# The Advent of Recursion & Logic in Computer Science

**MSc Thesis** (*Afstudeerscriptie*)

written by

**Karel Van Oudheusden**
**–alias Edgar G. Daylight**
(born October 21st, 1977 in Antwerpen, Belgium)

under the supervision of **Dr Gerard Alberts**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

**MSc in Logic**

at the *Universiteit van Amsterdam.*

**Date of the public defense:**
*November 17, 2009*

**Members of the Thesis Committee:**
Dr Gerard Alberts
Prof Dr Krzysztof Apt
Prof Dr Dick de Jongh
Prof Dr Benedikt Löwe
Dr Elizabeth de Mol
Dr Leen Torenvliet

Institute for Logic, Language and Computation

*"We 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."*

J.A.N. Lee in 1996 [73, p.54]

*"To many of our colleagues, history is only the study of an irrelevant past, with no redeeming modern value –a subject without useful scholarship."*

J.A.N. Lee [73, p.55]

"*[E]ven 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.*"

M.S. Mahoney [76, p.832]

*"Only do what only you can do."*

E.W. Dijkstra [103, p.9]

# Abstract

The history of computer science can be viewed from a number of disciplinary perspectives, ranging from electrical engineering to linguistics. As stressed by the historian Michael Mahoney, different 'communities of computing' had their own views towards what could be accomplished with a programmable computing machine. The mathematical logicians, for instance, had established what programmable computing machines with unbounded resources could not do, while the switching theorists had showed how to analyze and synthesize circuits. "But no science accounted for what finite machines with finite, random access memories could do or how they did it. That science had to be created." –Mahoney [78, p.6].

With the advent of the programmable computing machine, new communities were created, such as the community of numerical analysts. Unlike the logicians and the electrical engineers, the numerical analysts, by their very profession, took programming seriously. Several of them gradually became more involved in seeking specific techniques to overcome the tediousness in programming their machines. One such, and important, technique was the recursive procedure. While logicians had been well-acquainted with the concept of recursion for quite some time, and the development of mathematical logic had, itself, contributed to the advent of the programmable computing machine, it is unclear whether the idea of the recursive procedure entered the arena of programming languages via the logic community. More generally, it is unclear how and to what extent, exactly, ideas from logic have influenced the computer pioneers of the 1950-60s.

Both unclarities, described above, are addressed in this thesis. Concerning the first unclarity, the recursive procedure entered the arena of programming languages in several ways by different people. Special attention will be paid to the pioneer Edsger W. Dijkstra who, in 1960, received world-wide recognition for implementing recursive procedures for the `ALGOL60` programming language, i.e. by building a compiler. While recursive procedures remained highly controversial during the 1960s, Dijkstra was one of its few strong advocates. His views, led by linguistic ideals, were in sharp contrast to those that were led by specific machine features. With respect to the second unclarity, it will be shown that several ideas from logic that did influence some computer pioneers, were primarily received indirectly and without full comprehension. In fact, these pioneers, in the aftermath of their successes, openly stressed that they were not logicians and had not completely understood all of the logic underlying their sources of inspiration. Similarly, the logicians, themselves, did not initially grasp the connection between Turing's 1936 paper and the programmable computing machine either. Finally, emphasis will be laid on Dijkstra's ability, in later years, to connect the unsolvability of Turing's Halting Problem with the practical engineering problems that his community faced.

## Disclaimer

An historical-accurate narrative, as attempted here, often implies mathematical inaccuracy with respect to the current state of the art. In this thesis, the recursive procedure is 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.

# Contents

# Chapter 1

# Introduction

Rome, March, 1962: It was a sunny day and the streets were packed with tourists, many of whom were American. The occasion was Jacqueline Kennedy's visit to Pope John XXIII at the Vatican, where she received a beautiful gold casket. Among those few tourists who were fortunate enough to get a glimpse of Jacqueline Kennedy and the Pope were the two Americans Sarah Ingerman and Monica Weizenbaum.

Less fortunate, according to Sarah and Monica, were their husbands. Only a few blocks away, but inside the "Palazzo dei Congressi", Peter Ingerman and Joseph Weizenbaum were attending the International Symposium of Symbolic Languages in Data Processing. In one of Rome's largest conference rooms, filled solely with men, Peter and Joseph sat together listening to the famous 31-year old Dutch man, Edsger W. Dijkstra. To some extent irritated by Dijkstra's arrogance, but equally impressed by his remarks, Peter and Joseph were trying to make up their minds whether Dijkstra was right. Should a machine-independent programming language be able to express recursive procedures, as Dijkstra was advocating for? Or, was the majority of the attendees correct in claiming that recursive procedures were too inefficient to execute on a machine and, generally, of little or no practical value?

### The Ideal Narrative

In my opinion, the previous two paragraphs represent, with perhaps a slight exaggeration, the ideal historical narrative for my thesis. Unfortunately, a lot of it is made up. While Jacqueline Kennedy did[1] visit the Vatican in March 1962, I don't know whether it was a sunny day and whether Sarah Ingerman and Monica Weizenbaum actually existed and, if they did, whether they were there that day. In fact, the Symposium was held several days later. What is true is that Ingerman, Weizenbaum, and Dijkstra attended the International Symposium of Symbolic Languages in Data Processing, held in Rome at the

[1] Jacqueline Kennedy: The White House Years –Selections from the John F. Kennedy Presidential Library and Museum (Columbia Point, Boston).

"Palazzo dei Congressi" between March 26-31. Also correct is that Dijkstra was already famous, and, according to some, arrogant. Finally, and most importantly, the recursive procedure was indeed an important topic of debate during that symposium, and Dijkstra was one of its strong advocates[2].

**The Advent of Recursion (Chapter 2)**

The recursive procedure, which Dijkstra openly advocated for in Rome 1962, had already been introduced in 1960 in the *definition* of the programming language `ALGOL60`, albeit in a peculiar way. Subsequently, but still in 1960, Dijkstra and his colleague Zonneveld succeeded in being internationally the first to *implement* `ALGOL60`, i.e. by building the first `ALGOL60` compiler. Both the definition of `ALGOL60` and its implementation, with respect to the recursive procedure, constitute the *first theme* of my thesis, addressed in greater detail in Chapter 2. The reason to do so lies in my quest to understand how the general concept of recursion entered the arena of programming languages and, hence, also computer science.

In the previous paragraph, I have used the terms 'recursive procedure' and 'recursion'. The former, to be illustrated later, is an example of the latter. In general terms, recursion is *the act of returning*.

In the context of functions, recursion can be described more precisely as *the process for defining new functions from old at work* [12, p.67]. For instance, let $h$ be a function of two arguments and $g$ a function of three arguments. Then, in accordance with:

$$h(x, z) = g(x, y, h(x, y))$$

$h$ is defined by recursion on the basis of $g$ and the symbols in the previous line denote a recursive definition of the function $h$.

To illustrate another example of a recursive definition, it is instructive to consider the classical mathematical definition of a continued fraction (1) and to compare it with the definition in (2):

1. A continued fraction is a fraction whose numerator is an integer and whose denominator is an integer plus a fraction, whose numerator is an integer and whose denominator is an integer plus a fraction and so on ...

2. A continued fraction is a fraction whose numerator is an integer and whose denominator is an integer plus a continued fraction.

While the mathematical definition (1) is iterative in form, the definition (2) is circular, i.e. returning, and hence a recursive definition. As mentioned by Dijkstra in 1988, the recursive definition was, conceptually, a quantum leap forward.

[2]The validity of these claims will follow from Chapter 2. That Dijkstra was considered 'arrogant' follows from [45, p.4]. For the location of the symposium, see the proceedings of the IFIP Congress Münich, Germany, 1962.

```
(1)    procedure quicksort(A,M,N);  value M,N;
(2)        array A;  integer M,N;
(3)    begin
(4)        integer I,J;
(5)        if M < N then
(6)          begin
(7)            partition(A,M,N,I,J);
(8)            quicksort(A,M,J);
(9)            quicksort(A,I,N)
(10)         end
(11)   end quicksort
```

Table 1.1: `QuickSort` [54]

For, even in the 1980s, some mathematicians eschewed the recursive definition due to its circularity, claiming that circular definitions did not make sense [44].

In the context of recursive procedures, recursion can be described more precisely as *a technique involving the use of a procedure that calls itself one or more times* until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first [84]. To illustrate a recursive procedure, it suffices to glance at lines 1, 8, and 9 of Tony Hoare's `QuickSort` algorithm [54], presented in Table 1.1. Lines 8 and 9 each contain a recursive (i.e. a returning) procedure call to the procedure 'quicksort' in line 1. The algorithm was published in 1961 in the `ALGOL60` programming language[3].

In 1980, in his Turing-Aware lecture, Hoare expressed the importance of `ALGOL60`'s recursive procedures. Being able to use it concisely allowed him to discover his elegant and efficient `QuickSort` algorithm [57, p.145]. Likewise, Perlis in 1978 expressed the positive impact which `ALGOL60`'s recursive procedures had had:

> "Even though numerical analysis did not make much use of recursive procedures, inclusion of recursion made the language much more useful for important applications that would surface in the years immediately following `ALGOL60`'s birth." [101, p.86]

On the other hand, it is important to note that recursive procedures were controversial during the 1950s and 1960s [113, p.193]. Many researchers did not want to have recursive procedures in the `ALGOL60` language; others, such as Dijkstra, did. Recursive procedures and its controversy are discussed at length in Chapter 2 with a particular interest in Dijkstra's views.

Finally, it suffices to mention here that Dijkstra, in his advocation for recursive procedures, was led by what I call 'linguistic ideals'. Dijkstra, himself,

[3]The procedure 'quicksort', declared in line 1, serves the purpose of quickly sorting a list of numbers which are stored in array A. An auxiliary procedure 'partition' is used in line 7 but its definition has been omitted from Table 1.1.

used the term 'linguistic' to describe his work at the 1962 Rome symposium [27, p.241]. Likewise, and more generally, Perlis, in 1978, used the term 'linguistic' to distinguish between the research concerning the programming system `FORTRAN`, on the one hand, and the programming languages `ALGOL58` and `ALGOL60`, on the other hand:

> "Linguistic growth –unlike `FORTRAN`, which was designed for a specific machine, and for which the issues were coding efficiency and properly so, `ALGOL` was designed for arbitrary, unknown machines. Consequently, the design of `ALGOL` focused on linguistic structure. They were the first languages, both `ALGOL58` and `ALGOL60`, in which linguistic issues forged to the front." [101, p.146]

So far, I have not found any evidence to suggest that Dijkstra was influenced by the linguist Chomsky. On the other hand, Chomsky's work (e.g. [18]) may have influenced Backus, who, in 1959, developed a concise formal notation to describe the syntax of `ALGOL60`. This issue is briefly addressed in Section 3.2 and the relationship between Backus' notation and recursion is described in Appendix A. To conclude, then, linguistic ideas influenced several computer pioneers of the late 1950s and early 1960s, but these ideas did not necessarily originate from Chomsky's work.

**The Advent of Logic (Chapter 3)**

In 2001, as a computer-science engineer, I became very interested in logic after having read Martin Davis' book, *Engines of Logic: Mathematicians and the Origin of the Computer* [24]. One of the first passages in the book states:

> "Nowadays, as computer technology advances with such breathtaking rapidity, as we admire the truly remarkable accomplishments of the engineers, it is all too easy to overlook the logicians whose ideas made it all possible. This book tells their story." [24, p.xii]

Chapters are devoted to each of the following scholars: Leibniz, Boole, Frege, Cantor, Hilbert, Gödel, and Turing. Also Von Neumann and many other computer pioneers are treated towards the end of the book.

Davis' book has been praised by many, including Andrew Hodges, the biographer of Alan Turing.

> Hodges: "At last, a book about the origin of the computer that goes to the heart of the story: the human struggle for logic and truth. Erudite, gripping, and humane, Martin Davis shows the extraordinary individuals *through whom the groundwork of the computer came into being*, and the culmination in Alan Turing, *whose universal machine now dominates the world economy.*
> [24, first page, my italics]

Andrew Hodges and Martin Davis are two authors who have contributed greatly to increasing the public awareness of the importance of Alan Turing's work. With this objective in mind, Davis started the introduction of his book by contrasting the words of Howard Aiken[4] with those of Alan Turing[5].

> Aiken, 1956: "If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence I have ever encountered." [24, p.xi]
>
> Turing, 1947: "Let us now return to the analogy of the theoretical computing machines [...] It can be shown that a single special machine of that type can be made to do the work of all. It could in fact be made to work as a model of any other machine. **The special machine may be called the universal machine.**" [24, p.xi]

Later on in his book, Davis referred to the above words of Aiken:

> "Aiken made this *remarkable* assertion in *1956* when computers that could readily be programmed to do both of these things were already commercially available. If Aiken had grasped the significance of Alan Turing's paper published two decades earlier, he would never have made such a *preposterous* statement." [24, p.140, my italics]

In an attempt to summarize Davis' words: the computer pioneer Aiken was clearly lagging behind on the current events of his time. In contrast, conformity to Mahoney's *histories* of computing [77], leads to a more positive characterization of Aiken: speaking as a leading figure of his community, Aiken did not depend on Turing's 1936 theory of computation to further the research agenda of his community[6].

Inspired by Davis' book (and the previous quotes in particular), the *second theme* of my thesis, presented in Chapter 3, lies in trying to better understand how Turing, or logic in general, has *and* has not influenced specific developments in computer science. It will, for instance, turn out that many computer pioneers, including *logicians* such as Martin Davis himself, did not initially see the connection between Turing's 1936 paper and programmable computing machines. In fact, even as late as 1978, Turing's work had not been received among some prominent computer pioneers. In short, Howard Aiken's words, presented

[4] Aiken was one of the first people to build a programmable computing machine [118, p.31].

[5] Turing's 1947 quote implicitly refers to his 1936 paper [122], in which 'the universal machine' was defined.

[6] Furthermore, according to Copeland [20], Davis has taken Aiken's quote out of context and has misunderstood Aiken's point. Copeland defends the case that Aiken's quote had "nothing to do with the theoretical concept of universality and everything to do with the practicalities of dovetailing hardware to specific applications" [20, p.36].

above, were not exceptional for various communities of computing. While Turing had belonged to one community of computing, Aiken belonged to another[7].

The computer pioneers of the 1950s and 1960s can be roughly split into two groups. The first group of pioneers never really became well acquainted with Turing's 1936 paper or logic in general. The second group, including Dijkstra, did. In particular, Chapter 3 will show that Dijkstra, by grasping the relationship between the unsolvability of Turing's Halting Problem and practical engineering problems, was able to advance the state of the art in his own community, i.e. that of programming-language design and compiler building, in several ways. In retrospect then, Dijkstra's ability to link Turing's work with his own may have been exceptional for his time[8].

## 1.1 Related Work

Having described the contents of my thesis in general terms, I now address the related work.

#### Micro vs. Macro

Jacqueline Kennedy's visit to Rome in 1962 occurred at almost the same time as the Symposium on Symbolic Languages in Data Processing. From a microscopic perspective, this was probably a pure coincidence. Macro-scopically, however, the two events were related. A year prior, the missile crisis in Cuba had led to a climax in the Cold War. The West, in response, had intensified their cooperation on various levels. Jacqueline Kennedy's visit to the Pope signified, politically, the cooperation between the USA and Western Europe, while the symposium exemplified the military cooperation in terms of heavily-funded computer-science research. It lies beyond the scope of this thesis to grasp the implications that these world-wide events had on the computer pioneers of the 1950-60s; cf. Krige [70] and Campbell-Kelly [14].

#### The Contextual Development of Ideas

In the first conference on the history of computing in 1976, a plea was held by Richard Hamming to pursue the contextual development of ideas, instead of simply listing names, dates, places, and "firsts" [85, 73]. The historian Lee stressed this point further in 1996 by noting that, in the literature, ideas are

[7] The italicized words of Andrew Hodges, presented previously, may deserve some scrutiny as well. It is all too easy to overlook the engineers whose ideas and efforts made it all possible. In fact, Davis' distinction between logicians and engineers (in the very first quote) may not be the most relevant distinction to highlight. In my opinion, Turing and Von Neumann were exceptional exactly because they were *both* well-versed in logic and knowledgeable in engineering. And, I believe, it is *this* trait that, not only distinguished them from many logicians and engineers, but also led to the advent of the programmable computing machine. This thesis does not tell their story, but I could not refrain myself here from expressing my personal opinion, which, in retrospect, corresponds with Mahoney's views [76, 77, 78].

[8] Another exception was e.g. Hoare; cf. [55, 56].

almost always presented as one straight line of thought; side alleys or dead ends are typically not mentioned [73, p.56]. Given an innovation that came about, an historian should understand its environment by posing revealing questions. The personal attitudes of the people involved, their backgrounds, prior experiences, and achievements need to be studied thoroughly, along with the influences of the *known* state-of-the-art [73, p.56-57].

In Chapter 2, I will show that there were several pioneers who had, independently from each other, implemented recursive procedures for the first time; Dijkstra was one of them. In addition, I will try to capture Dijkstra's thoughts on recursive procedures and explain why he was such a strong advocate of it. A comparison will be made with e.g. English and German computer pioneers who were strong opponents of recursive procedures. And, I will try to understand how their opinions differed from Dijkstra's. Likewise, I wish to understand which parts of the state of the art were and which parts were not known among several computer pioneers of the 1950s and 1960s. To do so, I will start a specific investigation in Chapter 3, with the purpose of finding out which computer pioneers had eventually grasped Turing's 1936 theory of computation and logic in general.

Hamming, in his 1976 key-note address, also suggested that historians "go beyond the published documentation and to *speculate* about those elements of the history of projects and events that were still unrecorded" [73, p.55]. In this regard, I will present some general conclusions in Chapter 4. For instance, as mentioned previously, I speculate that, unlike many of his contemporaries, Dijkstra was able to apply the unsolvability of Turing's Halting Problem to progress the agenda of his research community. Another speculation that will be presented in Chapter 4 lies in the distinction that can be made between those pioneers who did not have a programmable computing machine at their disposal during the early 1950s and those who did. The former group consisted of people who were mainly led by linguistic ideals (i.e., Dijkstra, Van Wijngaarden, Naur), while the latter group consisted of people who were primarily concerned with down-to-earth engineering problems (i.e. Rutishauser, Samelson, Bauer, Strachey, Wilkes). In other words, being one of the firsts to have a programmable computing machine *may* have had both its advantages and disadvantages; cf. [1, p.124,125].

### Roots of Computer Science

In his 1990 paper [75], Michael Mahoney marked the electronic digital stored-program computer as the convergence of two lines of development: (i) the design of mechanical calculators capable of automatic operation, and (ii) the development of mathematical logic. In addition, Mahoney mentioned that neither electrical engineers nor mathematical logicians viewed *programming* as their main concern:

> "Working with the model of the Turing machine, mathematical logicians concerned themselves with questions of computability con-

> sidered independently of any particular device, while electrical engineers concentrated on the synthesis and optimization of switching circuits for specific inputs and outputs." [75, p.3]

Since programming is a central topic in my thesis, it is important to stress, in accordance with Mahoney's observations, the conceptual leap forward that was made by those computer pioneers who worked on what we now call programming languages and compilers. Many of these pioneers were, not incidentally, numerical analysts, since they were the ones who "embraced the machine as part of their subject and hence took programming it as part of their task" [75, p.3]. Several of these numerical analysts are introduced in Section 1.2; Dijkstra was one of them.

### Different Communities of Computing

Mahoney has also stressed the diversity amongst the creators and practitioners[9] of what we now call computer science [77, p.120]. Different groups of people saw different possibilities in computing [77, p.124]. With this in mind, Mahoney introduced the term *communities of computing* [77, p.124].

In my thesis, I mainly focus on the community of pioneers who sought techniques to overcome the tediousness in programming their machines. However, the ideas prevailing in this community were diverse. In Chapter 2, I distinguish between ideas that were used to advocate for recursive procedures and ideas that were used to oppose recursive procedures. And, in Chapter 3, I implicitly distinguish between those pioneers who eventually became acquainted with Turing's work or logic in general, and those who did not. Another, but much smaller community, is obtained by selecting those few pioneers who were well-versed in logic at the time that recursive procedures were introduced. Dijkstra, for instance, did not belong to this select group. As will be concluded in Chapter 4, the recursive procedure was promoted by some of these logically-inclined researchers, but also eschewed by others.

### Histories of Software

For any aspiring historian of computer science, Mahoney's words describe the road to follow:

> "[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." [77, p.127]

An important example of such a process is the recursive procedure, addressed in Chapter 2. As already illustrated with Hoare's `QuickSort` algorithm, the

[9] While doing so, Mahoney referred to the work of Campbell-Kelly, Cortada, Haigh, and others; cf. [77].

advent of recursive procedures changed the way in which computer practitioners thought about algorithms.

#### ALGOL60

After LISP, the first programming language to provide recursive procedures was ALGOL60. Since ALGOL60 is a central topic in Chapter 2, the reader may wish to study De Beer's thesis [9] as a secondary source: De Beer's thesis covers ALGOL60 in great generality, while my Chapter 2 solely focuses on ALGOL60's recursive procedures.

### Man-To-Man vs. Man-To-Machine Conversation

In an attempt to understand Dijkstra's views on recursion, Section 2.1 addresses the comparison Dijkstra made in 1963, between man-to-man conversation and man-to-computer conversation. Very briefly: while unpredictability prevails in man-to-man conversations, Dijkstra stressed that man-to-computer conversations can be and should be made completely predictable. In a recent paper [87], De Mol has essentially addressed the same comparison, but from a different angle. De Mol notes that, traditionally, programmers have always tried to reduce the unpredictability in man-to-machine conversations, and suggests to those interested in artificial intelligence to step away from this tradition. In De Mol's words: freer man-to-computer conversations can be obtained by accepting the unpredictability of the computer.

### The Mathematical Community

The advent of the programmable computing machine created new communities, such as that of the numerical analysts, and influenced others, such as that of the number theorists. In a very recent article [88], De Mol has studied the computer's historical impact on mathematics and its practice. In particular, she has investigated what kind of mathematical problems were implemented on the ENIAC[10] and how these implementations were perceived by two computer practitioners and mathematicians. Her main thesis is that, from its very beginning, the electronic general-purpose computer was conceived by the numerical analyst Von Neumann and the number theorist Lehmer as a mathematical instrument per se.

### Influence of Logic on Computer Science

As stated by the logician Martin Davis, concepts and methods from logic have proved to be important in computer science [22, p.149]. This claim can easily be backed up by mentioning the work of De Bakker, Scott, Rabin and several others. Instead of doing so, I prefer to merely present two specific papers. First, Wadler's didactic account in [123] of Gentzen's natural deduction and Church's

[10] Electronic Numerical Integrator and Computer.

lambda calculus shows that (i) proofs and functional programs are one and the same thing, and that (ii) simplifying a proof corresponds to executing a program. At the end of his paper, Wadler explains the practical relevance that this has with respect to mobile code for the Internet. Second, several computer scientists in [51] have presented examples from various fields: descriptive complexity, database query languages, type theory in programming languages, and automatic verification of semiconductor designs. Their main thesis states that:

> "[L]ogic has permeated through computer science during the past thirty years much more than it has through mathematics during the past one hundred years." [51, p.215]

## 1.2 Historical Context

In the rest of this introductory chapter, I introduce various important computer pioneers of the 1950s and 1960s.

While solving a fundamental problem in mathematical logic, Alan Turing found a mathematical model of an all-purpose programmable computing machine, which he published in his 1936 paper: 'On computable numbers, with an application to the Entscheidungsproblem' [122]. Several years later, between the mid-1940s and mid-1950s, a small group of engineers in the USA and England built some of the very first programmable computing machines, which were primarily used by applied mathematicians to solve numerical problems. Among those engineers and applied mathematicians were John Mauchly, Presper Eckert, Maurice Wilkes, John von Neumann, Herman Goldstine, Howard Aiken, and Alan Turing [22, 24].

In Germany, between the mid-1930s and mid-1940s, Konrad Zuse had been building some machines as well. One of them, the Z4, survived the Allied bombings [69, p.7]. While the war was ending, Zuse took his Z4 and moved westwards, away from potential Soviet occupation. Later, the Z4 would end up in Zürich where it would be used by Heinz Rutishauser, a Swiss mathematician. By first working on numerical problems, i.e. as a numerical analyst, Rutishauser started to seek specific techniques to overcome his tedious programming efforts. While doing so, he introduced algebraic expressions and a technique to translate them into machine code [69, 9, 118, p.24-25, p.9, p.41-50].

#### Rutishauser, Samelson, and Bauer

During part of World War II, Rutishauser was a Ph.D. student at the Eidgenössische Technische Hochschule in Zürich (ETH). In 1949, Rutishauser visited the computer pioneers Howard Aiken at Harvard and John von Neumann at Princeton, in order to acquire the state of the art in computing of that time. During his stay abroad, Rutishauser's boss, Eduard Stiefel, had managed in Zürich to rent Zuse's Z4 machine. So, after Rutishauser returned to Zürich, he found himself in a comfortable position. On the one hand, he was well aware of the state

of the art in computing and, on the other hand, he had a computing machine at his disposal [115, p.2].

By 1950, Rutishauser had begun working on numerical methods (i.e. scientific computing). In collaboration with Eduard Stiefel and Ambros Speiser, he wrote a series of four papers in which he covered topics such as possible number systems, fixed vs. floating point and complementation, arithmetic processes, etc.[11]. In 1951, Rutishauser submitted his habilitation at ETH, titled *Automatische Rechenplanfertigung,* in which he described a machine procedure for handling various portions of an arithmetic formula and how these could be combined to produce machine code [115, p.2-3].

Rutishauser's strong position, in terms of computing machinery and know-how, attracted his neighbours from Münich, Klaus Samelson and Friedrich Bauer. Gradually, during the 1950s, these three men increased their cooperation and friendship [115, p.3]. By the late 1950s, they were internationally respected for their expertise in automatically producing machine code from algebraic expressions[12].

The collaboration in scientific computing between Rutishauser, Samelson, Bauer, and some others, was hindered by the diversity of computing machinery: different machines were being built and used in Zürich and Münich (and in other parts of Europe). To overcome this diversity, Rutishauser appealed for a universal programming language in the 1955 GaMM[13] meeting (described in greater detail below). By 1958, The Swiss and Germans were collaborating with the Americans. This led to a one-week ACM[14]-GaMM meeting which was held in May 1958 at ETH. The chosen name for the universal programming language was initially `IAL` (International Algorithmic Language), later denoted as `ALGOL58`, but would by January 1960 change into `ALGOL` (Algorithmic Language) [9, p.31], and denoted as `ALGOL60` in this text. As programming languages, `ALGOL58` and `ALGOL60` would be drastically different [9, p.35].

### The `ALGOL` Effort

The 1955 GaMM meeting is commonly marked as the start of the `ALGOL` Effort, i.e. the international effort involved in creating a universal programming language. The `ALGOL` Effort is associated with the period that ended in 1968 with the publication of the `ALGOL68` report; cf. [9, p.4]. In this text, however, only the language `ALGOL60` (or plain `ALGOL`) is covered, not the language `ALGOL68`.

The GaMM meeting was held in Darmstadt, Germany in October 1955. Several participants of that meeting (e.g. Rutishauser) stressed the need for one *universal* and *machine-independent algorithmic* language [9, p.5]. Programs in `ALGOL` were meant to allow people to communicate algorithms with each other

[11] It is not clear to me how influential these papers were. However, Rutishauser was also the inventor of the now-famous QD algorithm, which he explained in a 74-page booklet *Der Quotienten-Differenzen-Algorithmus* [115, p.3].

[12] Cf. the comments of the American Perlis in [101] or Chapter 3 in [9].

[13] Gesellschaft für Angewandte Mathematik und Mechanik.

[14] Association for Computing Machinery.

without having to execute them on a machine [11, p.139]. The adjective *universal* referred to the aspiration that everybody would communicate with each other in the same algorithmic language. The *machine independence* expressed the desire that the language would be designed without having a specific machine in mind [9, p.6]. Of equal importance is the adjective *algorithmic*. It emphasized the fact that numerical computations were intended to be the main (if not the only) application domain of the language [91, p.101]. However, while the European participants of the `ALGOL` Effort were primarily academic numerical analysts, most of the Americans were not [101, p.141].

In Zürich, between May 27 and June 1, 1958, the Germans, Swiss, and Americans agreed on the following criteria:

1. The new language shall 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.

–cited after [100]. The `ALGOL` Effort would quickly become more international. For instance, the Dutch Aad van Wijngaarden and Edsger W. Dijkstra and the Dane Peter Naur would join the `ALGOL` Effort. The latter was to become the editor of the `ALGOL60` report [4], a document that became the standard for defining programming languages [9, p.35] for several decades[15].

### Van Wijngaarden and Dijkstra

Aad van Wijngaarden was, similar to Rutishauser, a specialist in numerical analysis who went abroad (England and the USA in 1947) to familiarize himself with the state of the art in computing. Unlike Rutishauser, he did not have a programmable computing machine at his disposal. Therefore, Van Wijngaarden and his team in Amsterdam had to build a machine themselves. Their ability to do so was strengthened by the arrival of Gerrit Blaauw, a Ph.D. graduate of Howard Aiken at Harvard, who joined the Amsterdam group in November 1952. By January 1954, the Amsterdammers had their first working programmable computing machine, the ARRA II [1, p.102-111].

Though it took quite some time (compared to the English, Americans, and Swiss) before the Dutch could actually run programs on a decent machine, as early as 1952 programs were written on paper by Edsger W. Dijkstra who had joined the Amsterdam group in March of that same year. Not being able to work with a real computer was, in hindsight, a blessing in disguise for Dijkstra and the rest of the team. Dijkstra could focus mainly on the problem domain

[15] Even today, `ALGOL`-like programming languages are being used extensively in industry (e.g. C and Java), and studied thoroughly in certain branches of theoretical computer science (e.g. [94, 95]).

and attempt to solve corresponding problems by writing his programs, without having to bother much about machine-related problems [1, 11, p.111,125, p.133].

Only from 1954 and onwards, could the Amsterdammers actually test their automatic-programming skills by running programs on their ARRA II and subsequent machines. Nevertheless, in 1960, they obtained world-wide recognition in compiler building: the Amsterdammers Dijkstra and Zonneveld were the first to implement a compiler for the `ALGOL60` programming language [1, 9, p.124-125, p.40].

### Backus, Perlis, McCarthy

In contrast to the Europeans, the Americans already had, prior to 1957, several (executable) algorithmic languages. This diversity was felt and was the incentive for the Americans to accept the invitation from the GaMM [9, p.11], resulting in the joint ACM-GaMM meeting in May 1958 at ETH, as described previously.

One of the available algorithmic programming systems in the USA was `FORTRAN`, invented by Backus and his team. Already in December 1953, John Backus proposed the `FORTRAN` project to his boss at IBM [5]. In contrast to the programming language `ALGOL60`, `FORTRAN` became a de facto standard programming language for scientific computing [9, p.11]. However, while `ALGOL60` was essentially machine independent, `FORTRAN` had six machine dependent language constructs [9, p.15].

Compared to other existing programming languages of the 1950s, `FORTRAN` was, in hindsight, the first high-level language that met two seemingly contrasting requirements. First, a `FORTRAN` program could be translated into machine code at a sufficiently low cost. Second, the obtained machine code was sufficiently economical in comparison to code that was hand written by an expert machine-level programmer[16]. To meet these requirements, Backus and his team primarily focused on the design of the translator and not on the design of the language [118, p.233].

While many computer pioneers were sceptical about the `FORTRAN` project, it is amusing to note that the creators of `FORTRAN` were, at times, as impressed by their achievements as their critics.

> Backus: "It was a really exciting period because by late summer and early fall we were beginning to get fragments of compiled programs out of the system, and we were often astonished at the code it produced. It often transformed the source code so radically that we would think it had made an error, and we'd study the program carefully, and finally realize it was correct. Many of the changes in the computation were surprising, even to the authors of the section responsible for them." [5, p.59]

After his `FORTRAN` years, Backus participated in the `ALGOL` Effort and made a significant contribution related to formal syntax (cf. Appendix A). Two other

[16] Both the words 'cost' and 'economical' refer to the combination of 'fast in time' and 'low in memory consumption'.

Americans who contributed substantially to the ALGOL Effort were Alan Perlis and John McCarthy. Perlis was the chairman of the ACM Programming Languages Committee in 1957 and a delegate to the meeting in Zürich in 1958 [17]. McCarthy was the inventor of the functional programming language LISP, which had a large influence on ALGOL60. In particular, recursive procedures were introduced by McCarthy and eventually included in the ALGOL60 language (cf. Chapter 2).

### Naur

In February 1959, Peter Naur from Denmark joined the ALGOL Effort [91, p.92]. He was the editor of the influential ALGOL report [4], and looking back in 1978 at his role, he stated:

> "I was led to the conviction that the formulation of a clear and complete description was more important than any particular characteristic of the language." [91, p.99]

After the publication of the report, he also initiated an ALGOL Bulletin, which served the purpose of discussing properties of ALGOL and promoting its use as a programming language [111, p.6].

# Chapter 2

# The Advent of Recursion

Rome, March, 1962: Edsger W. Dijkstra was sitting in the "Palazzo dei Congressi", attending the Panel Discussion on *Philosophies for Efficient Processor Construction* at the International Symposium of Symbolic Languages in Data Processing. Together with Naur, Duncan, and Garwick, he was defending the case for recursive procedures in the `ALGOL60` programming language. Even though he had become famous more than a year ago by being the first to build a compiler for the `ALGOL60` language[1], a large group of panel members remained sceptical about the usefulness of recursive procedures.

Inspection of the minutes of the panel shows that almost every panel member had a slightly different view towards why recursive procedures should or should not belong to a machine-independent programming language, such as `ALGOL60`. For instance, Dijkstra advocated for recursive procedures due to linguistic reasons (cf. Section 2.1), while Garwick was one of those few who was convinced that there were classes of problems for which recursion would come in handy [97, p.369].

According to Samelson and Strachey, general programming constructs, such as the recursive procedure, typically led to inefficient object programs. Their opinion –which was not shared by e.g. Naur and Van der Poel– made Strachey want to simplify or reduce the `ALGOL60` programming language by restricting (not necessarily discarding) the use of recursive procedures [97, 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. According to Samelson, the efficiency of the running program influenced the total cost the most and, *therefore*, he preferred to avoid general program constructs, such as the recursive procedure [97, p.364,372].

The moderator, Van der Poel, in turn, opposed Dijkstra's quest for generality, but, on the other hand, did not think that recursive procedures led to

[1] That is, the `ALGOL60` language *with* recursive procedures. Also, Dijkstra was the first to build such a compiler, *together* with his friend and colleague Zonneveld [71].

inefficient object programs[2].

The tension between several panel members was apparent [97, 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. It was directed towards those in favor of general language constructs:

> "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." [97, p.375]

## 2.1 Dijkstra's Ideology

To understand why Seegmüller's comment was loudly applauded, it will help to clarify the extreme views that Dijkstra had on programming-language design. To do so, I will rely on Dijkstra's written description of his ideology which he published only a year later in [29]. In that paper, Dijkstra wanted to concentrate on the programming language proper, not on a specific problem that could be solved by programming in that language[3]. In particular, Dijkstra wanted to focus on the linguistic demands that underly the design of a programming language [29, p.31]. To do so, he first took English as his language under study (Section 2.1.1), before he started reasoning about a programming language (Section 2.1.2) and `ALGOL60` in particular (Section 2.1.3). Finally, by presenting more excerpts from the 1962 symposium, I will show that Dijkstra's 1960 implementation of recursive procedures was *merely a by-product* of his agenda to pursue simplicity, based on linguistic ideals (Section 2.1.4). In hindsight, Dijkstra followed what today we would call a top-down design methodology, while the many who opposed his linguistic ideals were more in line with what we now call a bottom-up approach (Section 2.1.5).

### 2.1.1 The English Language as an Example

Dijkstra suggested to consider any English text that respected 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.

[2] Cf. [97, p.368,375]. On the other hand, Van der Poel later *did* express strong sympathy with "the Dijkstra language or another generalized Algol" in [98, p.642].

[3] Alternatively, I could also have based my exposition on [27], i.e. the paper Dijkstra presented at the 1962 Rome symposium.

5. A list of 2000 words is given and each word in that list may not be used.

Given any English text that obeys these five restrictions, Dijkstra remarked that (i) the readability of that text is not necessarily hindered and (ii) one can read such a text while being completely ignorant of the existence of the five restrictions. However, constructing a *correct* English text can become very problematic if more restrictions are added to the above list or especially if they impose highly implicit conditions. "In the extreme case one would need a large computer with a complicated program to check whether one's text does not violate the rules!" [29, p.31].

The previous paragraph already hints at Dijkstra's preference to avoid as many restrictions as possible, in the interest of being able to construct texts that are easy to validate in terms of correctness.

### 2.1.2 Human vs. Machine

To distinguish between a natural language (e.g. English) and a programming language, Dijkstra considered two scenarios. In the first scenario, a speaker communicates with a listener by talking in English. In the second scenario, a programmer communicates with a computer by programming. Dijkstra then explained the difference between both scenarios. Briefly, a listener is rather unpredictable in his reactions, while a computer can, essentially, be completely understood and, hence, be predictable. To exploit this advantage that a computer can have over a listener (i.e. a human), Dijkstra stressed the importance of avoiding an unnecessarily complicated computer [29, p.33,34], and expressed his disappointment with `ALGOL60` in this particular respect:

> "From this point of view the way in which `ALGOL60` is defined is rather alarming. 'Pure `ALGOL60`' is defined by the official *Report on the Algorithmic Language* `ALGOL60`, edited by Peter Naur, but reasonably speaking one cannot expect a user of the language to know this Report by heart. Specific implementations of the language are defined by translators, etc., of a couple of thousand machine instructions, a quantity which exceeds our powers of comprehension even further." [29, p.34]

### 2.1.3 Ideology Applied to `ALGOL60`

Later in his paper, Dijkstra applied his ideology to `ALGOL60`. Just like the five restrictions in his English-language example, Dijkstra wanted to reduce the number of 'unnecessary' restrictions in the `ALGOL60` language. To do so, he presented examples (see below) in which he advocated for *dynamic* instead of static constructions since they make the language more 'systematic' and 'powerful'.

One of Dijkstra's examples was based on the switch and procedure declarations in `ALGOL60`. Both declarations have a hybrid nature; i.e. an undesirable property, according to Dijkstra. On the one hand, the switch and procedure

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 switch and the procedure 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 switch or procedure declaration can not be used in such a 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 switch and procedure declaration. The result would then be that the declarators *switch* and *procedure* would only be followed by a list of identifiers, to which suitable assignments would eventually (i.e. at run time) be made [29, 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*." [29, p.36, my italics]

Dijkstra's ideology led him to the *extreme* of omitting all type indications and, hence, transferring all the type checking to the run-time system [29, p.36], which, as many observed, would have a negative effect on computation time[4]. Likewise, for arrays, Dijkstra suggested to remove the explicit specification of an array's subscript bounds, since they become determined at run time any way. The bounds were, in other words, 'redundant information' that need not be written down by the programmer. Omitting the array bounds, in turn, resulted in more freedom. For, now there was no reason to restrain an array to being rectangular; it could for instance just as well be triangular. Continuing in this manner, the homogeneity of an array need not be required either. For instance, some array elements could now be arrays again, or procedures, etc. [29, p.36] Dijkstra continued:

> "Once the type of a variable is always defined dynamically, there is not even a reason for it to be constant in time. The power of expression is increased as the language contains a smaller number of different kinds of elements and all kinds of artificial barriers have fallen away. An ordinary variable is nothing but a trivial example of a parameterless procedure. In short, the programmer now no longer needs to squeeze the relevant information into the rigid forms permitted by `ALGOL60`." [29, p.36]

With such an ideology in mind, Dijkstra was perceived as someone who totally neglected efficiency issues. Hence, it is no surprise that Dijkstra and his fellow

[4] An observation that Dijkstra did not contradict; cf. [29, p.41] and his abstract in [26]. These references also show that Dijkstra believed efficiency problems would be resolved (or at least become negligible) in the long term. 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 like Samelson, Bauer, Wilkes, and Strachey.

'linguists' were the laughing stock of Seegmüller's well-received comment. Indeed, for most people at the symposium, efficiency was important, or, at least, to a sufficient extent that it should be mentioned explicitly.

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 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*." [29, 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-dependent 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:

> "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." [29, p.30]

### 2.1.4 Ideology Leads to Recursive Procedures

Having described Dijkstra's pursuit for a general language, it is also important to note that at the 1962 symposium he was not alone. Duncan, for instance, compared a restricted version of `ALGOL60` with the actual `ALGOL60` language in accordance with Dijkstra's views.

> Duncan: "[T]here may be a significant class of problems for which, because of the restricted language, the source program may need to be more cumbersome and complicated than it would have been had the full powers of Algol 60 been available." [97, p.368]

An example of Duncan's 'restricted language' is `ALGOL60` in which procedures can not call other procedures[5]. As we shall see in Section 2.2.2, such a restriction was most notably supported by Samelson and Bauer. Dijkstra, on the other hand, allowed the programmer to use procedure calls in full generality. In response to Duncan's comments, Dijkstra replied:

> "[...] you are not only hindered by restrictions that prohibit you to do things [such as calling a procedure from within another procedure], it is even so that you gain by possibilities [such as a recursive

[5] That is, only the main body of the program can call a procedure.

> procedure] that are not actually used in the program at all." [97, p.368]

These words capture Dijkstra's insistence on generality: he wanted procedure calls of any kind to be in the language, so that the language would be simple and, hence, easy to translate. Allowing some kinds of procedure calls and prohibiting others, was in violation of Dijkstra's linguistic ideals. Once simplicity was obtained by means of generalization, positive, unexpected results would follow:

> <continued> "One of the 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." [97, p.368, my italics]

In other words, it seems that Dijkstra's prime –if not only– concern was to pursue simplicity by means of general principles[6]. Finding a way to implement recursive procedures was *merely a by-product* of his research agenda. In Section 2.2.2 we will see how Dijkstra, by means of generalization, implemented recursive procedures.

### 2.1.5 Bottom-Up vs. Top-Down

Seegmüller's remark was only one of many that expressed common dissatisfaction with the linguists à la Dijkstra. Another example was Strachey's comment, supporting the prevailing doctrine of taking efficiency into account during language design.

> Strachey: "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 [...]" [97, p.368]

Incidentally, it is interesting to note that Strachey used the verb 'simplifying' to denote the opposite action of what Dijkstra would have associated with that verb.

In response to Strachey's comment, the moderator of the panel, Van der Poel, responded by characterizing two disjoint groups of panel members. On the one hand, there were those who wanted to restrict the programming language (under study, i.e. `ALGOL`) to make it fast by avoiding recursion. On the other hand, there were those who wanted a general language, but who also claimed

[6] In May 2000, a symposium was held at the Department of Computer Science at the University of Texas to honour Dijkstra. The symposium was called: *In Pursuit of Simplicity.*

that their programs could be made just as fast[7] [97, p.369]. These two different schools of thought seem to correspond, more or less, with what Naur in 1978 called the "restrictionists" and the "liberalists" [91].

Instead of contrasting between restrictionists and liberalists, I prefer to distinguish between bottom-up and top-down design. The bottom or down part corresponds to the machine, while the up or top part denotes the machine-independent programming language `ALGOL60`.

Seegmüller, Strachey, Samelson, Bauer, and many others wanted to indirectly take specific machine features into account while defining the machine-independent language `ALGOL60`. For instance, since they believed that recursive procedures were inefficient to execute, they wanted to restrict the `ALGOL60` language such that recursive procedures were avoided, without tampering with `ALGOL60`'s machine-independence. That is, these people followed, more or less, what today we would call a bottom-up methodology. They certainly did not work in accordance with a top-down methodology.

Dijkstra, on the other hand, explicitly abstracted away specific machine features (cf. Section 2.1.3) and wanted to first concentrate on the language. Only *after* having defined the language, did he want to take efficiency issues into account. Hence, Dijkstra did work in conformance with what we would now call a top-down methodology.

The reason why Dijkstra followed a top-down methodology[8] may, as hinted in [1, p.124,125], lie in the fact that he did not have a programmable computing machine during the early 1950s, while those mentioned who worked bottom-up, did. Those who were "firsts" in having a programmable computing machine at their disposal, were confronted relatively quickly with its finite limitations. Presumably, this made them take efficiency to be their prime concern.

## 2.2 Dijkstra's Wonderful Year: 1960

The recursive procedure, which Dijkstra openly advocated for in Rome 1962 and which led to Seegmüller's nasty comment, had already been introduced in 1960 in the *definition* of the programming language `ALGOL60` by Van Wijngaarden and Dijkstra via a peculiar telephone call with Naur (Section 2.2.1). Subsequently, but still in 1960, Dijkstra and his colleague Zonneveld succeeded in being internationally the first to *implement* `ALGOL60`, i.e. by building the first `ALGOL60` compiler (Section 2.2.2). Both of Dijkstra's contributions were, as we shall see, due to his aptitude for linguistics.

[7] It is true that various participants of the symposium defended the case that recursive procedures did *not* lead to inefficient object programs. But Van der Poel's characterization is, to say the least, an over-simplification. Recall that Van der Poel, himself, did not believe that recursive procedures led to inefficient programs but that he was also against Dijkstra's quest for generality. Also, as mentioned before, Dijkstra did believe that recursive procedures, or any general program construct for that matter, incurred a run-time penalty, while he was in pursuit of a general language.

[8] Also Naur, van Wijngaarden and other 'linguists' followed this approach closely.

### 2.2.1 Defining Recursive Procedures

As part of the ALGOL 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. These language restrictions automatically prevented recursive procedure activations [91, p.108,109,151,154]. Such restrictions were exactly what Samelson and Bauer (and many others) wanted in accordance with their bottom-up methodology (cf. Section 2.2.2).

In contrast, on the other side of the Atlantic, John McCarthy *was* trying to use recursion in his programming. When working at IBM in the summer of 1958, he tried to use FORTRAN to write a 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 another derivative (namely that of $y$)– McCarthy realized that he needed recursive conditional expressions. Since FORTRAN did not contain recursion either, he tried to add it to the language, but without success. This, in turn, led him to develop his own programming language LISP [116, p.27], heavily inspired by previous work of Newell, Shaw, and Simon (cf. Section 3.2.3).

In August 1959, McCarthy wrote a letter[9] in which he openly advocated for recursive procedures [80], and, in January 1960, at the final ALGOL60 Paris conference, McCarthy suggested to add recursive procedures to the ALGOL60 language [116, p.30]. With regards to McCarthy's proposal to add recursive procedures, an American representative to the ALGOL60 Conference (perhaps McCarthy himself but probably not) proposed to add the delimiter **recursive** to the language, to be used in the context **recursive procedure** [91, p.112]. The American's proposal was, by voting, turned down by a narrow margin [91, 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 non recursive procedures by means of the proposed delimiter [91, 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 [91, p.159-160]– assumed that recursive procedures did not belong to the language. In short, and in Perlis' words: "it is not clear what the votes meant!" [91, p.160].

The voting, discussed previously, took place before other issues, concerning the (informal) semantics of procedures, had been clarified [91, p.160-161]. After the voting, and after several modifications were made to the semantics of procedures, the defined language (ALGOL60) implicitly allowed for recursive procedures to be expressed syntactically, contrary to the November 1959 decision (see above) to prohibit this explicitly [91, p.112].

Whether Van Wijngaarden and Dijkstra belonged to the first or the second category of people, described above, or to neither category, is not entirely clear. But, on approximately February 10, 1960, Van Wijngaarden, alongside Dijkstra,

[9] Unfortunately, I have not been able to find this letter, so I have not read it. But Perlis refers to this letter in [101, p.86].

called the ALGOL editor, Naur, by telephone to point to a lack of definition in Naur's report [91, p.112]. The Dutch had stumbled upon the possibility to syntactically express recursive procedure activations, noting that it was nowhere stated in the ALGOL report whether recursive procedures activations were indeed intended semantically or not. Therefore, Van Wijngaarden, after consulting Dijkstra, suggested to Naur to add one sentence to the ALGOL report so that it would be clear that recursive procedure activations were allowed 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 [91, p.112]. In line with Dijkstra's linguistic ideals, described in Section 2.1, it is clear that Van Wijngaarden and Dijkstra were primarily reasoning along linguistic lines and *not* in terms of specific machine features. Simplicity for them meant less language restrictions. Naur, being linguistically inclined as well, was charmed by the one-sentence clarification of the Dutch and added it to the ALGOL report.

> Naur in 1978: "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)." [91, p.112-113]

Naur's reference to Bauer shows that he was well aware of the Germans' strong will to prohibit recursive procedures from the language, in accordance with the November 1959 meeting.

### 2.2.2 Implementing Recursive Procedures

Dijkstra's ability to implement an ALGOL60 compiler in such a short time span, was, as we shall see, again due to his pursuit for simplicity. Fundamental to this discussion will be the concept of a *stack*, which Dijkstra borrowed from Samelson and Bauer (Section 2.2.2.1). I will mainly explain and to some extent speculate why Samelson and Bauer did not implement recursive procedures (Section 2.2.2.2), and then explain Dijkstra's generalization on how a stack could be used in order to implement recursive procedures (Section 2.2.2.3). Many examples presented below are conceptualizations of the actual translation techniques used by Samelson, Bauer, and Dijkstra.

#### 2.2.2.1 Illustration of Samelson and Bauer's Stack

Samelson and Bauer's stack principle [6] can be illustrated by means of the algebraic expression:

$$A + (B - C) \times (D/E + F)$$

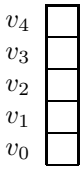
which was first translated into postfix notation:

$$A\,B\,C - D\,E\,/\,F + \times +$$

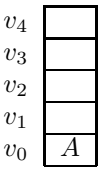
by a trivial algorithm[10].

In postfix notation, the operators (e.g. $-$) are placed after the operands (e.g. $B$, $C$). The presented postfix expression was subsequently read from left to right and, while doing so, the stack was used to store information, as is illustrated below. In the meantime, corresponding object code was also generated, as desired, but I shall not illustrate this explicitly[11]. In this simple example, the important thing to remember is that the stack was solely used prior to executing the object program; i.e. solely at what today we would call compile time.

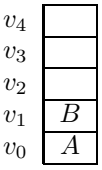
A stack was merely a part of the memory of the computing machine, consisting of successive memory locations: $v_0$, $v_1$, $v_2$, ...:



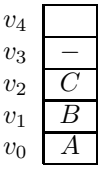

Scanning the postfix expression from left to right implies that the first symbol scanned is the symbol $A$. This symbol was placed on the stack by the translation algorithm[12], resulting in:




The second scanned symbol is $B$ and was likewise placed on the stack:




Similarly, scanning the next two symbols resulted in:




[10]This explicit translation step is already a simplification of Samelson's and Bauer's actual approach.

[11]For a more thorough example, see Knuth [69].

[12]The translation algorithm is illustrated in this section but the definition of the algorithm is not; cf. [69].

Contrary to the previous cases, we now have an operator on top of the stack[13]. This meant that enough information had been gathered on the stack to perform the corresponding operation. To be more precise, the minus sign was removed from the stack, together with the two operands $C$ and $B$. (Then, the machine code instruction of $B-C$ was generated and placed into the instruction memory of the computing machine.) The 'result' of the subtraction was placed on the stack[14]:

| | |
|---|---|
| $v_4$ | |
| $v_3$ | |
| $v_2$ | |
| $v_1$ | $B-C$ |
| $v_0$ | $A$ |

Next, the symbols $D$, $E$, and / were placed on the stack:

| | |
|---|---|
| $v_4$ | / |
| $v_3$ | $E$ |
| $v_2$ | $D$ |
| $v_1$ | $B-C$ |
| $v_0$ | $A$ |

Again, because an operator topped the stack, this meant that enough information had been gathered to perform the corresponding operation. That is, the division symbol was removed from the stack, together with the two operands $E$ and $D$. (Then, the machine code instruction of $D/E$ was generated and placed into the instruction memory of the computing machine.) The 'result' of the division was placed on the stack:

| | |
|---|---|
| $v_4$ | |
| $v_3$ | |
| $v_2$ | $D/E$ |
| $v_1$ | $B-C$ |
| $v_0$ | $A$ |

Next, symbol $F$ was read and placed on top of the stack:

| | |
|---|---|
| $v_4$ | |
| $v_3$ | $F$ |
| $v_2$ | $D/E$ |
| $v_1$ | $B-C$ |
| $v_0$ | $A$ |

[13]Strictly speaking, the operator did not have to be put on the stack, but for simplicity I have chosen to do so here any way.

[14]This is strictly speaking incorrect. The 'result' can only be known if the actual values of $B$ and $C$ are known at compile time, which, of course, was typically not the case. So, instead of the 'result', it was e.g. a register name $n$ that was stored in stack cell $v_1$, with $n$ being the name of the register that, at run time, would contain the result of $B-C$.

Then the + sign was read, placed on top of the stack, consequently removed from the stack, along with $F$ and $D/E$ ..., resulting in:

| | |
|---|---|
| $v_4$ | |
| $v_3$ | |
| $v_2$ | $D/E + F$ |
| $v_1$ | $B - C$ |
| $v_0$ | $A$ |

Next, the symbol $\times$ was read, placed on top of the stack, ..., resulting in:

| | |
|---|---|
| $v_4$ | |
| $v_3$ | |
| $v_2$ | |
| $v_1$ | $(B - C)\times(D/E + F)$ |
| $v_0$ | $A$ |

Finally, the last symbol + resulted in:

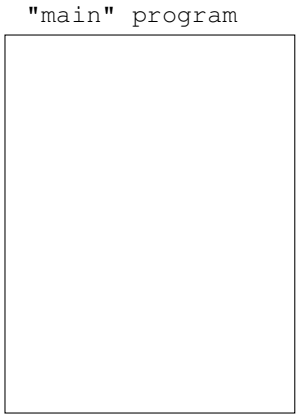
| | |
|---|---|
| $v_4$ | |
| $v_3$ | |
| $v_2$ | |
| $v_1$ | |
| $v_0$ | $A + (B - C)\times(D/E + F)$ |

and the entry of $v_0$ was removed, resulting in an empty stack.

The previous illustration is, I stress again, a conceptualization. The translation process started at the left most symbol in the postfix notation and with an empty stack. The translation process ended at the right most symbol and with an empty stack. In the meantime, machine code was generated and placed in the instruction memory of the computing machine, while the stack was used for administrative purposes. The actual values of $A$, $B$, $C$, ..., $F$ were, of course, not known during the translation. The crux lies in that the stack was only used prior to executing the object program. In other words, the generated machine code did not need a stack to execute; it only used a constant number of registers[15] to store the values of $A$, $B$, ... and to store intermediate results, such as the values of $B - C$ and $D/E$.

The algebraic expression, presented above, is a very simple example of a program that was written in a machine-independent programming language, such as `ALGOL60`. In general, the program could contain multiple algebraic expressions, and also for loops and other kinds of control-flow constructs. Let us denote such a program by "main", and depict it by an empty rectangle:

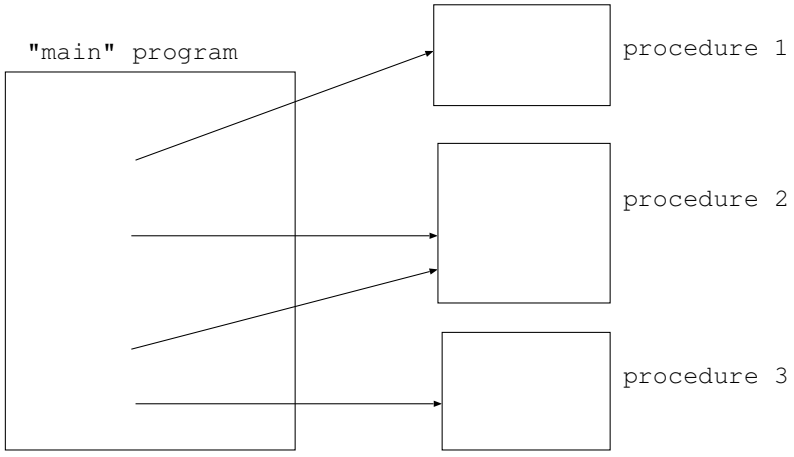
[15] Or, in more general terms, a fixed amount of memory.




This representation will come in handy in the next section.

#### 2.2.2.2 Extending Samelson and Bauer's Approach to Incorporate Procedure Calls

The previous example can now be extended. Instead of only considering a "main" program as above, we now also have some procedures. Each procedure, internally, contains an algebraic expression (or multiple algebraic expressions in general), similar to the one used in the previous discussion. Some of the algebraic expressions in the "main" program, however, contain procedure calls to one or more of the procedures. Conceptually, we have:




Each box denotes text in the machine-independent programming language, and each arrow denotes a procedure call from the "main" program to one of the procedures. Note, in particular, that each procedure does not contain procedure calls. For instance, procedure 1 does not call any of the procedures, including itself. Indeed, Samelson and Bauer did not allow procedures to call other procedures, in accordance with the November 1959 meeting (cf. Section 2.2.1). We will shortly see why they enforced this language restriction. That is, a program-

mer was not allowed to write a program in which one procedure was called from within another procedure: only the "main" program could contain arrows that pointed outwards.

A natural way to compile a program that was composed of a "main" program and three separate procedures, as pictured above, was to, conceptually speaking, compile each piece of text in conformance with the stack-based scheme addressed previously. That is, the "main" program was, essentially[16], translated as explained before, and each of the procedures was translated in a similar manner. The net effect was, again, that no stack was needed at run time, but, now, not only did the translated "main" program have its own working space, also each translated procedure had its own private fixed working space (which it would use at run time).

In other words, Samelson and Bauer treated each procedure *statically*, by providing each of them with their own *fixed* working space[17]. This approach naturally prevented a procedure to be activated more than once during the execution of the program, because only one fixed working space was available for each procedure. In particular, Samelson and Bauer's approach could not handle a procedure that recursively called itself. Dijkstra summarized Samelson and Bauer's approach:

> "If every subroutine [i.e. procedure] has its own private fixed working spaces, this has two consequences. In the first place the storage allocations for all the subroutines together will, in general, occupy much more memory space than they ever need simultaneously, and the available memory space is therefore used rather uneconomically. Furthermore –and this is a more serious objection– it is then impossible to call in a subroutine while one or more previous activations of the same subroutine have not yet come to an end, without losing the possibility of finishing them off properly later on." [26, p.312]

As the previous words indicate, and as Samelson and Bauer, themselves, admitted at the 1962 Rome symposium, their implementation led to excessive use of memory:

> [...] it was decided [to minimize run time], *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.* [112, p.214, my italics]

The first sentence in the quote stresses that Samelson and Bauer did not want to use a stack at run time –contrary to Dijkstra, as we shall see later– because they were afraid that it would increase the execution time of their programs.

[16] Some additional book-keeping was needed to implement the arrows in the figure.

[17] Today, we would say that Samelson and Bauer statically allocated memory space for their procedures. Dijkstra, as we shall see, advocated for dynamic memory management.

Therefore, they decided to avoid a run-time stack and, instead, statically assigned memory space to *each* procedure separately. This implementation choice, in turn, forced them to impose their language restriction: a programmer was not allowed to write a program in which one procedure called another procedure. In particular, recursive procedure activations were ruled out.

The last sentence in the quote is a bit ironic since Samelson and Bauer were strong advocates of *practical* (i.e. efficient) engineering-based solutions. Samelson and Bauer openly distanced themselves from people such as Dijkstra, Van Wijngaarden, and Naur who did not seem to care much about efficiency, but, instead, primarily advocated for `ALGOL60` to be as general as possible. The irony, thus, lies in the fact that Samelson and Bauer's own technique was not very good, even though they had restricted the use of the programming language, while Dijkstra, for instance, had not.

As mentioned before, the previously presented quote was stated by Samelson and Bauer [112] in the proceedings of the 1962 Rome symposium. It is important to note that this was two years *after* Dijkstra had already explained in his 1960 paper [26] how Samelson and Bauer's original stack principle [6] could be generalized in order to implement recursive procedures. It is not unthinkable that Samelson and Bauer, prior to 1960, did *not* know how to handle recursive procedures. On the other hand, this conjecture can be countered in two ways.

First, Samelson and Bauer explicitly stated, in their 1962 paper [112, p.214], that they simply did not see why recursion was important for numerical applications in the first place. As mentioned in Chapter 1 by citing Perlis, it is indeed so that the usefulness of recursion was questioned by many. For instance, only by 1963, did Rutishauser find two examples of recursion for numerical computations that he, himself, found convincing, i.e. in which recursion was indispensable. He also contrasted these examples with others[18] in which recursion could, and in his opinion should, be replaced by iteration [110].

Second, Rutishauser, who was Samelson and Bauer's close friend, may have had already implemented recursive procedures on his ERMETH several years earlier. In this regard, however, it is worth noting that Dijkstra, in his 1960 paper [26], mentioned that one of the referees had sent him a copy of a report[19] in which Rutishauser's ideas on recursion were described. Dijkstra insisted that his work was essentially different from that described in the report:

> "The author [i.e., Dijkstra] of the present paper thinks, however, that [in that report] the principle of recursiveness has not been carried through to this ultimate consequences [sic] which leads to logically unnecessary restrictions like the impossibility of nesting intermediate returns and the limitation of the order of the subroutine jump (cf. section F 44 of the report)." [26, p.313]

[18] 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 $(0, 1, 2, \ldots, n)$ than by recursive procedure activations $(n, n-1, n-2, \ldots, 0)$.

[19] The report 'Gebrauchsanleitung für die ERMETH' of the Institut für Angewandte Mathematik der ETH, Zürich –a description by Heinz Waldburger of some of the techniques developed by H. Rutishauser in his lectures.

So, in retrospect, while Dijkstra may not necessarily have been the sole inventor on how to implement recursive procedures[20], the previous quote leads to the conjecture that Dijkstra's approach was more generally applicable or simpler than many other existing implementation techniques (cf. Chapter 4). His approach is described next.

#### 2.2.2.3 Dijkstra's Approach

Samelson and Bauer's stack-based technique (cf. Section 2.2.2.1), denoted as $T$ in the sequel, was extended by Dijkstra in a 'simple elegant' fashion [107, p.181]. To illustrate this, recall the algebraic expression:

$$A + (B - C) \times (D/E + F)$$

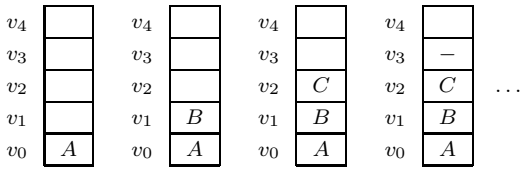
which, for brevity, we shall shorten to:

$$A + (B - C)$$

along with the corresponding post-fix notation:

$$A\,B\,C - + \tag{2.1}$$

Translating 2.1 in accordance with technique $T$ would result in the following stack-based behavior, presented as a sequence of stacks:




Dijkstra pondered about the applicability of technique $T$ in the case that variable $B$ was, instead, a compound term, such as:

$$B = (P/Q)$$

The corresponding algebraic expression would then be:

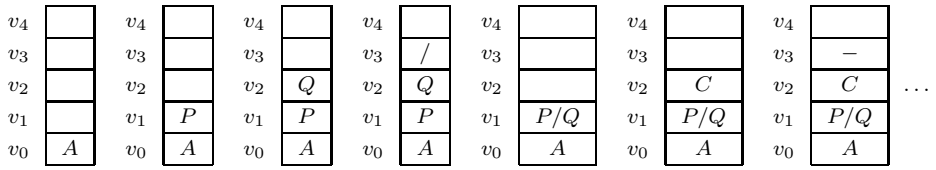
$$A + ((P/Q) - C)$$

which, in post-fix notation, would amount to:

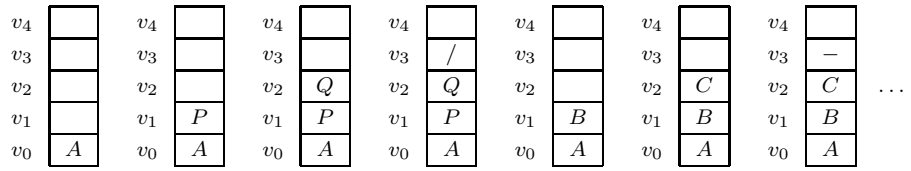
$$A\,P\,Q\,/\,C - +$$

The stack-based behavior, in accordance with $T$, would then be:

[20] In fact, in Section 2.3 we shall see that several people had invented ways to implement recursive procedures; Dijkstra was only one of them.




Now, if $B$ were to be substituted for $P/Q$ in the previous sequence of stacks, then this would result in a third sequence of stacks:




By comparing the first and the third sequence of stacks, Dijkstra concluded that: regardless of whether $B$ is ready-made or whether a number of next stack locations is needed for its evaluation, the net result is the same [26, p.314]. Likewise, whether $B$ is ready made or whether it is a call to some procedure that contains the expression $P/Q$, the net effect remains the same. In Dijkstra's exact 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 [i.e. procedure], 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." [26, p.314]

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. That is, while Samelson and Bauer solely used a stack[21] prior to execution, Dijkstra, on the other hand, suggested to use a stack during execution; i.e., at run time. As a by-product of Dijkstra's pursuit for simplicity, illustrated by the previous quote, 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." [26, p.317]

[21] Samelson and Bauer, in fact, used more than one stack (prior to execution) [6]. But my emphasis lies in explaining *when* the stack was used, rather than how many stacks were used.

## 2.3 Several Pioneers Implemented Recursive Procedures for the "First Time"

Dijkstra's 1960 paper [26], in which he generalized Samelson and Bauer's original stack principle [6] and thereby found a way to implement recursive procedures, became a very influential paper in subsequent years [107, p.181]. More generally, Dijkstra's successful pursuit for simplicity did not go unnoticed internationally, as illustrated by Wilkes' comments in the 1962 Rome symposium:

> "[Dijkstra] has indicated, and with good reason, that compiler writing may well become trivial, and is becoming trivial, and he has illustrated this very brilliantly in his compiler. [...] Dijkstra has reduced the writing of [an] Algol compiler to a triviality, in terms of his notations [...]" [96, p.198-199]

Wilkes was, very likely, referring to [27], the paper Dijkstra presented at the 1962 Rome symposium. Also Rosen's comment in his 1966 book on *Programming Systems and Languages* credits Dijkstra:

> "Recursive programming by Professor Dijkstra is an early and important contribution to the art of writing compilers. The problems involved in permitting recursive calls on subroutines are attacked and handled in a *simple elegant* fashion. 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." [107, p.181, my italics]

Nevertheless, several people had, by 1960, invented ways to implement recursive procedures. For instance, Irons and Feurzeig [61] came up with essentially the same idea as Dijkstra and in the same year [9, p.51]. Rutishauser, in his 1963 paper [110, p.50], did not only credit Dijkstra as the inventor of a technique to implement recursion, but also the Americans Sattley and Ingerman [114]. Also in the USA, the list processing language IPL of Newell, Shaw, and Simon [92, 93], which had recursion and a stack built in [83, p.192-193], already existed in 1957! Whether Dijkstra was aware of IPL or not is not clear. McCarthy, however, *was* inspired by IPL which, contrary to his own invented `LISP`, was a machine language [83, p.187]. McCarthy's `LISP` [81] was perceived by many as innovative partly due to its built-in recursion [113]. Recall that McCarthy had joined the `ALGOL` Effort after already having worked on `LISP` and that it was McCarthy who had advocated for recursion in `ALGOL60` in August 1959[22]. Perlis even stated in 1978 with respect to the `ALGOL` Effort that:

> "The implications of recursion were not really understood, except by McCarthy." [91, p.160]

[22] See [101, p.86], [116, p.30], and [80].

Finally, Turing, in the late 1940s, had already thought through the idea of using a stack for recursive procedures[23]. In fact, Bauer, confirmed this and also mentioned Rutishauser[24], Van der Poel, and Huskey as people who had implemented recursive procedures prior to 1960 [8, 7, p.39,-]. In a very recent paper [53], the suggestion has been made that 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 prior to Dijkstra's publication in 1960. In fact, at the end of his 1960 paper [26], Dijkstra thanked Huskey for the inspiring conversations that he had had with him in Amsterdam during the summer of 1959. More research is required to carefully investigate this matter.

[23] See [15] and [24, p.188, 237].

[24] Rutishauser, himself, mentioned in a 1963 paper [110, p.50] that he had implemented recursive subroutines for the ERMETH in the "pre-ALGOL days".

# Chapter 3

# The Advent of Logic

As stated by the logician Martin Davis:

> "There are many examples of important concepts and methods first introduced by logicians which later proved to be important in computer science." [22, p.149]

The objective of this chapter is to start a broad investigation in understanding how mathematical logic has and has not influenced various developments in computer science.

While Alan Turing and John von Neumann were both involved in building some of the very first programmable computing machines and were well versed in mathematical logic, most of their contemporaries were not. In Section 3.1, I will show that logic, in general, and the theory of computation, in particular, were not directly received by many computer pioneers of the 1950-60s. Also, logicians, themselves, did not initial see the connection between Turing's 1936 theory of computation and programmable computing machines. Finally, I will show that many computer pioneers, in the aftermath of their successes, openly distanced themselves from the mathematical-logic community. Nevertheless, in Section 3.2, I will stress that some ideas from logic did positively influence certain early developments in computer science, albeit in an indirect manner. Finally, in Section 3.3, special attention will be paid to Dijkstra who, unlike many of his contemporaries, not only became acquainted with Turing's 1936 theory of computation, but also was able to apply the unsolvability of Turing's Halting Problem to progress the agenda of his research community.

## 3.1 Theory of Computation vs. Programmable Computing Machines

While solving a fundamental problem in mathematical logic, Alan Turing found a mathematical model of an all-purpose programmable computing machine,

which he published in his now-famous 1936 paper [122]. Several years later, a small group of engineers built some of the very first programmable computing machines. Among them were Alan Turing and John von Neumann, two men who were well-versed in mathematical logic [22, 24].

Unlike Turing and Von Neumann, many computer pioneers did not see the connection between Turing's 1936 paper and modern electronic computers. For instance, recall from Chapter 1 that Martin Davis mentioned in [24, p.140] that even as late as 1956, there were pioneers, such as Howard Aiken, who clearly had not read or grasped the significance of Turing's 1936 paper. Davis also made a similar statement with respect to Herman Goldstine [22, p.167].

### 3.1.1 Sammet & Hopper

Even though the theory of computation became more popular during the 1950s and 1960s among some pioneers[1], whether the computing community at large, grasped Turing's theory, is another matter. In this respect, it is worthwhile to consider a fragment from Sammet's 1969 book, *Programming Languages: History and Fundamentals*.

> Sammet: "Recursive procedures were introduced by `ALGOL`. They certainly should be considered a significant contribution to the technology, but it is not clear how great a one. The advocates of this facility claim that many important problems cannot be solved without it; on the other hand, people continue to solve numerous important problems without it and even in a few cases manage to handle (sometimes in an awkward way) some of the problems which the recursion proponents claim cannot be done." [113, p.193]

Given that `ALGOL60`'s definition allows one to express potentially unbounded while loops [4, p.308], it follows from Kleene's normal form theorem[2] that recursive procedures are *not* needed. In other words, the expressive power of `ALGOL60` is not reduced by discarding recursive procedures. This immediately settles the question Sammet described in the above fragment and, therefore, illustrates that certain implications of the theory of computation, that are considered trivial today, were not understood by a respectable group of computer pioneers in the 1960s.

Grace Murray Hopper was one of the first people to develop a compiler [68, p.42]. Her pioneering work, however, was slowed down by her superiors who did not believe that computers could do more than plain arithmetic [58, p.9]. In 1978, she reflected upon these difficulties by stating:

> "I think I can remember sometime along in the middle of 1952 that I finally made the alarming statement that I could make a computer

[1] Several examples, presented later, will support this claim. For instance, Rice (Section 3.2.5) and Dijkstra (Section 3.3).

[2] Presented in Minsky's 1967 book [86, p.184], but already published by Kleene in 1936 in [65]. See also Harel's [52] in which he explains the relationship between Kleene's normal form theorem and the *while* construct.

> do anything which I could completely define. I'm still of course involved in proving that because I'm not sure if anybody believes me yet." [58, p.9]

The last sentence shows that, even as late as 1978, Hopper and, hence, also many of her colleagues, were not well acquainted with Turing's theory of computation. For, the crux of Turing's 1936 paper is that there *are* well-defined problems that cannot be computed. In fact, Dijkstra wrote in [42, p.13] that, even in 1978, Turing may have been unknown among many computer scientists.

### 3.1.2 Logicians Did Not Initially Connect Turing's 1936 Paper With Computing Machines Either

While many computer pioneers were either not aware of Turing's 1936 paper, or did not see the connection between Turing's paper and modern electronic computers, the following words from the logician Martin Davis show that logicians did not necessarily see the connection either:

> "My experience as an ORDVAC programmer led me to rethink what I had been doing with Turing machines in the course I had just finished teaching. I began to see that Turing machines provided an abstract mathematical model of real-world computers. (It wasn't until many years later that I came to realize that Alan Turing himself had made that connection long before I did.) I conceived the project of writing a book that would develop recursive function theory (...) in such a way as to bring out this connection. I hardly imagined that seven years would go by before I held in my hand a printed copy of *Computability & Unsolvability*." [23, p.60]

Davis also mentioned that one of the reviewers of his book *Computability & Unsolvability* [21], published in 1958, derided the connection he was proposing with actual computers [23, p.66]. In other words, around more or less the same time when Aiken made his "preposterous" statement (according to Davis; cf. Chapter 1), a reviewer of Davis' book essentially made a similar "mistake". Presumably, this reviewer *was* well-versed in logic while Aiken was not. Finally, the previous quote also shows that the *logician* Davis, himself, did not initially see the connection between Turing's 1936 work and programmable computing machines.

#### Blaricum, 1961

To illustrate how much the logicians were separated from the computer practitioners during the 1950s, it is worthwhile to mention the 1961 conference that was held in Blaricum, the Netherlands, and organized by the logicians. Among the attendees were Beth, Wang, McCarthy, Burks, and Chomsky –just to name a few (see [19]). Their goal was to address the implications that the computer might have on their own profession. In more specific terms, their objectives

were to (i) survey various non-numerical applications of computers (e.g. language translation, theorem proving) and (ii) address some aspects of the theory of formal systems. Selected works were published in a book [19] in 1963 with a preface stating:

> Symbol manipulation plays an important role both in the theory of formal systems and in computer programming and one would therefore *expect* some important relationships to exist between these domains. It may therefore seem *surprising that specialists in the two fields have only recently become interested in one another's techniques*. This situation is *probably* due to an original difference in motivation and to a phaseshift in time. [my italics]

The book shows how the logicians were discovering and sometimes rediscovering[3] the fruitful interplay between mathematical logic and the programming of a real (i.e. finite) computing machine. For example, Beth rediscovered the finiteness of practical computing and discussed the implications this had on conducting proofs in mathematical logic (see [19, p.29-30]).

The seventh chapter of the book, 'Programming and the Theory of Automata', written by A.W. Burks, deserves further comment for later (cf. Section 3.3.1). In that chapter, Burks explained the relationship between a "Turing Machine" and Von Neumann's cellular automaton and then formalized the notion of automatic programming[4]. One of Burks' main conclusions was the within-limits interchangeability of software and hardware [19, p.114].

### 3.1.3 Emphasis That They Were Not Logicians

In the late 1970s, many computer pioneers of the 1950s and 1960s took the effort to stress that they were *not* mathematical logicians, and that their pioneering work was either not based on logic, or, if it was, that they did not completely understand how. The following examples come from the 1978 symposium on 'History of Programming Languages'.

#### Perlis

Alan Perlis, while discussing the `ALGOL60` language in 1978, emphasized that the word 'types' did not come from the mathematical-logic community:

> "But clauses, blocks, types –types for example. Where did that come from? Did it come from a prolonged contact with logicians? Not at all! Types, in a sense, came into programming because we needed a

[3] Indeed, Turing and Von Neumann would have been well aware of some of the addressed issues if they had still been alive in 1961.

[4] Automatic programming served the purpose to make the programmer more effective in his programming by letting the computing machine take over the responsibilities of the programmer. E.g., the purpose of the `ALGOL` Effort, and `ALGOL60` in particular, was to improve the state of the art in automatic programming.

> word to indicate types, and we used it the same way logicians did; they don't own it." [101, p.145]

Perlis also mentioned that it took until 1967 when he, together with Galler, actually discussed constructive type theory in accordance with mathematical logic [101, p.80].

### Hopper

One of Grace Hopper's main career objectives was to provide a means for ordinary people (e.g. engineers and business people), not programmers, to solve problems on a computer. In 1978, she hoped that the programming-language community at large would recognize the great variety of people who wanted to solve problems on a computer. In this regard, she mentioned the need for different languages "rather than trying to force them all into the pattern of the mathematical logician", and subsequently emphasized that a lot of computer people were not mathematical logicians [58, p.11].

### McCarthy

John McCarthy, while discussing his `LISP` programming language in 1978, stated that, though he had borrowed Church's lambda notation to design `LISP`, he had not completely understood Church's lambda calculus. In particular, he had not understood Church's higher order functionals and, therefore, had chosen to use conditional expressions in `LISP` instead [83, p.176]. McCarthy continued by explaining that he did not know that his conditional expressions, together with recursion, were sufficient to express any computable function:

> "And so, the way in which to do that was to *borrow* from Church's Lambda Calculus, to borrow the lambda notation. Now, having borrowed this notation, one of the *myths* concerning `LISP` that people think up or invent for themselves becomes apparent, and that is that `LISP` is somehow a realization of the lambda calculus, or that was the intention. The truth is that *I didn't understand the lambda calculus, really.* In particular, *I didn't understand that you really could do conditional expressions [and] recursion* in some sense in the pure lambda calculus. So, it wasn't an attempt to make the lambda calculus practical, although if someone had started out with that intention, he might have ended up with something like `LISP`." [83, p.190, my italics]

In retrospect, while McCarthy stressed that he was not a mathematical logician, he had, to some extent, been influenced by Church's lambda calculus and, hence, by logic in general. Section 3.2 will present more of such examples.

**Strachey**

Around 1965, Strachey wrote a letter to the editor of the Computer Journal, titled, 'An impossible program' [120]. In this letter, Strachey presented his own proof of the unsolvability of Turing's Halting Problem, which Turing of course had already published in his 1936 paper [122]. The letter is so short that most of it is duplicated below, albeit in separate passages. As the following excerpts will show, Strachey expressed the acquaintance he had had with Alan Turing, but also, indirectly, made clear that he had *not* read Turing's 1936 paper.

Strachey started his letter with mentioning the unsolvability of Turing's Halting Problem:

> "A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run." [120]

It is not entirely clear what Strachey meant with a 'program' and a 'closed loop'. Let us first unrealistically assume that Strachey only wanted to consider 'programs' that could run on a pre-defined programmable computing machine. In this case, the previous passage would be definitely incorrect, since the Halting Problem is trivially decidable if only programs are considered that can fit into a pre-defined finite machine. Therefore, let us then assume that Strachey associated a 'program' with what we today call a Turing Machine or something equivalent. Then, the previous passage is still, strictly speaking, incorrect: a program (i.e. a Turing Machine) can go on forever by *either* running in a 'closed loop' (forever), *or* by using an unbounded amount of memory space[5].

Disregarding the inaccuracy of Strachey's introduction, we continue with his letter:

> <continued> "I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL[6], but not in any essential way." [120]

The first sentence in the previous quote strongly suggests that Strachey had not read Turing's 1936 paper. On the other hand, Strachey's ability to reproduce the essentials of the proof[7] is equally noteworthy. To do so, he introduced $T[R]$

[5] A more accurate wording would have been: "A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or not." Dijkstra and Minsky, just to name two people, *did* present more accurate wordings of the unsolvability of Turing's Halting Problem; cf. Section 3.3.2.

[6] CPL is an abbreviation for Combined Programming Language, invented by Strachey.

[7] By implicitly relying on the Church-Turing thesis; details omitted here.

as a Boolean function taking a routine (or program) $R$ with no formal or free variables as its argument such that, for any $R$:

- $T[R] = True$ if $R$ terminates if run, and
- $T[R] = False$ if $R$ does not terminate if run.

He then asked the reader to consider the routine $P$ which he defined in CPL as:

```
rec routine P
  ¢L: if T[P] to to L
    Return ¢
```

Finally, he noted that, if $T[P] = True$, then the routine $P$ will loop, and it will only terminate if $T[P] = False$. Therefore, in each case, $T[P]$ has exactly the wrong value and this contradicts, in turn, the existence of the function $T$. In short, Strachey had, presumably on his own, essentially re-invented Turing's diagonal argument.

## 3.2 Indirect Reception of Logic

Even though the connection between Turing's theory of computation and programmable computing machines was not well understood, logic in general did indirectly influence various early developments in computer science.

### 3.2.1 Bauer

Around the turn of the year 1950/51, Bauer had made a wiring diagram for a "logic calculator", called `STANISLAUS`, which he built by 1956. The calculator, containing a hardware stack, could directly evaluate a propositional formula, given a truth value for each variable in the formula. In 1955, inspired by `STANISLAUS`' stack, Samelson and Bauer invented their stack-based translation technique [6], published in 1959 in German and in 1960 in English, which Dijkstra subsequently generalized (cf. Section 2.2.2). The bottom line of this small exposition is that propositional *logic* was Bauer's first application domain, not numerical analysis, and it indirectly led to his invention of the stack, which, in turn, would have a strong impact on international compiler design in subsequent years [8, 9, p.30-32, p.41].

### 3.2.2 Wang

As confirmed by Minsky [86, p.200], Hao Wang was the first researcher to explicitly connect, in his 1957 publication [124], Turing's 1936 theoretical work with that of the computer practitioners of the 1950s. As an aside, note that this was one year *after* Aiken had made his "preposterous" statement (according to Davis, cf. Chapter 1). To conclude, Wang made a clear connection between Turing's theory of computation and programming-language design, thereby illustrating

that Turing's paper and logic in general, did influence computer science to some extent[8].

### 3.2.3 Newell, Shaw, and Simon

As stated in Section 2.3, McCarthy with his `LISP` language was greatly influenced[9] by the list processing language IPL of Newell, Shaw, and Simon [92, 93]. By 1957, these three men had implemented a system for automatic theorem proving. 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 [93, p.218,219]. IPL was the programming language used to implement the system LT [92, 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 [92, p.230,231,235].

#### Flexibility of Memory Assignments

IPL was a very flexible programming language. A user could express the creation of a list during the course of computation[10]. 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[11] [92, p.231].

#### Flexibility in the Specification of Processes

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

[8] It is interesting to note that Wang did not think that Turing's theory of computation had influenced much the actual construction of computing machines [124, p.63]. Minsky, however, advocated for a thorough historical investigation of the matter.

> Minsky: "While it is often said that the 1936 paper did not really much affect the *practical* development of the computer, I could not agree to this in advance of a careful study of the intellectual history of the matter." [86, p.104]

In fact, decades later, Davis has defended the case that Turing's 1936 paper has influenced (via Von Neumann) the actual construction of computing machines [22, 24]. I do not wish to address this controversial topic any further, except to remind the reader of Zuse's work during World War II (cf. Chapter 1).

[9] At the time of this writing, I have not studied `LISP` in sufficient detail to understand how different it was from IPL besides that it was a high-level programming language.

[10] Also known today as dynamic memory management.

[11] Also known today as aliasing.

references in the instructions or on what was referenced. Of particular interest is that processes could be defined implicitly, e.g. by *recursion* [92, p.231]. In more general terms:

> "[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." [92, p.231]

The previous quote in particular and this section as a whole, shows that Newell and Shaw (and Simon[12]) published ideas in 1957 that were similar to those that Dijkstra published in 1960 and 1963, as described in Section 2.1.

### Flexibility First, Efficiency Second

Just like Dijkstra, Newell and Shaw were focused more on the flexibility of their language IPL than on efficiency issues. 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 [92, p.232]. In other words, they followed a top-down methodology. 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." [92, p.232]

Unlike Samelson, Bauer, and many others (cf. Chapter 2), the prime concern here was the language and the ease of being able to express oneself in that language for the problem at hand.

### Recursion

The problem domain of theorem proving, i.e. mathematical logic, contains many natural examples of recursion. For instance, the definition of a proof in a given logic is typically stated recursively. Likewise, the syntax of a logic is defined recursively, similar to the `BNF` notation described in Appendix A. Therefore, in retrospect not surprisingly, LT had recursion built in as well.

A first form of recursion in LT was in its matching routine, which served the purpose of comparing two logical expressions. The matching routine would traverse recursively down the syntax trees of both expressions and pairwise compare syntactic entities. Details are omitted here; it suffices to note that the authors explicitly mentioned that their matching routine could just as well have

[12] Simon co-authored [93] but did not co-author the other paper [92] that is of interest here.

been implemented iteratively but, in their opinion, this would have been less elegant [92, p.239].

A second, and more fundamental, form of recursion in LT was situated in its executive routine, which governed the whole problem-solving behavior of the automatic theorem prover. In a nutshell, a problem (i.e. a theorem that had to be proved) was decomposed into subproblems (i.e. lemmas), which, in turn, were decomposed further, etc. The corresponding programming technique used was, thus, truly recursive in nature, since there was no guarantee that the complete process would terminate. To obtain program termination, a trivial stop criterion in terms of execution time was added to LT as well [92, p.239].

**Conclusion**

The previous discussion shows that the notion of recursion from mathematical logic heavily influenced the design of the IPL programming language which, in turn, would influence McCarthy and, consequently, the `ALGOL` Effort (cf. Chapter 2).

### 3.2.4 Backus

In 1959, Backus presented a formal notation to describe the syntax of machine-independent programming languages, such as `ALGOL60` [3]. His notation, later called Backus Normal Form, was a major contribution to the `ALGOL` Effort and to programming-language design in general [9, 66]. It was based indirectly on either Chomsky's or Post's work and, hence, on logic:

> "There's a strange confusion here. I swore that the idea for studying syntax came from Emil Post because I had taken a course with Martin Davis at the Lamb Estate [an IBM think tank on the Hudson]. ... So I thought if you want to describe something, just do what Post did. Marin Davis tells me he did not teach the course until long afterward [1960-61 according to Davis's records]. So I don't know how to account for it. I didn't know anything about Chomsky. I was a very ignorant person. [Martin Davis speculates that Richard Goldberg, a Harvard-trained logician and part of the Fortran team, may have discussed Post's or Chomsky's work with Backus.]."

The previous words of Backus also stress that he too was not a mathematical logician. The entire previous quote, including the words in between the square brackets, is from [116, p.17].

The immense success of the Backus Normal Form notation suffices as a single example to conclude that logic did, indeed, have a great impact on computer science, even though Backus, himself, was not well-versed in mathematical logic. Appendix A briefly connects Backus Normal Form notation with recursion.

### 3.2.5 Rice

In Chapter 2, I have distinguished between some of those computer pioneers who were for and some who were against recursive procedures. From Section 3.2.3, it follows that there were also pioneers (Newell, Shaw, and Simon) who were acquainted with logic and who used recursion extensively in their work. In this section, it will follow that the recursive-function theorist, Rice, opposed the use of recursive procedures.

In his two-page letter to the editor of the Communications of the ACM [105], published in September 1960, Rice made an explicit connection between recursion in *a* programming language[13] and recursive function theory (i.e. computability theory). Therefore, similar to Sammet's 1969 remark (cf. Section 3.1), Rice addressed the hype around recursion as well, albeit nine years earlier. Unlike Sammet, however, Rice, having expertise in recursive function theory, addressed the issues head on, as illustrated below[14].

Rice considered efficiency to be more important than mathematical elegance and, therefore, suggested *not* to use recursion if possible:

> "When to recur? Never if you can avoid it. Recursive definitions provide a neat way for mathematicians to define functions, and are very convenient for proving things about functions by mathematical induction. However, they are a poor form in which to specify functions for computation." [105]

Rice also made an analogy: computing a function from a recursive definition is like looking up its value in a serial memory, while computing a function from a closed form or analytical expression is like looking up its value in a random access memory and, hence, far more efficient [105]. So, recursion was not promoted by the recursive-function theoretician Rice[15] for efficiency reasons, while it was promoted by Dijkstra (and others) based on linguistic ideals (cf. Chapter 2).

Rice continued by noting that some form of recursion (not necessarily recursive procedures) was, however, needed in order to be able to compute any computable function. He then addressed the question of how to recur, if recursion was indeed needed for the problem at hand. He suggested to recur iteratively, whenever possible, again due to efficiency reasons [105].

Rice ended his letter by implicitly referring to the linguists, such as Dijkstra:

> "There are circumstances, other than the computing of functions from recursive definitions, in which the problem arises of a subroutine operating simultaneously on more than one level. Usually

[13] Even though he did not mention `ALGOL60` in his letter, he may just as well have referred to it since `LISP` and `ALGOL60` were the only two high-level programming languages that had recursion built in by 1960.

[14] A similar exposition was presented by Rice five years later in a short paper: 'Recursion and Iteration' [106]. In that paper, Rice referred to a paper by Peter [99] which I have yet to read in future work.

[15] Students who study recursive function theory today learn Rice's Theorem [117, p.191], thereby illustrating Rice's expertise in recursive function theory.

> the question is one of producing extremely general and unrestricted program components, which may be interconnected with complete *freedom*. The preceding remarks of course do not apply here." [105, my italics]

In other words, Rice understood the point of view taken by the linguists; in particular, the linguists' appeal for generality, regardless of whether recursive procedures were of practical use or not.

Finally, it is important to note, again, that recursive procedures in `ALGOL60` were not needed in order to be able to compute any computable function. This follows from Kleene's Normal Form Theorem and `ALGOL60`'s facility to express potentially unbounded while loops. Rice seems to have been well aware of this fact.

## 3.3 Dijkstra and the Halting Problem

Like many computer pioneers, Dijkstra too, was not a logician [46, p.346]. After graduating from the Gymnasium Erasmianum in Rotterdam, he obtained a degree in 1956 in mathematics and theoretical physics from the University of Leiden [45, p.88]. In the meantime, starting from 1952, Dijkstra also worked as a programmer at the Mathematisch Centrum in Amsterdam in the Computing Department of Aad van Wijngaarden, an expert in numerical analysis (cf. Section 1.2).

Between 1952 and 1969 Dijkstra was primarily a computer programmer, and between 1970 and 1999 he was mainly occupied with formal mathematics. The two occupations were, of course, strongly related. During the 1960s, Dijkstra had consistently advocated that programming "would and should become a mathematical activity", and, by for instance making a case against the goto statement (cf. Section 3.3.2), he had "(re)arranged the programming task so as to make it better amenable to mathematical treatment" [46, p.346]. But in order to actually apply formal reasoning himself, Dijkstra first had to become convinced by his contemporaries. For instance, Tony Hoare's 1969 article 'An Axiomatic Basis for Computer Programming' [55] showed Dijkstra how a programming-language semantics could be defined in terms of the axioms that were needed to prove program properties [46, p.346].

The purpose of this section is three-fold. First, I will try to roughly estimate when Dijkstra had most definitely become acquainted with Turing's 1936 paper[16]. Second, I will show that Dijkstra's work on high-level programming languages and compilers was positively influenced by his understanding of the unsolvability of Turing's Halting Problem. Third, I will end by very briefly addressing the difficult relationship that Dijkstra had with logicians who worked in computer science.

[16] I have received comments from people who claim that Dijkstra had already understood Turing's work during the 1950s. I do not contradict this claim in this text, but I currently do not have any evidence to support it either.

### 3.3.1 Dijkstra's Reception of Turing's 1936 Paper

While, during the 1950s at the Mathematisch Centrum, Dijkstra was mainly a computer practitioner (and not yet a theoretical computer scientist), he did have the opportunity to become acquainted with Turing's 1936 paper at a relatively early stage in his career. For, in March 1953, the mathematician Tamari gave a presentation, most likely in French, at the Mathematisch Centrum about Turing Machines and Post's word problems [121]. Nothing related to this presentation is, however, mentioned in the 1953 scientific annual report of the Mathematisch Centrum [64], nor any sign of reception of those ideas.

In 1962 at the IFIP congress in Münich, Dijkstra gave an invited talk, titled: 'Some Meditations on Advanced Programming' [28]. In this talk, Dijkstra showed that he was aware of some of the work of Turing and Von Neumann:

> "However, as I told you, the sky above the programmers' world is brightening slowly. Before I go on to draw your attention to some discoveries that are responsible for this improvement, I should like to state as my opinion that it is relatively unimportant whether these are really new discoveries or whether they are rediscoveries of things perfectly well known to people like, say, Turing or von Neumann." [28, p.536]

As we have seen previously, many computer pioneers did not see the connection between Turing's work and practical computer problems. While logicians, such as Wang and Davis, had to make that connection themselves, so did computer practitioners such as Dijkstra:

> <continued> "For in the latter case, the important and new thing is that a greater number of people become aware of such a fact, and that a greater number of people realize that these are not just theoretical considerations but that they may have tangible, practical results." [28, p.536]

Dijkstra was, in 1962, well aware of the message conveyed by Burks in 1961, discussed previously, that hardware and software are (within limits) interchangeable:

> "One important rediscovery is that of the well-known equivalence of designing a machine and making a program." [28, p.536]

While it is not clear whether Dijkstra had, by 1962, become fully acquainted with Turing's 1936 theory of computation; he clearly had done so by 1964. For in April of that year, Dijkstra gave a Dutch presentation [32] in which he talked about the practical (i.e. finite) limitations of electronic computers. In the beginning of his talk, he briefly but explicitly mentioned[17] Turing and his theory of computation, "Turing machines", and unsolvable problems [32, p.55].

[17] Dijkstra did not *refer* to Turing's 1936 paper, however.

### 3.3.2 Dijkstra Applies the Unsolvability of the Halting Problem

In 1962, after his IFIP address in Münich, Dijkstra became Professor of Mathematics at the Eindhoven University of Technology. The 1960s were, in general, the years in which Dijkstra strongly advocated for a mathematical treatment of programming; e.g. by making a case against the goto programming-language construct [46, p.346].

#### Dijkstra's Goto Unraveled

As professor, Dijkstra often received letters from practitioners in the field. Of particular interest is that, during the mid-1960s, two different programming department managers, on their own initiative, contacted Dijkstra to convey the same concern: that, in general, the quality of their programmers was inversely proportional to the density of goto statements in their programs [33, p.9].

Dijkstra reacted accordingly by conducting programming experiments himself. For several `ALGOL60` programs that contained goto statements, he wrote functionally-equivalent programs without using goto statements. While the latter programs were initially more difficult to make, they typically turned out to be "shorter and more lucid" [33, p.9].

In other words, Dijkstra's strive for elegance was leading him the way, once again. But, Dijkstra knew that experiments could, at best, merely be indications of a deeper underlying problem. Therefore, he tried to understand whether and why it was, *in general*, beneficial not to use goto statements. To do so, Dijkstra resorted to his theoretical knowledge of computing, by applying the unsolvability of Turing's Halting Problem[18]:

> "The origin in the increase in clarity is quite understandable. As is well known *there exists no algorithm to decide whether a given program ends or not.* In other words: each programmer who wants to produce a flawless program must at least convince himself by inspection that his program will indeed terminate. In a program, in which unrestricted use of the goto statement has been made this analysis may be very hard on account of the great variety of ways in which the program may fail to stop. After the abolishment of the goto statement there are only two ways in which a program may fail to stop: either by infinite recursion –i.e. through the procedure mechanism– or by the repetition clause. This simplifies the inspection greatly." [33, p.10, my italics]

In short, Dijkstra used the unsolvability of the Halting Problem to infer a criterion of elegance in higher-level programming. In retrospect, this may have been an unprecedented application of the Halting Problem. Dijkstra did not,

[18] The italicized words describe the Halting Problem by implicitly relying on the Church-Turing thesis.

however, use the previous explanation in his official 1968 publication 'Go To Statement Considered Harmful' [34]. In fact, he did not mention the Halting Problem at all.

The Halting-Problem argument in the previous quote explains that, due to the "great variety of ways in which the program may fail to stop", the goto clause is inferior to the procedure mechanism and the repetition clause. Dijkstra's letter 'Go To Statement Considered Harmful' conveys the same message but implicitly. The thesis of Dijkstra's letter is that every programming-language clause "should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way". By first showing how this can be accomplished by various clauses, including the procedure mechanism and the repetition clause, Dijkstra then explained why goto clauses make it "terribly hard to find a meaningful set of coordinates in which to describe the process progress" [34].

Dijkstra was not the first person to make a case against the goto statement. Naur already did so in 1963 [89], and Van Wijngaarden had illustrated in 1964 how to get rid of gotos to labels by replacing them with calls to procedures generated from the corresponding labels [13, 72, p.27,-]. Finally, Knuth also mentions that Schorre and Forsythe were pioneers in trying to avoid using goto statements in their programs [67, p.264]. However, Dijkstra may well have been the first to motivate a case against the goto by relying on the unsolvability of Turing's Halting Problem.

### Correct-by-Construction Programming

In 1967, Marvin Minsky published his book *Computation: Finite and Infinite Machines* [86], in which he explained the concept of 'effective procedure' by introducing the work of the logicians Turing, Post and others to the non-logician. Dijkstra had read the book between 1967 and 1971 because he explicitly referred to it in his 1971 lecture notes *A Short Introduction to the Art of Programming* [35]. Dijkstra's reference to Minsky's book was made in Chapter 2 of his lecture notes where he re-applied Turing's diagonal argument to prove the unsolvability of the Halting Problem. Dijkstra's application of the diagonal argument, however, was not conducted with Turing Machines but in the high-level programming language `ALGOL60`, and was, hence, similar to Strachey's proof[19] discussed in Section 3.1.3.

In writing Chapter 2 of *A Short Introduction to the Art of Programming* [35], Dijkstra first distinguished between a *proper* algorithm, which halts on all inputs, and an *improper* algorithm, which does not halt on all inputs. As the names suggest, Dijkstra only considered algorithms that halt on all inputs to

[19] Minsky's proof of the unsolvability of the Halting Problem was conducted at the low level of Turing Machines, not at the higher level of `ALGOL60` programs (see Section 8.2, page 148 in [86]). Minsky did show in his book the computational equivalence between Turing Machines and various other systems, including "Universal Program Machines with Two Registers" on page 255, but he did not prove the unsolvability of the Halting Problem directly in any of those computationally equivalent systems.

be practically relevant[20]. Afterwards, he explained that it is not possible to algorithmically distinguish between a proper and an improper algorithm, by explicitly referring to the unsolvability of Turing's Halting Problem [35, p.15].

Dijkstra subsequently proved the unsolvability of the Halting Problem, as mentioned previously, and then interpreted it in his own original way:

> "The moral of this story is that it is an intrinsic part of the duty of everyone who professes to compose algorithms to supply a proof that his text indeed represents a proper algorithm." [35, p.16]

The originality of this statement follows by contrasting it with Minsky's interpretation:

> "[...] it is impossible to devise a uniform procedure, or computer program, which can look at any computer program and decide whether or not that program will ever terminate. *This means that computation scientists cannot aspire to evolve a completely foolproof 'debugging' program.*" [86, p.153, my italics]

Both Dijkstra and Minsky only considered programs that halt on every input to be practically relevant. But, while Minsky stressed a negative implication of the unsolvability of the Halting Problem, Dijkstra provided a more positive conclusion. Minsky stated that it is not possible to completely automate a posteriori verification of an arbitrary program. Dijkstra, on the other hand, aware of this negative result, used it to motivate that programming and correctness proving should go hand in hand, also known as correct-by-construction programming. According to Dijkstra, the programmer should restrict his programs so much that he can prove that they halt. Dijkstra's approach would become one of the cornerstones of the field of activity called programming methodology [36, p.6].

### Dijkstra's Advice to Industry

On Friday, March 3, 1978, Dijkstra wrote an open letter [41] to a Lt. Col. William A. Whitaker, to comment on the design documentation of Ironman[21], that he had received a few days earlier, presumably from Whitaker himself. In his letter, Dijkstra explained the practical infeasibility of Ironman's requirement which stated that:

> "There shall be no language restrictions that are not enforceable by translators." [41, p.0]

To do so, Dijkstra again used the unsolvability of the Halting Problem in a non-trivial manner (see Appendix B).

[20] Contrary to the computer practitioner, the recursive-function theorist (i.e. mathematical logician) was primarily interested in algorithms that did not halt on every input.

[21] Ironman was the next version of the requirement specifications for the high-level programming language Ada [43, p.3].

Similar to his lecture notes *A Short Introduction to the Art of Programming* [35], discussed in the previous section, Dijkstra stressed the responsibility of the user (i.e. programmer) and, hence, conveyed the infeasibility of completely automating a translator (i.e. compiler) that would respect all of the requirements stated in Ironman's documentation:

> "[W]e need to distinguish between the notion of 'a legal program' and the notion of 'a correct program'. From translators we can require that they reject illegal programs; for legal programs the language definition should define the *proof obligations to be met [by the user]* in order to make the legal program also a correct program."
> [41, p.2, my italics]

In summary, Dijkstra applied the unsolvability of the Halting Problem to illustrate the infeasibility of one of Ironman's requirements. Based on Dijkstra's first words to the colonel:

> "[T]his letter is an almost immediate reaction to the 4 kg. of language design documentation that reached me last Monday; it was written because –at least in those few days– *I failed to discover an adequate treatment of an issue that now seems to be in urgent need of clarification.*" [41, p.0, my italics]

Dijkstra may, perhaps yet again, have provided a novel application of the unsolvability of the Halting Problem in the fields of higher-level programming-language design and compiler building.

### 3.3.3 Dijkstra vs. Logic

From 1970 and onwards, Dijkstra became more and more occupied with formal reasoning. And in the mid-1970s, he realized that many researchers were sceptical about axiomatic-based systems, and, hence about the kind of work he himself was conducting:

> "[. . .] over the past years I have discovered that many people are very suspicious about either the legitimacy or the adequacy of axiomatic methods. And recently I came to the conclusion that I could not understand why." [39, p.0]

Those people included of course the logicians whose suspicion was based on Gödel's Incompleteness Theorem. Dijkstra wanted to understand whether this suspicion was justified in his domain of computer programming:

> "It begins with Gödel's Theorem, and here I start already with a display of ignorance. [. . .] That I did not study [Kleene's 'Introduction to Metamathematics'], but only read (parts of) it, was caused by the fact that the reading of it did not give me the feeling that its

> contents really concerned me. [...] Is it, because Gödel's Theorem denied the possibility of a form of perfect understanding that had never been my ideal in the first place?" [39, p.0]

The above quote was written by Dijkstra some time[22] between 1973 and 1975. In a letter, written in 1974, Dijkstra acknowledged Hoare's 1969 article 'An Axiomatic Basis for Computer Programming' [55] as a source of inspiration for him to start working on formal methods, but at the same time he again expressed concerns:

> "[Tony Hoare's] article attracted me in the sense that it tied in with the syntactical structure of the program text, but the separation in axioms and rules of inference –a logical tradition, but not my tradition– worried me." [37, p.2]
>
> "The only thing the traditional logicians did was to try find a model for the real world, but, since in the form of computing science[,] logic has also become an engineering activity, I prefer the real world to provide a model for my dreams..." [37, p.10]

In short, while Dijkstra was, on the one hand, inspired by some of the ideas from mathematical logic, as for instance some of those presented in Hoare's paper [55], he was also careful only to use those ideas which he was able to justify for his own cause, namely that of computer programming.

Further research is required to understand Dijkstra's perspective on logic for computer science. It is already clear, however, that Dijkstra had a troubled relationship with the logicians: Broy has written two pages on this particular topic in [13, p.90-91] and Apt has mentioned in [2] the devastating review[23] of Dijkstra's 1990 book *Predicate Calculus and Program Semantics*, co-authored by Scholten.

[22] Dijkstra did not put a date on all of his `EWD`s. But the Dijkstra Archive at the University of Texas [25] does provide an approximated date for each `EWD`.

[23] Egon Börger in Science of Computer Programming 23, pp. 1-11, 1994.

# Chapter 4

# Conclusions & Future Work

Two themes have been addressed in my thesis: the Advent of Recursion and the Advent of Logic in computer science. Conclusions and future work are presented for each theme, followed by other future work which I hope to pursue as well.

## 4.1 Recursion

The recursive procedure, as a particular but important example of recursion, entered the arena of programming languages and, hence, computer science in several ways by different people. Particular attention has been paid to Edsger W. Dijkstra, who was one of the first to implement recursive procedures by building an `ALGOL60` compiler and, who, in later years, strongly advocated for recursive procedures while many of his contemporaries opposed them.

Dijkstra, along with some of his colleagues, were primarily led by linguistic ideals and did not necessarily see the direct practical applicability of the recursive procedure. Furthermore, it seems that Dijkstra's sole purpose was to pursue simplicity in language design and subsequent compiler building. As a by-product, he obtained a language that could express recursive procedures and a simple compilation technique for that language. Other proponents of recursive procedures were Newell, Simon, and Shaw. They *did* introduce and subsequently use recursive procedures for practical programming problems. Nevertheless, the end product of Dijkstra's efforts and those of Newell, Simon, and Shaw were very similar: a flexible programming language, i.e. a language implemented by means of what we would today call *dynamic memory management*.

Indeed, in technical terms, the advent of the recursive procedure implied the advent of dynamic memory management, cf. a run-time stack. Many opposed the recursive procedure, because, in their opinion, the run-time management was too expensive in terms of execution time –a claim that was not contradicted by Dijkstra. For instance, Samelson and Bauer preferred to statically allocate all procedures (i.e. prior to execution), in the name of efficiency. Unfortunately for them, their implementation choice led to a dramatic usage of

memory space: all their procedures had to be allocated statically, while Dijkstra's method, essentially, only allocated space when a procedure was activated at run time. In addition, Samelson and Bauer's implementation choice forced them to restrict the `ALGOL60` language by disallowing procedures to call other procedures and, hence, recursive procedures in particular. Presumably, another reason why Samelson and Bauer did not implement recursion was because they simply did not see why it would be useful in practice.

### Bottom-Up vs. Top-Down

The previous paragraph illustrates that Samelson and Bauer followed what we today would call a bottom-up approach: their implementation choice influenced the design of the language. Dijkstra, Newell, Simon, and Shaw, on the other hand, pursued a top-down approach by first solely studying the language, prior to incorporating machine features into their study.

The reason why Dijkstra followed a top-down approach may, as hinted in [1, p.124,125], lie in the fact that he did not have a programmable computing machine during the early 1950s, while many of those who worked bottom-up, did. After some further reflection, however, this reasoning is probably too simplistic. For instance, Newell, Shaw, and Simon may well have had a programmable computing machine at their disposal during the early 1950s[1] and they did work top-down.

### Dijkstra's Extreme Stance

While Dijkstra had made a great contribution to language design and compiler building by the end of 1960, his ideology was considered to be too impractical according to many participants of the 1962 Rome symposium. Dijkstra's ideology led him, for instance, to the extreme of omitting all type indications and, hence, transferring all the type checking to the run-time system, which, as many observed, would have a negative effect on computation time –an observation that Dijkstra did not contradict. In fact, Dijkstra believed that efficiency problems would be resolved, or at least become negligible, in the long term. 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 like Samelson, Bauer, Wilkes, and Strachey.

Nevertheless, with such an ideology in mind, Dijkstra was perceived as someone who totally neglected efficiency issues. Hence, it is no surprise that Dijkstra and his fellow 'linguists' were the laughing stock of Seegmüller's well-received comment at the Rome symposium. A closer look at Dijkstra's ideology, however, shows that his agenda was not to neglect efficiency issues per se, but to

[1] A statement that needs to be checked in future work. In retrospect, however, Newell, Simon, and Shaw were not as linguistically inclined as Dijkstra. Recall that they pursued a flexible language in order to implement their theorem prover, not because they were interested in language per se.

avoid as many language restrictions as possible, in the interest of being able to construct texts that were easy to validate in terms of correctness.

### Many "Firsts"

While Dijkstra was clearly not the sole inventor of a technique to implement recursive procedures, it seems that Dijkstra's approach was more generally applicable or simpler than many others. Why else, would Dijkstra's 1960 paper [26] stand out during the 1960s, compared to the other approaches? Further research is required to understand how Dijkstra's notations, as illustrated in [27], simplified the task of compiler building. Also some other papers of Dijkstra require further investigation in this regard, namely [30, 31].

### Logic in General

While there were both proponents and opponents of the recursive procedure, it is also interesting to zoom in on those pioneers who were well-versed in logic at the time the recursive procedure was introduced. Dijkstra, Samelson, and Bauer, for instance, did not belong to this select group[2]. As we have seen, the recursive procedure was promoted by Newell, Shaw, and Simon who were well-acquainted with logic, but it was also eschewed by the recursive-function theorist Rice.

### Dijkstra Discards his own Invention

To conclude the discussion on recursive procedures, it is fascinating to note that in October 1974, by the time the recursive procedure had presumably become generally accepted, Dijkstra casted doubts upon it [38].

> Dijkstra, March, 1975: "The other day I heard that by casting doubts on the central role of recursion, I had caused commotion at some places [. . .]" [40]
>
> Dijkstra, April, 1975: "The discovery that doubts may have to be casted [sic] upon recursion –in Computing Science for more than a decade the hallmark of academic respectability!– was something of a shock for me. [. . .]". [36, p.9]

Understanding Dijkstra's line of thought, undoubtedly, requires further research. Clearly, he was a man who wanted to understand the fundamentals of his discipline and, therefore, was willing to discard his own research contributions.

[2] If Samelson and Bauer had grasped Rice's 1960 paper and applied Kleene's Normal Form theorem, then they would have had a strong argument not to implement recursive procedures for the `ALGOL60` language.

## 4.2 Logic and Turing's Halting Problem

As stressed by the historian Michael Mahoney, different 'communities of computing' had their own views towards what could be accomplished with a programmable computing machine. The mathematical logicians belonged to one such community, while Howard Aiken, for instance, belonged to another.

As I have shown, many computer pioneers belonged to a computing community in which Turing's 1936 paper was not read. In fact, even those who did read Turing's paper, including logicians, did not initially grasp the connection between the paper and programmable computing machines. Therefore, Martin Davis' negative characterization of pioneers, such as Howard Aiken and Herman Goldstine, and Andrew Hodges' positive characterization of Alan Turing, are misleading.

During the late 1970s, many computer pioneers of the 1950s and 1960s took the effort to stress that they were *not* mathematical logicians, and that their pioneering work was either not based on logic, or, if it was, that they did not completely understand how. Nevertheless, even though the connection between Turing's theory of computation and programmable computing machines was not well understood, logic in general did indirectly influence various early developments in computer science; most notably, Backus' notation and Newell, Shaw, and Simon's IPL language.

Finally, I speculate that, unlike many of his contemporaries, Dijkstra was able to apply the unsolvability of Turing's Halting Problem to progress the agenda of his research community. Three historical events support this claim. First, in the mid-1960s, Dijkstra's understanding of the unsolvability of the Halting Problem led him to write his now-famous 1968 letter 'Goto Considered Harmful' even though he did not mention it in his letter. Second, in his 1971 lecture notes, *A Short Introduction to the Art of Programming*, Dijkstra provided a constructive and alternative interpretation of the unsolvability of the Halting Problem by advocating for correct-by-construction programming. Third, in a 1978 letter, he applied the unsolvability of the Halting Problem in a non-trivial manner to prove the practical infeasibility of a programming-language design requirement which he had obtained from industry. In future work, I hope to further investigate to what extent Dijkstra was and was not influenced by mathematical logic.

## 4.3 More Future Work

In addition to the above, I hope to continue this historical line of research in several ways. First, Chomsky's influence on the computer pioneers of the 1950s deserves investigation[3]. Second, Dijkstra's contributions in later years are of particular interest to me and many other people. In fact, while writing this last chapter, I became aware of two more secondary sources: a thesis written by G. van den Hove [59] in which he covers Dijkstra's scientific contributions between

[3]Naturally, I will first have to acquaint myself with secondary sources.

the years 1951 and 1968, and a Ph.D. thesis of P.M. Priestley [102] in which logic is covered extensively with respect to the development of programming languages. Unfortunately, I have not yet been able to study these two sources. Third, I am also eager to study various developments in Complexity Theory between 1936 and 1971, in an attempt to understand how those developments were and were not related with the `ALGOL` Effort. Finally, if the reader shares some of my interests, I would be most grateful to hear from him or her.

# Bibliography


[1] G. Alberts, H.T. de Beer, 'De AERA. Gedroomde machines en de praktijk van het rekenwerk aan het Mathematisch Centrum te Amsterdam', Studium 2, pp. 101-127, 2008.

[2] K.R. Apt, "Obituary. Edsger W. Dijkstra (1930-2002): A Portrait of a Genius", Formal Aspects of Computing 14:92-98, 2002.

[3] J.W. Backus, 'The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM Conference', in: IFIP Congress, pp.125-132, 1959.

[4] J.W. Backus et al, Edited by P. Naur, 'Report on the algorithmic language `ALGOL60`', Commun. ACM 3:5, pp. 299-314, 1960.

[5] J.W. Backus, 'The History of `FORTRAN` I, II, and III' and the corresponding transcripts of: presentation, discussant's remarks, question and answer session, in *History of Programming Languages*, Richard L. Wexelblat, (Ed.), New York: Academic Press, pp. 25-70 , 1981.

[6] F.L. Bauer, K. Samelson, 'Sequentielle Formelübersetzung'. Elektronische Rechenanlagen 1, 176-182, 1959; publ. in English as: 'Sequential formula translation', Comm. Ass. Comp. Mach. 3, 76-83, 1960.

[7] F.L. Bauer, 'My years with Rutishauser', LATSIS Symposium ETH Zürich, Feb. 2002.

[8] F.L. Bauer, 'From the Stack Principle to `ALGOL`', in: Manfred Broy and Ernst Denert, editors, *Software pioneers: contributions to software engineering*, Berlin: Springer, pp. 26-42, 2002.

[9] H.T. de Beer, *The History of the* `ALGOL` *Effort*, Masters Thesis, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, August 2006.

[10] C. Böhm, 'Calculatrices digitales: Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme [Digital computers: On the deciphering of logical-mathematical formulae by the machine itself during the conception of the program]', Annali di Matematica Pura ed Applicata (4) 37, pp. 175-217, 1954.

[11] A. van den Bogaard, 'Stijlen van programmeren 1952-1972', Studium 2, pp. 128-144, 2008.

[12] G.S. Boolos, J.P. Burgess, R.C. Jeffrey, *Computability and Logic*, fifth edition, Cambridge University Press, 2007.

[13] L. Böszörményi, S. Podlipnig, *People Behind Informatics*, Institute of Information Technology, University of Klagenfurt, 2003. With contributions from M. Broy, T. Dahl, and M. Nygaard.

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

[15] B.E. Carpenter, R.W. Doran, 'The Other Turing Machine', Computer Journal, Vol. 20, pp. 269-279, 1977.

[16] R. Cartwright, J.L. McCarthy, 'Recursive programs as functions in a first order theory', Mathematical Studies of Information Processing, pp.576-629, 1978.

[17] T. Cheatham, '`ALGOL` session', History of Programming Languages, New York, NY: ACM Press, pp. 171, 1978.

[18] N. Chomsky, *Syntactic Structures*, Walter de Gruyter GmbH & Co. KG, 10785 Berlin, 1957.

[19] *Computer Programming and Formal Systems,* edited by P. Braffort and D. Hirschberg as a Special Issue 1963 in *Studies in Logic and the Foundations of Mathematics*, editors L.E.J. Brouwer, E.W. Beth and A. Heyting, North-Holland Publishing Company Amsterdam.

[20] B.J. Copeland, 'Unfair to Aiken', IEEE Annals of the History of Computing, pp. 33-36, 2004.

[21] M. Davis, *Computability & Unsolvability*, McGraw-Hill, New York 1958; reprinted with an additional appendix, Dover 1983.

[22] M. Davis, 'Mathematical Logic and the Origin of Modern Computers', Studies in the History of Mathematics. Mathematical Association of America, 1987, pages 137-165. Reprinted in: *The Universal Turing Machine - A Half-Century Survey*, Rolf Herken, editor. Verlag Kemmerer & Unverzagt, Hamburg, Berlin 1988; Oxford University Press, pp. 149-174, 1988.

[23] M. Davis, 'From Logic to Computer Science and Back' Chapter 3 (pages 53-85) in *People and Ideas in Theoretical Computer Science*, edited by C.S. Calude, Springer 1999.

[24] M. Davis, *Engines of Logic: Mathematicians and the origins of the Computer* (1st ed.), New York NY: W.W. Norton & Company, ISBN 0-393-32229-7 (pb.), 2000.

[25] E.W. Dijkstra Archive (University of Texas), *http://ww.cs.utexas.edu/~EWD/welcome.html*

[26] E.W. Dijkstra, 'Recursive Programming', Numerische Mathematik 2, pp. 312-318, 1960.

[27] E.W. Dijkstra, 'Unifying Concepts of Serial Program Execution', in Proceedings of the Symposium Symbolic Languages in Data Processing, pp. 236-251, Rome, March 26-31, 1962.

[28] E.W. Dijkstra, 'Some Meditations on Advanced Programming', IFIP Congress, pp. 535-538, Münich, Germany, 1962.

[29] E.W. Dijkstra, 'On the Design of Machine Independent Programming Languages', in Annual Review in: Goodman, Richard, editor, Automatic Programming 3, pp. 27-42, 1963.

[30] E.W. Dijkstra, 'An `ALGOL60` Translator for the X1', in: Goodman, Richard, editor, Annual review in automatic programming 3, pp. 329-345, 1963.

[31] E.W. Dijkstra, 'Making a Translator for `ALGOL60`', in: Goodman, Richard, editor, Annual review in automatic programming 3, pp. 347-356, 1963.

[32] E.W. Dijkstra, 'Over de beperkte omvang van ons rekentuig', pp. 55-71 in *NRMG '59/'64*, Uitgave ter gelegenheid van het eerste lustrum van het Nederlands Rekenmachine Genootschap (NRMG), 2e Boerhaavestraat 49, Amsterdam. Voordrachten gehouden op 3 April, 1964.

[33] E.W. Dijkstra, `EWD`117, 'Programming Considered as a Human Activity', approximation: 1964-1967.

[34] E.W. Dijkstra, 'Go To Statement Considered Harmful', Letters to the Editor, Communications of the ACM, Vol. 11 (2), pp. 147-148, 1968.

[35] E.W. Dijkstra, *A Short Introduction to the Art of Programming*, August 1971. Also available as `EWD`316.

[36] E.W. Dijkstra, `EWD`450, 'Correctness concerns and, among other things, why they are resented', Invited paper, to be presented at the 1975 International Conference on reliable Software, 21-23 April 1975, Los Angeles, California, USA.

[37] E.W. Dijkstra, `EWD`454, 'A letter to Dr. Bekic', 8th October, 1974.

[38] E.W. Dijkstra, `EWD`456, 'Determinism and recursion versus nondeterminism and the transitive closure', 14th October, 1974.

[39] E.W. Dijkstra, `EWD`463, 'Some questions', approximation: 1973-1975.

[40] E.W. Dijkstra, EWD487, 'Letter to the Burroughs Recipients of the EWD-series', 20 March, 1975.

[41] E.W. Dijkstra, EWD658, 'On language constraints enforceable by translators', An open letter to Lt.Col. William A. Whitaker, 3rd of March, 1978.

[42] E.W. Dijkstra, EWD682, 'The Nature of Computer Science (first draft)', approximation: 1976-1979.

[43] E.W. Dijkstra, EWD1287, 'Dear Tony, dearest Jill, and other people, dear or not', approximation: 1995-2000.

[44] E.W. Dijkstra, EWD1024, Talk held at the Anniversary Celebration of the Department of Computing Science, ETH Zürich, 19 October, 1988.

[45] E.W. Dijkstra, 'EWD1166: From My Life' Chapter 4 (pages 86-92) in *People and Ideas in Theoretical Computer Science*, edited by C.S. Calude, Springer 1999.

[46] E.W. Dijkstra, 'EWD1308: What led to "Notes on Structured Programming"', in: Manfred Broy and Ernst Denert, editors, *Software pioneers: contributions to software engineering*, Berlin: Springer, pp. 341-346, 2002.

[47] R.W. Floyd, 'Assigning meanings to programs', Proc. Amer. Math. Soc. Symposia in Applied Mathematics, Vol. 19, pp. 19-31.

[48] S. Ginsburg, H.G. Rice, 'Two Families of Languages Related to ALGOL', J. ACM 9:3, 1962.

[49] A.A. Grau, 'Recursive processes and ALGOL translation', Commun. ACM 4:1, pp. 10-15, 1961.

[50] A.A. Grau, U. Hill, H. Langmaack, *Handbook for Automatic Computation Volume 1 Part b Translation of ALGOL 60,* Springer-Verlag Berlin Heidelberg, 1967.

[51] J.Y. Halpern, R. Harper, N. Immerman, P.G. Kolaitis, M.Y.Vardi, V. Vianu, 'On the unnusual effectiveness of logic in computer science', Bulletin of Symbolic Logic 7(2), pp. 213-236, 2001.

[52] D. Harel, 'On Folk Theorems', Comm. ACM, Vol. 23, Nr. 7, pp. 379-388, 1980.

[53] S. Henriksson, 'A brief history of the stack', SHOT, 2009.

[54] C.A.R. Hoare, 'Algorithm 64: Quicksort', Comm. ACM, Vol. 4, Issue 7, p. 321, July 1961.

[55] C.A.R. Hoare, 'An Axiomatic Basis for Computer Programming', Communic. ACM 12(10): 576-580, 1969.

[56] C.A.R. Hoare, D.C.S. Allison, 'Incomputability', ACM Comput. Surv. 4(3), pp. 169-178, 1972.

[57] C.A.R. Hoare, 'The Emperor's Old Clothes', Commun. ACM 24(2), pp. 75-83, 1981.

[58] G.M. Hopper, Keynote Address of the Opening Session and the corresponding transcript of question and answer session, in *History of Programming Languages*, Richard L. Wexelblat, (Ed.), New York: Academic Press, 7-22, 1981.

[59] G. van den Hove, *Edsger Wybe Dijkstra: First Years in the Computing Science (1951-1968)*, Masters thesis, University of Namur, Academic year 2008-2009.

[60] P.Z. Ingerman, 'Thunks: a way of compiling procedure statements with some comments on procedure declarations', Commun. ACM 4:1, pp. 55-58, 1961.

[61] E.T. Irons, W. Feurzeig, 'Comments on the Implementation of Recursive Procedures and Blocks in ALGOL-60', `ALGOL` Bull. Sup 13.2, pp. 1-15, 1960.

[62] E.T. Irons, 'A syntax directed compiler for `ALGOL60`', Commun. ACM 4:1, pp. 51-55, 1961.

[63] IBM, 'Preliminary Report `FORTRAN`'.

[64] Jaarverslag 1953, Mathematisch Centrum in Amsterdam, 2e Boerhaavestraat 49, Amsterdam.

[65] S.C. Kleene, 'General recursive functions of natural numbers', Math. Annalen 112, pp. 727-742, 1936.

[66] D.E. Knuth, 'Backus Normal Form vs. Backus Naur Form', Commun. ACM 7:12, pp. 735-736, 1964.

[67] D.E. Knuth, 'Structured Programming with go to Statements', Computing Surveys, Vol. 6, No. 4, December, 1974.

[68] D.E. Knuth, 'The Early Development of Programming Languages', Encyclopedia of Computer Science and Technology 7 (New York: Marcel Dekker, Inc., 1977), pp. 419-493. Reprinted in *Selected Papers on Computer Languages*, by Donald E. Knuth, 2003, Center for the Study of Languages and Information Leland Stanford Junior University.

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

[70] J. Krige, *American Hegemony and the Postwar Reconstruction of Science in Europe,* Cambridge, MA: MIT Press 2006.

[71] F.E.J. Kruseman Aretz, *The Dijkstra-Zonneveld* `ALGOL60` *compiler for the Electrologica X1 (historical note SEN, 2)*, SEN-N0301, ISSN 1386-369X, 2003.

[72] J.A.N. Lee, 'Computer pioneers', IEEE Computer Society Press, Los Alamitos, 1995.

[73] J.A.N. Lee, ' "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, Vol. 18, No. 2, pp. 54-62, 1996.

[74] P. Lucas, 'The Structure of Formula-Translators', `ALGOL` Bull. Sup 16, pp. 1-27, 1961.

[75] M.S. Mahoney, 'The Roots of Software Engineering', An expanded version of a lecture presented at CWI on February 1990. CWI Quarterly 3,4, pp. 325-334, 1990.

[76] M.S. Mahoney, 'What Makes History?', Appendix A in *History of Programming Languages*, ed. Thomas J. Bergin, Jr., and Richard B. Gibson, Jr. (NY: ACM Press), pp. 831-832, 1996.

[77] M.S. Mahoney, 'The histories of computing(s)', Interdisciplinary Science Reviews, Vol. 30, No. 2, pp. 119-135, 2005.

[78] M.S. Mahoney, 'Software as Science - Science as Software', Preprint version of article published in Hashagen, Keil-Slawik, and Norberg (eds.), History of Computing: Software Issues, Berlin, Springer Verlag, 2002.

[79] J. McCarthy, 'The Inversion of Functions Defined by Turing Machines', Automata Studies, edited by C. Shannon and J. McCarthy, Princeton University Press, 1956.

[80] J. McCarthy, 'On Conditional Expressions and Recursive Functions (Letter)', Comm. ACM 2(8): 2-3, Aug. 1959.

[81] J. McCarthy, 'Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I', Commun. ACM 3(4): pp.184-195, 1960.

[82] J. McCarthy, 'A Basis for a Mathematical Theory of Computation', Computer Programming and Formal Systems, edited by P. Braffort and D. Hirshberg, published by North-Holland, 1963. An earlier version was published in 1961 in the Proceedings of the Western Joint Computer Conference.

[83] J. McCarthy, 'History of `LISP`' and the transcripts of: presentation, discussant's remark, question and answer session, in *History of Programming Languages*, Richard L. Wexelblat, (Ed.), New York: Academic Press, 173-195, 1981.

[84] Merriam-Webster's Online Dictionary, *www.merriam-webster.com/dictionary*

[85] N. Metropolis, J. Howlett, G-C. Rota, *A History of Computing in the Twentieth Century*, New York: Academic Press, Inc. 1980.

[86] M. Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, Inc. 1967.

[87] L. De Mol, 'How to talk with a computer? An essay on Computability and Man-Computer conversations.', Off Topic: Ubersetzen. Zeitschrift für Medienkunst der KHM, 2008.

[88] L. De Mol, 'Doing Mathematics on the ENIAC. Von Neumann's and Lehmer's different visions.', Mathematical Practice and Development throughout History. Proceedings of the 18th Noverbertagung on the History, Philosophy and Didactics of Mathematics, 2009.

[89] P. Naur, 'Go to statements and good Algol style', BIT 3, 3, pp. 204-208, 1963.

[90] P. Naur, 'Proof of Algorithms by General Snapshots', BIT 6, pp. 310-316, 1966.

[91] P. Naur, 'The European side of the last phase of the development of `ALGOL60`', in *History of Programming Languages*, Richard L. Wexelblat, (Ed.), New York: Academic Press, 92-138, 147-170, 1981.

[92] A. Newell, J.C. Shaw, 'Programming the logic theory machine', Proc. Western Joint Computer Conf., pp. 230-240, 1957.

[93] A. Newell, J.C. Shaw, H.A. Simon, 'Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristic', Proc. Western Joint Computer Conf., pp. 218-230, 1957.

[94] P. O'Hearn, R. Tennent, *Algol-like Languages (Progress in Theoretical Computer Science)*, Birkhäuser Boston, December, 1996, ISBN: 0817639373.

[95] C.-H.L. Ong, 'Observational Equivalence of 3rd-Order Idealized Algol is Decidable', LICS, 17th Annual IEEE Symposium on Logic in Computer Science, pp. 245, 2002.

[96] Panel Discussion: 'Languages for Aiding Compiler Writing', p.187-204 in Proceedings of the Symposium Symbolic Languages in Data Processing, Rome, March 26-31, 1962.

[97] Panel Discussion: 'Efficient Processor Construction', p.363-381 in Proceedings of the Symposium Symbolic Languages in Data Processing, Rome, March 26-31, 1962.

[98] Panel Discussion: 'Reflections from Processor Implementors on the Design of Languages', p.625-642 in Proceedings of the Symposium Symbolic Languages in Data Processing, Rome, March 26-31, 1962.

[99] R. Peter, 'Recursive Functionen', Akadémia Kiadó, Akademischer Verlag, Budapest, 1961.

[100] A.J. Perlis, K. Samelson, 'Preliminary Report: International Algebraic Language', Commun. ACM 1:12, p.9, 1958.

[101] A.J. Perlis, 'The American side of the last phase of the development of `ALGOL`', in *History of Programming Languages*, Richard L. Wexelblat, (Ed.), New York: Academic Press, 75-91, 139-147, 1981.

[102] P.M. Priestley, *Logic and the Development of Programming Languages, 1930-1975*, University College London, PhD thesis, May, 2008.

[103] 'In Pursuit of Simplicity', a symposium held at the Department of Computer Science at the University of Texas to honour Dijkstra, May 2000.

[104] B. Randell, L.J. Russell, *ALGOL 60 Implementation: The Translation and Use of ALGOL 60 Programs on a Computer,* Academic Press Londan and New York, 1964.

[105] H.G. Rice, 'Letters to the Editor', Comm. ACM, L12-L13, September, 1960.

[106] H.G. Rice, 'Recursion and Iteration', Comm. ACM, Vol. 8, Number 2, February, 1965.

[107] S. Rosen, *Programming Systems and Languages,* New York, 1966.

[108] H. Rutishauser, 'Über automatische Rechenplanfertigung bei programmgesteuerten Rechenanlagen'. Z. Angew. Math. Mach. 31, 255, 1951.

[109] H. Rutishauser, 'Some Programming Techniques for the ERMETH', Journal of the ACM, Vol. 2, No.1, pp. 1-4, January 1955.

[110] H. Rutishauser, 'The Use of Recursive Procedures in `ALGOL60`', in Annual Review in Automatic Programming 3, Richard Goodman Ed. Pergamon Press, 43-52, 1963.

[111] H. Rutishauser, *Description of* `ALGOL60`, Handbook for Automatic Computation, Vol. 1, part a. Berlin and New York: Springer-Verlag, 1967.

[112] K. Samelson, F. Bauer, 'The ALCOR project', in: Gordon and Breach, editors, Symbolic languages in data processing: Proc. of the Symp. organized and edited by the Int. Computation Center, Rome, 26-31, pp. 207-218, New York, 1962.

[113] J.E. Sammet, *Programming Languages: History and Fundamentals*, Prentice-Hall Series in Automatic Computation, 1969.

[114] K. Sattley, P.Z. Ingerman, 'The Allocation of Storage for Arrays in `ALGOL60`'. Internal progress report, Office of Computer Research and Education, University of Pennsylvania, Nov. 1960.

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

[116] D. Shasha, C. Lazere, *Out of their minds: The Lives and Discoveries of 15 Great Computer Scientists,* Copernicus, Springer-Verlag, 1995.

[117] M. Sipser, *Introduction to the Theory of Computation*, Second Edition, Thomson Course Technology, 2006.

[118] R. Slater, *Portraits in Silicon*, MIT Press, First edition 1989, Third printing, 1992.

[119] C. Strachey, M.V. Wilkes, 'Some proposals for improving the efficiency of `ALGOL60`', University Mathematical Laboratory Technical Memorandum No. 61/5, 1961.

[120] C. Strachey, 'An impossible program', Letter to the Editor of the Computer Journal, p. 313, approximated date: 1965.

[121] D. Tamari, 'Machines de Turing et problemes de mot', Stichting Mathematisch Centrum, 2e Boerhaavestraat 49, Amsterdam, DR 10, 31 March, 1953.

[122] A.M. Turing, 'On computable numbers, with an application to the Entscheidungsproblem', Proceedings of the London Mathematical Society, Vol. 42, 1936-37, pages 230-265; reprinted in A.M. Turing, *Collected Works: Mathematical Logic,* pages 18-53.

[123] P. Wadler, 'Proofs are Programs: 19th Century Logic and 21st Century Computing', June 200, updated November 2000.

[124] H. Wang, 'A Variant to Turing's Theory of Computing Machines', J.ACM 4(1):63-92 (1957).

[125] J.H. Wegstein, 'From formulas to computer oriented language', Commun. ACM 2:3, 1959.

[126] A. van Wijngaarden, 'Ontwikkelingen op Computergebied', pp. 59-79. in: F. Van Der Blij, H. Freudenthal, J.J. De Iongh, J.J. Seidel, A. Van Wijngaarden, *Een Kwart Eeuw Wiskunde 1946-1971*, Serie voordrachten in het kader van de Vacantiecursus 1971, Mathematisch Centrum Amsterdam, 1973.

[127] M.V. Wilkes, 'Some reflections on Automatic Programming and on the design of Digital Computers', University Mathematical Laboratory Cambridge Technical Memorandum No. 61/2, 1961.

# Appendix A

# Backus Naur Form

Prior to 1959, the syntax of a programming language, such as `FORTRAN` and `IAL`, was described informally, e.g. in plain English. Table A.1 illustrates a fragment of the definition of `FORTRAN`'s syntax. In (i) the syntax of real numbers is defined (informally) and in (ii) some examples are provided. Note, in particular, that the numbers $10^{38}$ and $10^{-38}$ express a machine-dependent characteristic, thereby illustrating that `FORTRAN` was a machine-dependent language.

Table A.2 defines the syntax of real numbers for the machine-independent programming language `IAL` (later known as `ALGOL58`). Here, too, the definition is provided informally. Unlike the definition in Table A.1, however, this definition is machine independent. I.e., it does not contain specific numbers such as $10^{38}$ or $10^{-38}$.

As is illustrated in de Beer's thesis [9, p.26-27], informal definitions are, in general, ambiguous and incomplete, not to mention lengthy. Therefore, the definitions in Tables A.1 and A.2 were problematic to use in practice.

In 1959 at the IFIP congress in Zürich [3], Backus came to the rescue by introducing a notation which was later called Backus Normal Form. Naur noticed the power of the notation and, after making some small but important modifications, used it to define `ALGOL60`'s syntax. The corresponding notation is therefore often called Backus Naur Form (`BNF`) [66].

`BNF` is illustrated in Table A.3 on the same running example of real numbers. Line 1 defines the syntactic category <digit> to be either 0 or 1 or 2 or ... or 9. That is, | denotes 'or' and a digit is defined syntactically to be any number between 0 and 9. Line 2 *recursively* defines an integer to be either a digit or a concatenation of an integer and a digit. For example, the digit 5 is an integer. And, 57 is an integer but not a digit. Finally, line 3 merely defines a real to be either an integer (without a decimal point) or an integer followed by a decimal point and another integer. For example, the integer 5 is a real. And, 5.9 is a real but not an integer.

An important remark concerning Table A.3 is that the recursiveness in line 2 allows an arbitrary large (but finite) integer to be denoted. That is, line 2 only works if the finiteness of the machine is abstracted away. So, Table A.3 is the

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

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

(ii) Examples:
17.0
5.0
256.32
0.0033

Table A.1: The description of real numbers in the `FORTRAN` report [63].

Form: $N \sim G.G_{10} \pm G$ where each $G$ is an integer as defined above.
$G.G$ is a decimal number of conventional form. The scale factor ${}_{10} \pm G$ is the power of ten given by $\pm G$. The following constituents of a number may be omitted in any occurrence:
The fractional part $.00\ldots0$ of integer decimal numbers;
the integer 1 in front of a scale factor;
the + sign in the scale factor;
the scale factor ${}_{10} \pm 0$.
Examples: 4711 137.06 $2.9997_{10}10$ ${}_{10} - 12$ $3_{10} - 12$

Table A.2: The description of real numbers in the `IAL` report [100], with $G$ denoting a string of digits.

| | | | |
|---|---|---|---|
| (1) | <digit> | := | 0 \| 1 \| 2 \| 3 \| 4 \| 5 \| 6 \| 7 \| 8 \| 9 |
| (2) | <integer> | := | <digit> \| <integer><digit> |
| (3) | <real> | := | <integer> \| <integer>.<integer> |

Table A.3: An example in Backus Naur Form.

| SYNTAX | finite | infinite |
|---|---|---|
| informal | `FORTRAN` | `IAL` |
| BNF (formal) | 'clumsy' | `ALGOL60` |

Table A.4: Various ways to describe the syntax of a programming language.

formal equivalent of Table A.2 but *not* of Table A.1. Using `BNF` notation to capture the formal equivalent of Table A.1 is only possible by enumerating all possibilities, as opposed to using recursion. For example, suppose we have

<digit> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

and we wish to define any integer that is smaller than 10000. Then we would have to explicitly write:

<integer> := <digit> | <digit><digit> | <digit><digit><digit> |
<digit><digit><digit><digit>

Likewise, if we wish to define any integer that is smaller than 9999, then we would have to write –even more cumbersomely– all possibilities explicitly.

Table A.4 summarizes the previous discussion. `FORTRAN`'s syntax was defined informally and with respect to the finiteness of a particular machine. `IAL`'s syntax was defined informally as well, but infinite capacity (of some abstract machine) was assumed. I.e., the infiniteness expressed machine independence[1]. `ALGOL60`'s syntax was defined formally by using `BNF` notation and `BNF`'s power to express recursion concisely under the assumption that arbitrarily large numbers can be stored (in the abstract machine). Finally, the entry 'clumsy' in Table A.4 expresses that `BNF` is cumbersome (in many cases, impractical) to use if abstraction of the finiteness of the machine is not allowed.

Finally, we cite from Backus' 1959 paper:

> "[T]here must exist a precise description of those sequences of symbols which constitute legal `IAL` programs. [... But,] heretofore there has existed no formal description of a machine-independent language." [3, p.129]

Backus' formal notation was not applicable to his very own machine-dependent programming language `FORTRAN`. The advent of a machine independent programming language, such as `IAL`, was what was required for Backus to apply his formal notation. In Perlis' words in 1978, linguistics entered the arena of programming-language design with the advent of machine independence:

> "Linguistic growth –unlike `FORTRAN`, which was designed for a specific machine, and for which the issues were coding efficiency and properly so, `ALGOL` was designed for arbitrary, unknown machines. Consequently, the design of `ALGOL` focused on linguistic structure.

[1] Perhaps the infiniteness also served the purpose of closing the gap between the programming language `IAL` and the intended application domain of numerical analysis, where arbitrarily large mathematical objects prevail.

> They were the first languages, both `ALGOL58` and `ALGOL60`, in which linguistic issues forged to the front." [101, p.146]

# Appendix B

# On language constraints

On Friday, March 3, 1978, Dijkstra wrote an open letter [41] to a Lt. Col. William A. Whitaker, to comment on the design documentation of Ironman, that he had received a few days earlier.

In his letter, Dijkstra explained the practical infeasibility of Ironman's requirement:

> "There shall be no language restrictions that are not enforceable by translators." [41, p.0]

To do so, Dijkstra first addressed the meaning of the statement:

(1) "the integer procedure $f$ is free from side effects" [41, p.0]

To keep his exposition simple, Dijkstra assumed that $f$ was an integer procedure without formal parameters. In addition, calling $f$ would result, in general, in a value functionally dependent on the initial values of some of its global variables.

Dijkstra's first proposal in capturing the meaning of (1) was:

(2) Within its scope, the inner block
*begin integer* $h$; $h := f$ *end*
is semantically equivalent to the *empty* statement

He justified this choice by remarking that when $f$ is free of side effects in accordance to (2), then the following transformations might be undertaken as harmless by an optimizing compiler:

(i) transform $y := f * f$ into *begin integer* $h$; $h := f$; $y := h * h$ *end*
(ii) transform $b$ *or* $f = 1$ into *if* $b$ *then true else* $f = 1$
(iii) transform $a * f$ into *if* $a = 0$ *then* 0 *else* $a * f$

Dijkstra then argued as follows. Either severe and undesirable restrictions are made on the text of the procedure $f$, e.g. by defining a language that prohibits *all* possible side effects[1], or

[1] Dijkstra showed in his letter that there are procedures that have side effects and are practically relevant.

> "it is *impossible* for a translator to 'enforce' that the function procedure $f$ is in the above sense [e.g. (i)-(iii)] free of side effects, as it would require the solution of the halting problem, [...]
>
> [*begin integer* $h$; $h := f$ *end*] is *not* equivalent to the empty statement under all circumstances in which the calling of $f$ leads not to a properly terminating computation!
>
> In general, $f$ computes a partial function and calling $f$ only leads to a properly terminating computation provided some condition $D$ –describing its domain– is initially satisfied." [41, p.1]

Given the inadequacy of (2), Dijkstra provided a second proposal to capture the meaning of (1):

(3) *if* $D$
*then begin integer* $h$; $h := f$ *end*
*else skip*

and concluded:

> "In general, the condition $D$ with respect to which a function procedure is free of side effects needs to be stated explicitly; and the *user* of the function procedure *has to ensure* that this condition is satisfied wherever the function procedure may be invoked."
> [my italics] [41, p.1-2]

Similar to his lecture notes *A Short Introduction to the Art of Programming* [35], discussed in the previous section, Dijkstra stressed the responsibility of the user (i.e. programmer) and, hence, conveyed the infeasibility of completely automating a translator (i.e. compiler) that would respect all of the requirements stated in Ironman's documentation:

> "[W]e need to distinguish between the notion of 'a legal program' and the notion of 'a correct program'. From translators we can require that they reject illegal programs; for legal programs the language definition should define the *proof obligations to be met [by the user]* in order to make the legal program also a correct program."
> [my italics] [41, p.2]

In summary, Dijkstra applied the unsolvability of the Halting Problem to illustrate the infeasibility of one of Ironman's requirements. Based on Dijkstra's first words to the colonel:

> "[T]his letter is an almost immediate reaction to the 4 kg. of language design documentation that reached me last Monday; it was written because –at least in those few days– *I failed to discover an adequate treatment of an issue that now seems to be in urgent need of clarification.*" [my italics] [41, p.0]

Dijkstra may, perhaps yet again, have provided a novel application of the unsolvability of the Halting Problem in the fields of higher-level programming-language design and compiler implementation.