient is extremely important. I disagree fundamentally with Dijkstra, about the necessity of having everything as general as possible in all possible occasions as I think that this is a purely theoretical approach [...][8]

It is therefore no surprise that Dijkstra and his fellow linguists were the laughing stock of Seegmüller's well-received comment at the 1962 Rome symposium. Indeed, for most people at the symposium efficiency was *the* prime concern.

A closer look at Dijkstra's ideology, however, shows that his agenda was not to neglect efficiency issues per se, but to focus on the more general objective of increasing programming comfort. In Dijkstra's words:

> In order to get as clear a picture as possible of the real needs of the programmer, I intend to pay, for a while, no attention to the well-known criteria 'space and time'. Those who on the ground of this remark now doubt the honest fervour with which the following is written, should remember that, in the last instance, a machine serves one of its highest purposes when its activities significantly contribute to our *comfort*. [14, p.30, my italics]

In other words, to better understand the real underlying problems of programming, Dijkstra suggested to temporarily ignore (i.e., abstract away) machine-specific features. While a decrease in execution time or memory footprint may, indeed, contribute to an increase in programming comfort, other criteria, such as program correctness, could contribute much more. According to Dijkstra:

[8] Cited from [2, p.368]. Incidentally, it is interesting to note that Strachey used the verb 'simplifying' to denote the opposite action of what Dijkstra associated with that verb.

> I am convinced that these problems [of program correctness] will prove to be much more urgent than, for example, the exhaustive exploitation of specific machine features, if not now, then at any rate in the near future. [14, p.30]

As a final example, by countering Strachey and Wilkes's efficiency-driven proposal to explicitly delimit the use of recursive procedures, Dijkstra clearly explained where he stood on these matters:

> [Strachey and Wilkes] make an appeal to the fact that "... a recursive procedure is both longer and slower than a non-recursive one." But the recursive procedure is such a neat and elegant concept that I can hardly imagine that it will not have a marked influence on the design of new machines in the near future. And this influence could quite easily be so considerable, that the possible gain in efficiency that can still be booked by excluding recursiveness, will become negligible. [13, 14, p.17, p.41]

To clarify, Strachey and Wilkes were in favor of static solutions enforced by language restrictions in the interest of machine efficiency. They viewed the requirement that all `ALGOL60` procedures could potentially be recursive as an example of "unnecessary generality" [13, 14, p.16, p.40]. Instead, they wanted all procedures to be nonrecursive by default and only recursive if explicitly delimited by the programmer. By doing so, it would be possible to write compilers that could generate far more efficient machine code. Dijkstra, by contrast, wanted to avoid such case distinctions in order to obtain a simple language and corresponding implementation technique. To do so, he advocated for general language constructs and corresponding dynamic solutions. Dijkstra acknowledged that his general recursive-programming approach led to inefficient machine code, but he also stressed that this would probably be resolved in the nearby future (cf. [14, p.41] and his abstract in [19]). In hindsight, it seems that Dijkstra had anticipated the advent of the hardware stack.

To conclude, Dijkstra's philosophy is in many ways similar to the Gorn-Brown-Carr philosophy of 1954. And, his appeal for dynamic constructs is, albeit for different reasons, similar to the work of Newell, Shaw, and Simon. Furthermore, Dijkstra believed that efficiency problems would be resolved in the nearby future, or at least become negligible. According to Dijkstra, generalization of a programming language allowed for simplification in compiler building and this would in the long term prevail over the short-term engineering problems that concerned people such as Bauer, Samelson, Strachey, and Wilkes.

3.2 Generalizations of the Stack Principle

Dijkstra's urge to generalize was not only felt at the level of language definition. Also when designing the Dijkstra-Zonneveld compiler did he seek general principles. For example, Dijkstra generalized the manner in which a stack was used in two essential ways. First, he showed how the functionality of the stack can be further generalized in time, by not only using it to evaluate arithmetic

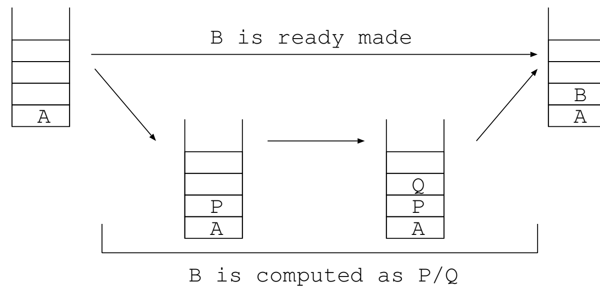

Figure 1: Generalizing the functionality of the stack.

expressions, but also expressions containing procedure calls. That is, procedures were treated like arithmetic expressions by means of some additional bookkeeping [19]. Second, he showed how several different kinds of items (operators, operands, states, priorities, etc.) can be stored on one general stack, instead of using multiple specialized stacks for each kind of item [48].

3.2.1 Further Generalization in Time

By the late 1950s, the stack had been invented over and over again by several independent researchers[9]. It was the "stack principle" of Bauer and Samelson [50] which Dijkstra referred to in his now-famous 1960 article [19], in which he elaborated on how to use the stack as a run-time object.

Figure 1 helps to illustrate the concepts underlying Dijkstra's generalization. The top-most horizontal arrow shows how B is placed on the stack, assuming that B is ready made. If, however, B is not ready made but, instead, has to be calculated by means of the formula $B = P/Q$, then the alternative sequence of arrows in the figure is applicable: P and Q are placed on the stack and eventually removed from the stack in order to compute P/Q.

The generalization lies in the fact that a stack can be used for both scenarios: whether B is ready made or has to be computed, by temporarily using a number of next stack locations, the net result remains the same. More generally, and not explicitly shown in Figure 1, B may just as well be computed by means of a call to a procedure which contains the expression P/Q. In Dijkstra's words:

> [I]t is immaterial to the 'surroundings' in which the value B is used, whether the value B can be found ready-made in the memory, or whether it is necessary to make temporary use of a number of the next stack locations for its evaluation. When a function occurs instead of B and this function is to be evaluated by means of a subroutine, the above [illustration] provides a strong argument for arranging the subroutine in such a way that it operates in the first free places of the stack, in just the same way as a compound term written out in full. [19, p.314]

[9]See Chapter 2 in [49].

To arrange the procedure in such a way that it operates in the first free places of the stack, Dijkstra subsequently, in his paper, explained how *one* run-time stack could do the job[10]. As a *by-product* of Dijkstra's generalization, recursive-procedure activations became feasible:

> The subroutine only has to appear in the memory once, but it may then have more than one simultaneous 'incarnation' from a dynamic point of view: the 'inner-most' activation causes the same piece of text to work in a higher part of the stack. Thus the subroutine has developed into a defining element that can be used completely recursively. [19, p.317]

Dijkstra's generalized stack was able to store parameters and local variables of activated procedures. And, to make these elements accessible, Dijkstra had devised a mechanism to delve deep down in his stack, a mechanism which was not needed for the evaluation of simple arithmetic expressions.

Dijkstra was not the first implementor of recursive activations[11]. The relatively simple recursive mechanisms of the list-processing languages `IPL` and `LISP` had already been implemented by means of a stack [39, p.192-193]. Rutishauser, in his 1963 paper [5, p.50], not only credited Dijkstra as the inventor of a technique to implement recursion for `ALGOL60`, but also the Americans Sattley and Ingerman (cf. [53]), who had worked closely with Floyd, Irons, and Feurzeig (see e.g. [54]). Furthermore, Turing had, by 1945, already thought through the idea of using a stack for recursive activations but had not implemented it[12]. Bauer has confirmed Turing's contributions and has also mentioned Rutishauser[13], Van der Poel, and Huskey as researchers who had implemented recursion prior to 1960 [57, 58, -,p.39].

It is Dijkstra's generalizing style which stands out when a comparison is made with the work of his contemporaries. Rutishauser, for example, had limited the order of his procedure activations in his run-time system [59], while Dijkstra had no such restriction. Floyd's work [60, p.42-43] relied on three specialized "yo-yo" lists (i.e., stacks) instead of one general stack. Likewise, the MAD translator [61, p.28] used several specialized tables. Also, and most importantly, the `ALCOR` compilers were severely restricted in that they could not handle several `ALGOL60` language constructs, including the recursive procedure (cf. Section 3.4). Finally, though the Irons-Feurzeig system [54] *did* implement recursive-procedure activations *and* by means of one general run-time stack, it was, in the interest of efficiency, sophisticated in its run-time capabilities and, hence, unlike the run-time system of Dijkstra and Zonneveld (cf. Section 4.2).

---

[10]Dijkstra emphasized that one universal stack was sufficient to implement `ALGOL60` [63, p.344].

[11]Contrary to what is suggested in [51, p.5] and [52, p.96].

[12]See [55] and [56, p.188, 237].

[13]Rutishauser, himself, mentioned in his 1963 paper [5, p.50] that he had implemented recursive subroutines for the ERMETH in the "pre-ALGOL days".

3.2.2 Further Generalization in Space

To further illustrate Dijkstra's appeal for generalization, I summarize the essential ideas [62] underlying the paper [48] he presented in Rome 1962. In his paper, Dijkstra described, what he called, the condensation of his meditations after having implemented `ALGOL60`. He presented a language of a stack-based machine which was "of a perverse inefficiency" [62, p.1]. For, again, Dijkstra's main objective was to pursue "extreme simplicity and elegance" [62, p.1], by devising a uniform way in which his machine could perform different operations. Some examples from his paper are presented below.

Consider the program

$$5 \quad + \quad 39 \quad / \quad (\quad 7 \quad + \quad 2 \quad * \quad 3 \quad) \quad - \quad 6 \quad ;$$

Its corresponding postfix notation is

$$5 \quad 39 \quad 7 \quad 2 \quad 3 \quad * \quad + \quad / \quad + \quad 6 \quad -$$

which is read from left to right by the stack-based machine in the following manner. When it encounters a number, it is copied to the top of the stack. When it encounters an operator, the corresponding operation is performed at the top of the stack.

Dijkstra remarked that the function of an operator is, thus, a double one. On the one hand, it indicates that copying words to the stack has to be interrupted. On the other hand, it also specifies the operation that has to be performed at the top of the stack. Dijkstra suggested to separate these two functions, by treating arithmetic operators *in the same way* as numbers: the operator should *also* be copied onto the stack, and its evaluation should be performed by a new and separate word E (for Evaluate). In accordance with these new conventions, the postfix notation would then be

$$5 \quad 39 \quad 7 \quad 2 \quad 3 \quad * \quad E \quad + \quad E \quad / \quad E \quad + \quad E \quad 6 \quad - \quad E$$

Whenever the word read is unequal to E, the word is copied onto the stack. Whenever the word read is equal to E, it is not copied; instead, the operation on the top of the stack is performed. As the reader can check, such an execution would finally result in the number 2 being the top element of the stack, as desired.

Further on in his paper, Dijkstra introduced variables which could be placed on the stack as well. For instance, the expression

$$x \quad + \quad 4 \quad ;$$

corresponds to the post-fix notation

$$x \quad E \quad 4 \quad + \quad E$$

Upon execution, x would first be placed on top of the stack. After encountering the first occurrence of E, the variable x would be substituted by the value it denotes, which depends on the state of the process at that moment. For example, the top element of the stack, x, could be substituted by 3, implying

that the final result would be a stack with as top element the value 7. In short, Dijkstra treated variables as operators and, as explained previously, he treated operators as numbers.

Dijkstra's further generalization was led by the observation that a number is only a special kind of expression. That is, while in the previous examples the final addition to the stack was merely a number, it should, more generally, be possible to devise examples in which the final result is a general expression. To achieve this generalization, Dijkstra allowed the word E to be placed on the stack as well by introducing a new word P (for Postponement). Upon evaluation of P, it would be replaced by the word E. As an example, consider the following text with three variables x, y, and *plinus*:

$$x \quad P \quad E \quad y \quad P \quad E \quad plinus \quad E \quad P \quad E$$

Upon execution, the top of the stack would show in succession

$$\begin{array}{lllllll}
\ldots & x & & & & & \\
\ldots & x & P & & & & \\
\ldots & x & E & & & & \\
\ldots & x & E & y & & & \\
\ldots & x & E & y & P & & \\
\ldots & x & E & y & E & & \\
\ldots & x & E & y & E & plinus & \\
\ldots & x & E & y & E & + & \\
\ldots & x & E & y & E & + & P \\
\ldots & x & E & y & E & + & E
\end{array}$$

The last line in the previous illustration contains the string of words which, when read as a piece of program, would effectuate the evaluation of the expression $x + y$. Likewise, if the value of the variable *plinus* had been $-$, then the resulting string of words would have corresponded to the expression $x - y$. In other words, the net effect of the previous illustration is that the expression x *plinus* y has been partially evaluated with as result *another expression*.

Dijkstra continued in his paper by showing that the distinction between numbers and instructions was superfluous as well. His generalizations furthermore led him to introduce a second stack, which he called the stack of activations. In his words:

> We could try to merge our two stacks into one. This merging would present itself in a completely natural fashion if the two should expand and shrink "in phase" with one another. In general, however, this is not the case and trying to merge the two stacks into a single one would give a highly unnatural construction. [48, p.247]

In summary, while Dijkstra had only needed one stack to implement `ALGOL60`, he needed two stacks to implement his more general stack-based programming language. In both cases it was his quest to generalize which stands out.

3.3 Machine-Independent Object Language

Dijkstra's single most important ALGOL60-related contribution, in terms of generalization, is without doubt his design of an intermediate *machine-independent* object language[14]. That is, a language which serves the purpose of describing the behavior of a general stack-based machine and *not* the X1 machine in particular. Dijkstra situated this language in between ALGOL60 and the machine instructions of the X1, and thereby created –what we would today call– a separation of concerns, which heavily simplified the implementation of the Dijkstra-Zonneveld compiler.

The separation of concerns is twofold: a translation stage followed by an interpretation stage. The translation, from ALGOL60 to the object language, is accomplished in a machine-independent manner and, hence, without any appeal to machine efficiency. Candidates for this translation stage are jobs that can be done once and for all; i.e., jobs that are independent of the execution of the program and, hence, intrinsically static. An example is determining which brackets in an ALGOL60 program form pairs [47, p.349]. In the second stage, the obtained object program is processed by an interpreter written in the machine code of the X1. Only at this stage does dynamic physical memory management come into play. An important example is mapping the run-time software stack onto X1's memory [63, p.336].

It is, again, Dijkstra's top-down perspective, from ALGOL60 to the machine, which stands out in comparison to the machine-specific approaches of Bauer, Samelson, and others[15]. In Dijkstra's own words:

> [W]e are completely free to determine how the computer should be used, this in contrast with machines for which considerations of efficiency force us in practice to a special manner of use, that is, force us to take account of specific properties and peculiarities of the machine. [63, p.330]
>
> [T]he making of an ALGOL translator is a relatively simple job if the translator may formulate the object program in operations cut out for the problem. [63, p.344]

To design his object language, Dijkstra had made several abstractions. For instance, he had assumed the presence of a sufficiently large homogeneous store, and had thereby abstracted away from X1's heterogeneous memory; i.e., the presence of a fast, small store and a slow, large store. Likewise, he had assumed X1's arithmetic unit to be extremely fast, thereby allowing him to extensively use subroutines without being concerned with computation time [63, p.330].

Similar to Brown and Carr in 1954 and as mentioned before, Dijkstra was fully aware of the prolonged-computation times introduced by his abstractions.

[14] Today we could view Dijkstra's work as a precursor to the Java Virtual Machine.

[15] See e.g. [15, p.210] and [18, 16]. Also, the account of Randell and Russel [64, p.3] indicates that Dijkstra's machine-independent object language was unconventional at that time. However, once again, I stress that *no* claim is made in this paper that Dijkstra was the *first* person to introduce a machine-independent object language. Other people had come up with similar ideas independently –see e.g. the remarks of Galler and Gallie in [65, p.528] and McCarthy's work on LISP in particular [22].

Moreover, he acknowledged the short-term limitations of his solution:

> We are fully aware of [...] a certain prolongation of the calculating time, and we can imagine that for some computer which is still in use today one cannot accept this delay. There are twofold reasons why we nevertheless made this choice, one of principle and one practical [...] [63, p.330]

3.4 Reception of Dijkstra

Dijkstra and Zonneveld succeeded very quickly in building an `ALGOL60` compiler. Their success did not go unnoticed, as the British researcher Randell recollects:

> [One] week of discussions with Dijkstra were spent [...] These discussions we documented in a lengthy report [64] [...]. For the next few years Dijkstra used our report to defend himself from the numerous further requests he was getting from people who wanted to visit him and find out about the `X1` compiler. [66, p.3]

Dijkstra's generalizing style had clearly influenced subsequent practical developments in compiler technology during the 1960s –as explicitly confirmed by Naur [67, p.105]. By breaking away from the efficiency regime, Dijkstra and Zonneveld had succeeded in building a general and fast `ALGOL60` translator for a relatively small computer, the `X1`.

Contrary to the Amsterdam team, several `ALCOR` members already had functional translation technology available before the recursive procedure entered the `ALGOL60` definition (cf. telephone call between Van Wijngaarden and Naur) [4, 15, p.129, p.210]. In the interest of machine efficiency, `ALCOR` had decided to minimize their run-time system as much as possible by following a strict static approach: each procedure was allocated a fixed working space prior to program execution. This meant that procedures could not be activated more than once during program execution and, hence, that recursive-procedure activations, in particular, were ruled out. At the 1962 Rome symposium, however, Bauer and Samelson expressed their regret in choosing a static solution:

> [I]t was decided, to assign static data storage to each procedure separately within the block containing the procedure, which of course rules out recursive procedures. The waste of static storage, in conflict with our original cellar [i.e., stack] principle, was considered *regrettable*. [15, p.214, my italics]

These words are ironic because `ALCOR` was a strong proponent of practical engineering-based solutions. Some `ALCOR` members, such as Seegmüller, had openly distanced themselves from Dijkstra, Van Wijngaarden, and Naur who did not seem to care much about machine efficiency, but, instead, strove for a general algorithmic language. The irony, thus, lies in the fact that `ALCOR`'s approach had failed, even though they had restricted the use of the programming language, while Dijkstra, for instance, had not.

Naur's team in Copenhagen had initially joined ALCOR and, hence, had followed an efficiency-driven philosophy as well [18, p.118]. And, even after having started a close collaboration with the Amsterdam team in March 1960, the Danes remained reluctant in subscribing to the Amsterdam approach. In Naur's words:

> [In March 1960, t]he Dutch group impressed us greatly by their very general approach. However, although they were prepared to put their solution of the problem of recursive procedures at our disposal we decided to stick to the more modest approach which *we had already developed* to some extent. The reasons for this reluctance were practical. [...A]t the time we feared the loss of running speed of a system which included recursive procedures (a fear we now know was unfounded). [18, p.118-119, my italics]

Hence, the Danes were initially not able to handle recursive procedures. And, in line with ALCOR's doctrine, they had chosen static solutions [16, p.441]. Furthermore, the Danes had explicitly allowed information transfers between core store and drum to be expressed in their ALGOL60 programs [16, p.441]. Such machine-specific programming was in sharp contrast to the Dijkstra-Zonneveld approach and was very similar to the work of Strachey and Wilkes [12, p.491] and Backus's Speedcoding (cf. Section 2).

In their later 1962 GIER compiler, however, the Danes did follow "the Amsterdam school" and, therefore, were able to handle recursive procedures[16]. The ALCOR group would need some more years before, they too, would finally adopt the Amsterdam approach, as exemplified by the the compiler described in [70].

4 The Dichotomy Outlived the ALGOL60 Effort

As a successful ALGOL60 compiler writer, Dijkstra was invited to give a keynote address in Munich (autumn 1962). In his presentation, he made a sharp contrast between reliability and optimization and, unsurprisingly, championed the former. In his words:

> In deciding between reliability of the translation process on the one hand, and the production of an efficient object program on the other hand, the choice often has been decided in favour of the latter. But I have the impression that the pendulum is now swinging backwards. [71, p.537]

Subsequently, Dijkstra mentioned that there are two ways one can use a new and more powerful computer. The classical reaction, he said, is to use the new machine as efficiently as possible. The alternative, however, is to recognize that the new machine is indeed faster and that, hence, time does not matter so much any more. That is, the cost per operation in the new machine is less than in the old machine. Hence, it becomes more realistic to invest some of the machine's

[16] See [18, 68, p.118, 120-124, p.94]. Also Samelson described the Amsterdam and ALCOR approaches as two different "schools of thought" [69, p.490].

speed in other things than sheer production, such as programming comfort, elegance, and reliability [71, p.537].

Dijkstra viewed optimizations, on the one hand, and reliability and trustworthiness, on the other hand, as opposing goals. He described optimizing as "taking advantage of a special situation" (cf. specialization). And, according to him, the construction of an optimizing compiler is "nasty", in comparison to "straightforward but reliable and trustworthy" compiler technology [71, p.538].

Afterwards, Dijkstra described `ALGOL60` as a "great promotor of non-optimizing translators", a remark which I shall return to shortly, when discussing the Irons-Feurzeig system. Dijkstra continued by contrasting the Dijkstra-Zonneveld compiler to the many other compilers which were based on language restrictions:

> The fact is that the language, as it stands, is certainly not an open invitation for optimization efforts. For those who thought that they knew how to write optimizing translators –be it for less flexible languages– this has been one of the reasons for rejecting `ALGOL60` as a serious tool. In my opinion these people bet on the wrong horse. [71, p.538]

The last sentence misleadingly suggests that the dichotomy between specialization and generalization was equally obvious to other researchers, as it presumably was to Dijkstra himself in 1959 and 1960. On the other hand, however, the dichotomy did indeed become more apparent *after* 1960, as is illustrated by the comments of Galler [72, p.525], Gallie [73], and Randell's 1964 recollections[17].

4.1 Twin Approach

While the Dijkstra-Zonneveld compiler was closely mimicked by Randell and his colleagues in Leicester, it is equally important to note that the British did not follow Dijkstra all the way. For, in later years, the Leicester team had two fully functional `ALGOL60` compilers at their disposal: the Whetstone compiler and the Kidsgrove compiler. While the former closely resembled the Dijkstra-Zonneveld compiler and was, therefore, fast in translating `ALGOL60` programs but poor in generating fast programs, the latter compiler showed little resemblance with the Dijkstra-Zonneveld compiler and was a slow compiler that produced very fast programs. The practical approach taken by the Leicester team was twofold. A new `ALGOL60` program was first quickly compiled by the Whetstone compiler and tested. And, after the programmer became convinced of the correctness of his program, he would subsequently use the Kidsgrove compiler to re-compile his `ALGOL60` program in order to obtain fast machine code. The Leicester team had thus come up with a "twin approach" in which Dijkstra's philosophy was

[17] Cf. [74, p.2]. See also Samelson's 1978 recollections in which he contrasts between "liberalists" such as Naur and "restrictionists" such as himself, and states that: "[U]nification burdens the normal case [in terms of machine efficiency] with the problems arising from the exceptional case" [4, p.132-133]. But, as Naur correctly points out [4, p.136], this partitioning of the `ALGOL60` community into two opposing groups only took place in the years following the publication of the `ALGOL60` report. See, in particular, the opposing views expressed at the Rome 1962 conference.

embodied in one compiler and the efficiency regime was embodied in the other compiler[18]. Hence, in a very practical sense, the dichotomy outlived the ALGOL60 effort, thereby partly contradicting Dijkstra's key-note address in Munich 1962.

4.2 Run-Time Optimizations

Though Dijkstra viewed ALGOL60 as a "great promotor of non-optimizing translators", it is interesting to note that, in the same year in which Dijkstra's paper on recursion [19] was published, Irons and Feurzeig had also come up with an implementation technique for recursive procedures [54]. Unlike Dijkstra's approach, however, the Irons-Feurzeig system applied *run-time optimizations*.

By means of some additional book-keeping, the Irons-Feurzeig system detected, at run time, whether the procedure under investigation was involved in recursion or not. That is, the system *specialized at run time* by distinguishing between recursive- and nonrecursive-procedure activations, and treating each kind of activation separately. As a result, only the truly recursive procedures were slowed down and taxed in terms of storage allocation; while in the Dijkstra-Zonneveld compiler, *all* procedures were treated conservatively[19].

Irons and Feurzeig's run-time specialization was implemented by using several case distinctions[20]. Hence, from Dijkstra's point of view, the Irons-Feurzeig system was sophisticated. From an efficiency-driven perspective, however, the system was outstanding in that it not only provided fast ALGOL60 programs, but it also did so *without* having to restrict the ALGOL60 language.

In retrospect, then, the dichotomy between specialization and generalization does not hold in the case of the Irons-Feurzeig system. For, that system applied run-time specializations *and* handled general language constructs, such as the recursive procedure. The essence of the system is that optimizations were performed at run time, rather than enforcing language restrictions in adherence with the ALCOR doctrine. Nevertheless, part of Dijkstra's key-note address in Munich 1962 remains valid in that the run-time efficiency pursued by Irons and Feurzeig still opposed the simplicity Dijkstra wanted in the interest of correctness and reliability.

[18] Cf. [11, p.34]. Also Rutishauser, in 1962, had expressed the desire for such an approach [65, p.527].

[19] In the Dijkstra-Zonneveld implementation, no detection was made to check whether an activated procedure P was recursive or not. Therefore, whenever P was to call another procedure Q, the conservative assumption had to be made that P was recursive and, hence, would be called again later. As a result, all the local variables and arrays of P had to be placed on the software stack prior to calling Q such that they could retain a unique identity for the special recursive case that the procedure were to be called again. In the Irons-Feurzeig system, such expensive stack management was only carried out *after* it was determined that P would be called more than once.

[20] For instance, Irons and Feurzeig used several specialized thunks, explained in [75, p.57].

5 Final Remarks

Several researchers have written about the history of computing (e.g., [76, 77, 78, 56, 79, 80]) and, in particular, about the history of programming languages (e.g., [81, 43, 82, 29, 20, 83]). Also, an increasing number of students are interested in past developments (e.g., [52, 84, 35, 49]). Nevertheless, in accordance with Mahoney's words, I believe more research is needed:

> [Historians] remain largely ignorant about the origins and development of the dynamic processes running on [computers], the processes that determine what we do with computers and how we think about what we do. The histories of computing will involve many aspects, but primarily they will be histories of software. [79, p.127]

The recursive procedure is an example par excellence of such a dynamic process. To the best of my knowledge, it has not been treated technically in any previous historical account. Nor has the history of software been written by contrasting Dijkstra's views with those of his contemporaries, as I have attempted in this paper.

Future Work

Research contributions of Gödel, Carnap, Turing, and Tarski have been studied and documented over and over again by logicians and philosophers themselves. Computer scientists, by contrast, have yet to commence with similar work concerning the ideas of their fathers: Dijkstra, McCarthy, Hoare, and others. This, in turn, explains my motivation to write this paper.

More historical accounts, written by other researchers, are however needed in order to obtain a more objective understanding of Dijkstra's contributions. Based on an earlier draft of this paper, I have already received the comment that Dijkstra's ideas on programming-language design, elaborated on in Section 3.1, were only meant for theoretical purposes and that I have thus misinterpreted Dijkstra to some extent by connecting his abstract thoughts with his practical work. A similar remark has been made concerning Dijkstra's abstract machine, explained in Section 3.2.2.

Based on my understanding of Dijkstra as a man who did not want to distinguish between theory and practice, as a man who practiced what he preached, I believe the two aforementioned remarks are ill-founded. The remarks remind me of Seegmüller's and Strachey's criticism to Dijkstra, claiming that the latter's work was purely theoretical. Not so! Dijkstra's abstract ideas clearly sidle through his applications. For example, Dijkstra's temporary abstraction of efficiency, while discussing the quality of a programming language [13, 14, p.10, p.35], is directly reflected in the two-stage design of the Dijkstra-Zonneveld compiler. Likewise, his thoughts on abstract-machine languages is clearly related to the stack-based object language which he introduced to simplify the translation of `ALGOL60`. Finally, Dijkstra used one universal stack to implement `ALGOL60` while many of his contemporaries used multiple specialized stacks, and he used

two stacks in his later work presented in Rome 1962. Is it a coincidence that most 1st generation stack machines, which were designed to execute `ALGOL`-like languages, only had a single stack [49, p.36], and that 2nd generation stack computers separated evaluation and control flow by means of two stacks [49, p.10]? Though more research is needed concerning these two questions, I would not be surprised if Dijkstra's abstract thoughts, again, had a direct bearing on such practical matters.

Several other questions concerning Dijkstra's career have yet to be investigated. To give one detailed example, and as suggested recently in [85], Dijkstra may have become acquainted with Turing's work on 'Reversion Storage' (i.e., Turing's stack principle) via Huskey who had visited Van Wijngaarden and Dijkstra in Amsterdam in the summer of 1959. In fact, at the end of his 1960 paper [19], Dijkstra explicitly thanked Huskey for the *inspiring* conversations that he had had with him in Amsterdam. Twenty years later, however, Dijkstra's recollections were quite the opposite:

> Harry D. Huskey had just spent a few sabbatical months at the Mathematical Center, working on an algebraic compiler, but his style of work differed so radically from mine that, personally, I could not even use his work as a source of inspiration; the somewhat painful discussion with my boss [i.e., Van Wijngaarden], when I had to transmit to him that disappointing message, is remembered as one of the rare occasions at which I banged with my fist on the table. [46, p.572]

More research is thus needed to understand how Huskey did or did not influence Dijkstra in implementing `ALGOL60`. But it is important to stress that, during the 1950s, the USA was technologically advanced compared to the Netherlands (cf. Section 2). The American Huskey may well have played an important role in terms of technology transfer by visiting Amsterdam in 1959.

On the one hand, Huskey's style seems to have been very much at odds with Dijkstra's quest to generalize, as follows from Huskey's negative comments on general language constructs in [2, p.379-380] and his various specialized lists in [86, 87]. In retrospect then, the previous quote may not be so surprising after all[21].

On the other hand, however, Keese and Huskey's 1962 compiler technique [89] is very similar to that of Dijkstra and Zonneveld. It has an intermediate machine-independent object language, and dynamic storage assignment and recursive procedures are processed by the assembler. Had Huskey learned these techniques from Dijkstra or vice versa? As the historian Mahoney has noted, "even when we can't know the answers, it is important to see the questions. They too form part of our understanding. If you cannot answer them now, you can alert future historians to them" [90, p.832].

Finally, it should be remarked that, due to focusing on Dijkstra and `ALGOL60`, several important researchers have not been mentioned in this paper. Examples

[21] In this respect, see also Knuth's comments on Dijkstra's "strong sense of aesthetics" and that Dijkstra "didn't want to compromise his notions of beauty" [88, p.41].

are Zuse, Hopper, Laning, and Zierler. Likewise, concerning nationalities, also French (e.g. Vauquois), Norwegian (e.g. Garwick) and British (e.g. Woodger) researchers actively participated in defining ALGOL60. And, other important topics, such as syntax-directed compilation, have only been mentioned in passing and deserve more attention in future work.

Conclusions

Two messages lie at the heart of this paper: (i) the early history of programming languages, and the ALGOL60 effort in particular, can be perceived as a dichotomy between specialization and generalization, and (ii) Dijkstra's continual appeal for generalization led to practical breakthroughs in compiler technology. Specialization, as promoted by Bauer, Samelson, Strachey, and Wilkes, refers to language restrictions, static solutions, and the exploitation of machine-specific facilities —in the interest of efficiency. Generalization, as promoted by Van Wijngaarden and Dijkstra, refers to general language constructs, dynamic solutions, and machine-independent language design —in the interest of correctness and reliability.

In Western Europe, Van Wijngaarden and Dijkstra seem to have been rather the exception than the rule in reasoning linguistically. Dijkstra in 1961 devoted an entire report [13] in countering the language restrictions that Strachey, Wilkes, and others had proposed in the interest of machine efficiency. Here the battle between generalists and specialists is vividly illustrated in Dijkstra's own words. Likewise, the panel discussion [2] at the Rome 1962 conference shows the tensions underlying the dichotomy.

Dijkstra's simple solutions, due to his generalizing style, stand out in comparison to the work of his contemporaries and it is this what made the Dijkstra-Zonneveld compiler shine. Furthermore, it was his quest to unify which led him, together with Van Wijngaarden, to promote the recursive procedure in the first place, and to find a corresponding implementation technique. For, just like many of his contemporaries, he did not see any real necessity in using the recursive procedure to solve practical programming problems. In his own words in 1962:

> One of our great features of our compiler is that it happens to turn out that it is very easy to have a good recursive function in it. I am very fond of them. They are hardly used by customers. Nevertheless, it is very important that they are in. The reason is that they give us possibilities that make the tool inspiring. [2, p.368]
>
> In practical computations these [general] features are not too frequently used, but the bare fact that the programmers could use them if they wanted to made the language very appealing. [91, p.127]

Though Dijkstra had succeeded, together with Zonneveld, in implementing ALGOL60 with recursive procedures, many participants at the Rome 1962 symposium remained very sceptical about Dijkstra's emphasis on general language constructs and corresponding dynamic implementation techniques. For, they

had, as many observed, a negative effect on computation time. Not surprisingly then, Dijkstra was perceived as someone who totally neglected efficiency issues, while most people at the symposium considered efficiency to be of prime importance. It seems, however, that Dijkstra was anticipating some of the enormous advances that would soon follow in electronic technology.

Contrary to the Amsterdam team, the `ALCOR` group already had a lot of technical experience with `ALGOL58` when the international community turned towards defining the `ALGOL60` language in 1959 and 1960. Led by an efficiency-driven philosophy, they eschewed the recursive procedure and other dynamic language constructs. Van Wijngaarden and Dijkstra, on the other hand, were in search of a general language. They were led more by language aesthetics than by practical limitations of actual computing machines. And, again, only as a result of such a philosophy, did they become proponents of the recursive procedure.

Dijkstra's appeal for generalization resulted in practical breakthroughs in compiler technology, with British, Danes, West-Germans, and others copying parts of his unifying work and, in particular, his run-time system and top-down compiler-design approach. Such immense success and recognition led Dijkstra to present his rather bold views on compiler technology in his key-note address in Munich 1962. Clearly aware of a dichotomy between specialization and generalization, he championed the latter and viewed `ALGOL60` as a "great promotor of non-optimizing translators". However, as the work of Randell, Irons, Feurzeig, and others illustrate, Dijkstra's generalizing style, while indeed influential, did not override the efficiency regime. In fact, in a very practical sense, the dichotomy outlived the `ALGOL60` effort.

Acknowledgements

Many thanks go to Gerard Alberts, David Nofre, Raphael Poss, and the anonymous referees for discussing previous versions of this paper.

References

[1] Lee, J.A.N. (1996) "Those Who Forget the Lessons of History Are Doomed To Repeat It" or, Why I Study the History of Computing. IEEE Annals of the History of Computing, 18:2, 54-62.

[2] (1962) Proc. of the International Symposium of Symbolic Languages in Data Processing. Rome, 26-31 March, pp. 363-381. Gordon and Breach Science Publishers, New York and London.

[3] Bauer, F.L. and Samelson, K. (1959) The problem of a common language, especially for scientific numeral work (motives, restrictions, aims and results of the Zurich Conference on ALGOL). IFIP, Paris, 120-125. UNESCO, Paris.

[4] Naur, P. (1981) The European side of the last phase of the development of ALGOL60. Wexelblat, R.L. (ed), History of Programming Languages. Academic Press, New York.

[5] Rutishauser, H. (1963) The Use of Recursive Procedures in ALGOL60. Goodman, R. (ed), Annual Review in Automatic Programming 3. Pergamon Press, New York.

[6] Backus, J.W. et al. (1960) Report on the algorithmic language ALGOL60. Naur, P. (ed). CACM 3:5, 299-314.

[7] Hoare, C.A.R. (1961) Algorithm 64: Quicksort. CACM, 4:7, 321.

[8] Hoare, C.A.R. (1981) The Emperor's Old Clothes. CACM, 24:2, 143-161.

[9] Grau, A.A. (1961) Recursive processes and ALGOL translation. CACM, 4:1, 10-15.

[10] Irons, E.T. (1961) A syntax directed compiler for ALGOL60. CACM, 4:1, 51-55.

[11] Randell, B. and Russell, L.J. (1964) ALGOL60 Implementation: The Translation and Use of ALGOL60 Programs on a Computer. Academic Press, London.

[12] Strachey, C. and Wilkes, M.V. (1961) Some proposals for improving the efficiency of ALGOL60. (University Mathematical Laboratory Technical Memorandum No. 61/5.) Also in CACM, 4:11, 488-491.

[13] Report MR 34. Dijkstra, E.W. (1961) On the Design of Machine Independent Programming Languages. Mathematisch Centrum, Amsterdam.

[14] Dijkstra, E.W. (1963) On the Design of Machine Independent Programming Languages. Goodman, R. (ed), Annual Review in Automatic Programming 3. Pergamon Press, New York.

[15] Samelson, K. and Bauer, F. (1962) The ALCOR project. Symbolic languages in data processing, Rome, March, 207-218. Gordon and Breach Science Publishers, New York and London.

[16] Jensen, J. Mondrup, P. and Naur, P. (1961) A Storage Alllocation Scheme for ALGOL60. CACM, 441-445.

[17] Bachelor, G.A. and Knuth, D.E. et al. (eds) (1961) SMALGOL-61. CACM, 4:11, 499-502.

[18] Naur, P. (1963) Compiler Construction and Data Processing. BIT, 3:2, 124-140 and 3:3, 145-166. Also available in Naur, P. (1992) Computing: A Human Activity. ACM Press, Addison-Wesley Publishing Company, New York.

[19] Dijkstra, E.W. (1960) Recursive Programming. Num. Mathematik, 2, 312-318.

[20] Rosen, S. (ed) (1967) Programming Systems and Languages. McGraw Hill, New York.

[21] McGowan, C.L. (1972) The "most-recent"-error: its causes and correction. Proc. ACM Conf. on Proving assertions about programs, SIGPLAN Notices, 7:1, 191-202.

[22] Stoyan, H. (1984) Early LISP History (1956–1959). LFP'84 Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, Austin, 5-8 August, 299-310. ACM.

[23] Alberts, G. and de Beer, H.T. (2008) De AERA. Gedroomde machines en de praktijk van het rekenwerk aan het Mathematisch Centrum te Amsterdam. Studium, 2, 101-127.

[24] (2008) School of Mathematics and Statistics University of St. Andrews, Scotland. JOC/EFR Copyright, *http://www-history.mcs.st-andrews.ac.uk/Biographies/Rutishauser.html*

[25] (1954) Symposium on Automatic Programming for Digital Computers, Washington D.C., 13-14 May. Office of Naval Research, Department of the Navy.

[26] Gorn, S. (1954) Planning Universal Semi-Automatic Coding. Symposium on Automatic Programming for Digital Computers, Washington, D.C., 13-14 May, pp. 74-83. Office of Naval Research, Department of the Navy.

[27] Brown, J., Carr III, J. (1954) Automatic Programming and its Development on the MIDAC. Symposium on Automatic Programming for Digital Computers, Washington, D.C., 13-14 May, pp. 84-97. Office of Naval Research, Department of the Navy.

[28] Backus, J.W., Herrick, H. (1954) IBM 701 Speedcoding and Other Automatic-Programming Systems. Symposium on Automatic Programming for Digital Computers, Washington, D.C., 13-14 May, pp. 106-113. Navy Advisory Math. Panel, Office of Naval Research, Department of the Navy.

[29] Sammet, J.E. (1969) Programming Languages: History and Fundamentals. Prentice Hall.

[30] Backus, J.W. (1959) The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM Conference. IFIP, Paris, 120-125. UNESCO, Paris.

[31] Backus, J.W. (1980) Programming in America in the 1950s – Some Personal Impressions. In Metropolis, N., Howlett, J. and Rota, G-C. (eds), A History of Computing in the twentieth century. Academic Press, Orlando.

[32] Knuth, D.E. (1964) Backus Normal Form vs. Backus Naur Form. Letters to the Editor of the CACM, 7:12, 735-736.

[33] Perlis, A.J. and Samelson, K. (1958) Preliminary Report: International Algebraic Language. CACM, 1:12.

[34] (1954) IBM, Programming Research Group, 'Preliminary Report – Specifications for the IBM Mathematical FORmula TRANslating System `FORTRAN`'. Technical Report IBM, New York.

[35] de Beer, H.T. (2006) The History of the `ALGOL` Effort. Masters Thesis, Tech. Univ. Eindhoven, Department of Mathematics and Computer Science. *http://heerdebeer.org/ALGOL*

[36] Newell, A. and Shaw, J.C. (1957) Programming the logic theory machine. Proc. Western Joint Computer Conf., Los Angeles, 26-28 February, pp. 230-240. ACM, New York.

[37] Newell, A. Shaw, J.C. and Simon, H.A. (1957) Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristic. Proc. Western Joint Computer Conf., Los Angeles, 26-28 February, pp. 218-230. ACM, New York.

[38] Shasha, D. and Lazere, C. (1998) Out of their minds: The Lives and Discoveries of 15 Great Computer Scientists. Copernicus, Springer-Verlag, New York.

[39] McCarthy, J. (1981) History of `LISP`. And the transcripts of: presentation, discussant's remark, question and answer session. In Wexelblat, R.L. (ed), History of Programming Languages. Academic Press, New York.

[40] Aspray, W. (1986) International Diffusion of Computer Technology, 1945–1955. IEEE Annals of the History of Computing, 8, 4.

[41] Nofre, D. (2010) Unraveling Algol: US, Europe, and the Creation of a Programming Language. IEEE Annals of the History of Computing, 32:2, 58-68.

[42] Perlis, A.J. (1981) The American side of the last phase of the development of `ALGOL`. In Wexelblat, R.L. (ed), History of Programming Languages. Academic Press, New York.

[43] Knuth, D.E. (1962) A History of Writing Compilers. Computers and Automation, 11:12, 8-18. Reprinted in: Knuth, D.E. (2003) Selected Papers on Computer Languages. Center for the Study of Languages and Information, Leland Stanford Junior University.

[44] Slater, R. (1989) Portraits in Silicon. MIT Press, Boston.

[45] McCarthy, J. (1959) On Conditional Expressions and Recursive Functions (Letter). CACM, 2:8, 2-3.

[46] Dijkstra, E.W. (1980) A Programmer's Early Memories. In Metropolis, N., Howlett, J. and Rota, G-C. (eds), A History of Computing in the twentieth century. Academic Press, New York.

[47] Dijkstra, E.W. (1963) Making a Translator for `ALGOL60`. In Goodman, R. (ed), Annual Review in Automatic Programming 3. Pergamon Press, New York.

[48] Dijkstra, E.W. (1962) Unifying Concepts of Serial Program Execution. Proceedings of the Symposium Symbolic Languages in Data Processing, Rome, 26-31 March, pp. 236-251. Gordon and Breach Science Publishers, New York and London.

[49] LaForest, C.E. (2007) Second-Generation Stack Computer Architecture. Thesis for Bachelor of Independent Studies, University of Waterloo, Canada.

[50] Bauer, F.L., Samelson, K. (1959) Sequentielle Formelübersetzung. Elektronische Rechenanlagen, 1, 176-182. Also published in English as: (1960) Sequential formula translation. CACM, 3, 76-83.

[51] In Memoriam: Edsger W. Dijkstra (1930–2002). Available from: *http://userweb.cs.utexas.edu/users/EWD/MemRes(A4).pdf*

[52] van den Hove, G. (2009) Edsger Wybe Dijkstra: First Years in the Computing Science (1951-1968). Masters thesis, University of Namur.

[53] (1960) The Allocation of Storage for Arrays in `ALGOL60`. Internal progress report, Office of Computer Research and Education, University of Pennsylvania. Also in: Sattley, K. and Ingerman, P.Z. (1961) The Allocation of Storage for Arrays in `ALGOL60`. CACM, 4:1, 60-65.

[54] Irons, E.T. and Feurzeig, W. (1960) Comments on the Implementation of Recursive Procedures and Blocks in `ALGOL60`. `ALGOL` Bull. Sup, 13.2, 1-15.

[55] Carpenter, B.E. and Doran, R.W. (1977) The Other Turing Machine. Computer Journal, 20, 269-279.

[56] Davis, M. (2000) Engines of Logic: Mathematicians and the origins of the Computer. W.W. Norton & Company, New York.

[57] Bauer, F.L. (2002) My years with Rutishauser. LATSIS Symposium, ETH Zürich.

[58] Bauer, F.L. (2002) From the Stack Principle to `ALGOL`. In Broy, M. and Denert, E. (eds), Software pioneers: contributions to software engineering. Springer, Berlin.

[59] Waldburger, H. (1960) Gebrauchsanleitung für die ERMETH, of the Institut für Angewandte Mathematik der ETH, Zürich.

[60] Floyd, R.W. (1961) An Algorithm for Coding Efficient Arithmetic Operations. CACM, 4:1, 42-51.

[61] Arden, B.W., Galler, B.A. and Graham, R.M. (1961) The Internal Organization of the MAD Translator. CACM, 4:1, 28-31.

[62] Dijkstra, E.W. (1962) `EWD`28: Substitution Processes (Preliminary Publication). I.e., a preliminary draft of [48].

[63] Dijkstra, E.W. (1963) An `ALGOL60` Translator for the X1, in: Goodman, R. (ed.), Annual Review in Automatic Programming 3. Pergamon Press, New York.

[64] Randell, B. and Russell, L.J. (1962) Discussions on `ALGOL` Translation, at Mathematisch Centrum, W/AT 841, Atomic Power Division, English Electric Co., –a record of discussions with Dr. E. W. Dijkstra, at Mathematisch Centrum, Amsterdam, during 4–8 Dec., 1961. *http://www.cs.ncl.ac.uk/publications/trnn/papers/34.pdf*

[65] (1962) Panel on Techniques for Processor Construction. Information Processing 1962 –Proceedings of IFIP Congress 1962, Rome, March, pp. 524-531. Gordon and Breach Science Publishers, New York and London.

[66] No. CS-TR-1190, Randell, B. (2010) Reminiscences of Whetstone `ALGOL`. Technical Report Series, Newcastle University.

[67] Naur, P. (1981) Aad van Wijngaarden's Contribution to `ALGOL60`. In de Bakker and van Vliet (eds), Algorithmic Languages. Amsterdam: North-Holland. See also: Naur, P. (1992) Aad van Wijngaarden's Contribution to `ALGOL60`. In Wegner, P. (ed), Computing: A Human Activity. ACM Press, Addison-Wesley Pub. Company.

[68] Naur, P. (1968) Successes and Failures of the Algol Effort. `ALGOL`-Bulletin, 28, 58-62. See also: Naur, P. (1992) Successes and Failures of the Algol Effort. In Wegner, P. (ed), Computing: A Human Activity. ACM Press, Addison-Wesley Pub. Company.

[69] Samelson, K. (1962) Programming Languages and their Processing. Information Processing 1962 –Proceedings of IFIP Congress 1962, Munich, 27 August – 1 September, pp. 487-492. North-Holland.

[70] Gries, D. Paul, M. and Wiehle, H.R. (1965) Some Techniques Used in the ALCOR ILLINOIS 7090. CACM, 10:12, 804-808.

[71] Dijkstra, E.W. (1962) Some Meditations on Advanced Programming. Information Processing 1962 –Proceedings of IFIP Congress 1962, Munich, 27 August – 1 September, pp. 535-538. North-Holland.

[72] Galler, A. (1962) Remarks on Compiler Construction, p. 525 in Panel on Techniques for Processor Construction, Information Processing 1962 –Proceedings of IFIP Congress 1962, Munich, 27 August – 1 September, pp. 524-531. North-Holland.

[73] Gallie, T. (1962) Techniques for Processor Construction, pp. 526-527 in Panel on Techniques for Processor Construction, Information Processing 1962 –Proceedings of IFIP Congress 1962, Munich, 27 August – 1 September, pp. 524-531. North-Holland.

[74] Randell, B. (1964) Whetstone `ALGOL` Revisited, or Confessions of a Compiler Writer. APIC BulletinIssue 21, Brighton, 20 May, Automatic Programming Information Centre, College of Technology, Brighton.

[75] Ingerman, P.Z. (1961) Thunks: a way of compiling procedure statements with some comments on procedure declarations. CACM, 4:1, 55-58.

[76] Campbell-Kelly, M., Aspray, W. (1996) Computer: A History of the Information Machine. Basic Books, New York.

[77] Campbell-Kelly, M. (2003) From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry. MIT Press, Boston.

[78] Ceruzzi, P.E. (2003) A History of Modern Computing. MIT Press, Boston.

[79] Mahoney, M.S. (2005) The histories of computing(s). Interdisciplinary Science Reviews, 30:2, 119-135.

[80] Metropolis, N. Howlett, J. and Rota, G-C. (1980) A History of Computing in the Twentieth Century. Academic Press, New York.

[81] Knuth, D.E. (1977) The Early Development of Programming Languages. Encyclopedia of Computer Science and Technology, 7, 419-493. See also: Knuth, D.E. (2003) The Early Development of Programming Languages. In Knuth, D.E. (eds), Selected Papers on Computer Languages. Center for the Study of Languages and Information Leland Stanford Junior University.

[82] Priestley, P.M. (2008) Logic and the Development of Programming Languages, 1930-1975. University College London, PhD thesis.

[83] Wegner, P. (1976) Programming Languages – The First 25 Years. IEEE Transactions on Computers, 25:12, 1207-1225.

[84] Van Oudheusden, K. –alias Daylight, E.G. (2009) The Advent of Recursion & Logic in Computer Science. Master's Thesis in Logic, Univ. of Amsterdam, *http://www.illc.uva.nl/Publications/ResearchReports/MoL-2009-12.text.pdf*

[85] Henriksson, S. (2009) A brief history of the stack. SHOT, Pittsburgh, October. SIGCIS.

[86] Huskey, H.D. and Wattenburg, W.H. (1961) A Basic Compiler for Arithmetic Expressions. CACM 4:1, 3-9.

[87] Huskey, H.D. (1961) NELIAC—A Dialect of ALGOL. CACM, 3:11, 463-468.

[88] CHM Reference number: X3926.2007. Oral History of Donald Knuth – Interviewed by E. Feigenbaum in 2007. Mountain View, California, Computer History Museum.

[89] Keese, W.M. and Huskey, H.D. (1962) An Algorithm for the Translation of Algol Statements. Information Processing 1962 –Proceedings of IFIP Congress 1962, Munich, 27 August – 1 September, pp. 498-502. North-Holland.

[90] Mahoney, M.S. (1996) What Makes History?. Appendix A in Bergin, T.J. Jr., and Gibson, R.B. Jr. (eds), History of Programming Languages. ACM Press, New York.

[91] Report MR 47, Dijkstra, E.W. (1962) Operating Experience with ALGOL60. Mathematisch Centrum Amsterdam. See also: Dijkstra, E.W. (1962) Operating Experience with ALGOL60. Computer Journal, 5:2, 125-127.