Konrad Zuse's Plankalkül: The First High-Level, "non von Neumann" Programming Language

WOLFGANG K. GILOI

Konrad Zuse was the first person in history to build a working digital computer, a fact that is still not generally acknowledged. Even less known is that in the years 1943–1945, Zuse developed a high-level programming model and, based on it, an algorithmic programming language called Plankalkül *(plan calculus). The Plankalkül features binary data structure types, thus supporting a loop-free programming style for logical or relational problems. As a language for numerical applications, the Plankalkül already had the essential features of a "von Neumann language," though at the level of an operator language. Consequently, the Plankalkül is in some aspects equivalent and in others more powerful than the von Neumann programming model that came to dominate programming for a long time. To find language concepts similar to those of the Plankalkül, one has to look at "non von Neumann languages" such as APL or the relational algebra. This paper conveys the syntactic and semantic flavor of the Plankalkül, without intending to present all its syntactic idiosyncrasies. Rather, it tries to point out that the Plankalkül was not only the first high-level programming language but in some aspects conceptually ahead of the high-level languages that evolved a decade later.*

To Konrad Zuse In Memoriam

Introduction

We know Konrad Zuse (Fig. 1) as an engineer who started developing program-controlled binary calculators from 1936 onward and completed the first fully operational digital computer in 1941. Zuse received the inspiration to use the binary number system from the *Dyadik* of G.W. Leibniz (1646–1716), realizing that the functions his binary machines had to perform could be described by logical expressions. This triggered in him a strong interest not only in propositional calculus but in mathematical logic in general.[1]

Through Brian Randell's book,[2] an English translation of Zuse's first patent application of 1936 has become more widely known, proving that Zuse had already developed many of the major concepts of the digital computer years before John von Neumann, Arthur Burks, and Herman Goldstine wrote their famous report *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument.*[3] However, Zuse's vision of digital machines and their potential capabilities went beyond the purely sequential computer that is known to this day under the name "von Neumann machine." For example, the possibility of array processing and even of parallel processing is already mentioned in his patent application of 1936.[2]

His first machine, called the Z1, was a mechanical digital computer. It was the size of a pool table and featured mechanical binary logic and flip–flop memory. Its arithmetic unit worked with a "semilogarithmic" or floating-point number representation that Zuse invented. The original Z1 did not survive the World War II air raids on Berlin. By initiative of the German National Research Center for Information Technology (GMD) and with contributions from the industry, two students and a mechanic under Zuse's supervision built a replica in 1988–1989 using his original documentation. The replica was installed at the Museum of Technology in Berlin in 1989. It proves that the civil engineer Zuse was, among his other talents, an ingenious designer of mechanical hardware.

Zuse began building the Z1 in 1936 and completed it in 1938. The mechanical arithmetic logic unit of the Z1 had some problems: Under certain conditions, it would get jammed and had to be reset (the replica emulates—unintentionally—even that feature). In 1938, while he was having a hard time making his mechanical design work, he started designing a second machine, the Z2. The arithmetic logic unit of the Z2 employed telephone relays instead of mechanical switches, while the memory still consisted of mechanical flip–flops. Its successor, the Z3 (1939–1941), was completely a relay

This paper is the revised version of a laudation rendered by the author at the celebration of Zuse's 80th birthday at his alma mater, the Technical University of Berlin.

1058-6180/97/$10.00 © 1997 IEEE

machine; it was the first fully operational programmable digital computer in history. Immediately after its completion, Zuse started the design of an upgraded version, the Z4. He had the good luck that toward the end of World War II, both he and the Z4 were evacuated to a safe place in Bavaria, before the Russian army entered Berlin. In 1951, Zuse leased the Z4 to the Technical University of Zurich, where it was in operation until 1953.

Fig. 1. Konrad Zuse (1910–1995).

Earlier, Zuse's work on digital computers had been temporarily interrupted by the outbreak of World War II, when he was drafted into the German army. Later, he was exempted from military service in order to work for an aircraft company. There he designed the special-purpose computers S1 and S2, both relay machines that were used to compute wing corrections needed to make the unmanned airplane HS293, a form of guided "buzz bomb" that today would be called a "cruise missile," fly. The S2 featured an integrated analog-to-digital conversion under program control, which makes it the world's first process control computer. Zuse received the prestigious IEEE Computer Pioneer Award for that achievement but, ironically, not for having built the first operational general-purpose digital computer. However, at the time, Zuse's computers were not considered important for the German war effort. Consequently, he got hardly any governmental support for their development, which forced him to build them in his spare time and with scrap material.

In the first postwar years, when the situation in Germany made it impossible for Zuse to continue building digital computers, his restless spirit found some ersatz occupation in the development of a high-level "programming schema" (today we would call it a programming model and language) for his digital computers. This work resulted in a programming language called *Plankalkül* (plan calculus). The concepts of the Plankalkül, for which there was no precedent whatsoever, evolved in the second half of the 1940s, and Zuse laid down those concepts in a monograph in 1945. His work on the Plankalkül was also meant to become a PhD thesis. Zuse wrote the thesis in 1944 but had no chance to submit it in the chaotic last days of the Third Reich. Not until three decades later was an English translation of it provided by the GMD in 1976 and published as a GMD report—alas, in rather poor English.[4] However, I doubt that many people ever read this publication, and even fewer people may have studied Zuse's concepts in detail, since GMD reports are not well-known outside Germany. A remarkable exception is D.E. Knuth and Trapp Pardo's paper,[5] which compares about 20 early programming language concepts that evolved before 1957. An exemplary algorithm, called TPK, is used to demonstrate the flavor of the different languages. The first language demonstrated and discussed in detail in that paper is the Plankalkül.

In the present paper, I shall first present the basic concepts of the Plankalkül. I will try to show that the programming paradigms found in it took a route somewhat different from the von Neumann programming model that evolved in the 1950s as the "canonical form" of computing and has maintained a dominating position to this day. To this end, I have to define the basic characteristics of a von Neumann language. I will try to explain why Zuse's Plankalkül had no impact on the development of von Neumann languages such as Algol-60 and its successors. I will also show that the Plankalkül anticipates notions and features of later "non von Neumann languages."

The Plankalkül

Basic Concepts

With the Plankalkül, Zuse wanted to provide a formal schema, or "calculus," for the construction of "computing plans" or, as the world has come to call them, "programs." Thus the name *Plankalkül*—a short expression in German for "calculus for computing plans." Zuse's aim was, in his own words, "to provide a purely formal description for any computational procedure."

Zuse writes about his motivations:

> I was not concerned with programs for numerical calculations when I developed the PK [Plankalkül]. I did not expect any difficulties in this field then, and consequently I concentrated my efforts mainly on the logical problems beyond the common numerical calculations.[4]

Consequently, the Plankalkül was designed for treating combinatorial problems as much as for formulating numerical algorithms. For Zuse, the archetype of a combinatorial problem was the chess game—Zuse's monograph devotes a 44-page chapter to programming chess games. From the beginning, he had the vision that his machines would eventually be able to play chess better than humans.

Let me start by listing some major concepts of the Plankalkül.

Explicitly Defined Binary Representations

All object types have binary representations that the programmer explicitly defines. This seems to be rather low level, yet it has the advantage that the user can efficiently program sophisticated nonnumerical operations in the form of Boolean expressions, for example, in chess playing.

Scalar Types and Data Structure Types

The Plankalkül features the scalar types common in all von Neumann languages. In addition, it comprises a variety of data struc-

ture types. Because of the visible binary representation of data objects, all variable types including scalars are data structures composed of Boolean elements. Therefore, all variables—scalars as well as data structures—are denoted as *structure*. Consequently, for every variable, a *structure specification* must be provided that defines implicitly the *variable type*, e.g., scalar, pair, list, or array.

Variables

The Plankalkül distinguishes input variables, intermediate variables, and output variables. Variable names consist of a letter and an integer number. The letter denotes the use of the variable: V stands for an input variable, Z for an intermediate variable, and R for an output variable. Constants are denoted by C. An attached index number identifies the individual variable. A variable is called by value. Its scope is the program in which it is declared.

Program and Program Equations

A program consists of a sequence of *program equations*. A program has a name under which it can be invoked. Programs and program equations have input and output variables. Internally, program inputs are inputs to program equations, and some of the program equation outputs will be the program outputs. An intermediate variable is used if the result of a program equation is not a program output but merely passed as input to a following program equation.

Semantically, a program equation plays the role of the function-type subroutine of the later high-level languages. Typically, a program equation executes a single Boolean or numerical expression with the inputs as arguments, assigning the result to one or several output variables or intermediate variables. A program equation may be composed of subequations.

Syntax

The syntax of the Plankalkül differs significantly from that of the later high-level languages. Every program equation consists of three or more lines. Only the first line has the familiar form of a statement, consisting (from left to right) of an expression containing input variables, the assignment symbol (⇒), and the name(s) of the output (result) variable(s). In this first line, variables are represented only by their use designator, V, Z, or R—a somewhat redundant feature, as the distinction between input or output parameters is already made by their position left or right of the assignment symbol.

The second line (V line) lists the identifiers (indices) of the individual variables. The V line may or may not be followed by a K line in which specific components of the (structured) variables are denoted by their index. Components may themselves be composed. In this case, subcomponents are denoted by subindexing, expressed by a dot notation. The last line (S line) declares the variable structure by a structure notation. For example, the structure notation 1.*n* means that the variable is a scalar (has one element) represented by a bit vector of length *n*. If a variable is already declared in a preceding program equation, of course, the structure specification need not be repeated. Structure specifications may be dynamic. This is expressed by the symbols σ and τ, which both denote an unspecified structure that will be dynamically defined during execution.

For example, a variable V_0 may be a bit vector with 32 bits or, in Zuse's notation, a structure of the type S1.*n* with $n = 32$, and if an operation is to be performed on the second least significant bit (bit 30), this would be indicated by writing

V	0
K	30
S	1.32

As an illustration, we present a program equation, E1.1, for the addition of two numbers, R0 = V0 + V1, and another equation, E1.2, for incrementing a counter variable Z (taken from Zuse's own work,[4, p.158]).

*E*1.1

	V	+	*V*	⇒	*R*
V	0		1		0
S	1.*n*		1.*n*		1.*n*

*E*1.2

	Z + 1 ⇒ *Z*
V	2 2

The syntax of the Plankalkül differs significantly from that of the later high-level languages.

As mentioned earlier, a program equation may invoke other programs or program equations as a subprogram, with any depth of nesting allowed. To call a program as a subprogram, its result variable R, indexed by its identifier, is referenced, followed by the variable(s), set in parentheses, of the calling equation with which the subprogram is to be executed. The scope of the subprogram variables is that of the calling program. For example, to call program P4.1 with result R as subprogram of equation E1.3, one writes[4]:

*E*1.3

	*R*4.1	(*Z*) ⇒ *Z*	
V	0	0	1
S		1.*n*	1.*n*

If a program is solely used as a subprogram, this may be indicated by adding the letter Z to its name (in the example above, one would write PZ instead of P and RZ instead of R).

Structure Specification

I mentioned above that for any variable of any type, a structure specification must be given that specifies the binary representation of the variable, thereby implying its type. Fig. 2 gives examples of structure specifications.

Control Constructs

Zuse's early machines did not have a branch instruction. Consequently, the Plankalkül does not have a GOTO statement. Zuse states in *Comment of the Plankalkül* written 25 years later:

> I had in mind to introduce [a] method of program control into the basic syntax of the PK [that] would have corresponded to the GOTO statement and the use of program labels. But I hesitated to take this step since, at that time, I did not have a satisfactory overview of the possible consequences.[4]

There are two control constructs available in the Plankalkül: one for the conditional execution of a program equation and the

other for the repetitive execution of a program. The conditional execution corresponds to the IF statement of later languages.

The FIN and FIN^2 constructs allow program executions to be terminated or iterated, respectively, when a given condition becomes true. This is mechanized by assigning a Boolean value to FIN or FIN^2, respectively. FIN provides an exit from a program that may be used when further execution becomes unnecessary. (Note: APL has such a mechanism, too. I refrain here from elaborating on the potential danger of uncontrolled program exits.) FIN^2 leads to the renewed execution of the program rather than terminating it. As an example of a possible use of FIN, Zuse considers the disjunctive connection of three Boolean variables, which can be terminated as soon as one of the three operands becomes true.

$$V \vee V \vee V \Rightarrow R$$
$$0 \quad 1 \quad 2 \quad 0$$

$$V \Rightarrow Z \mid Z \Rightarrow \text{FIN}$$
$$0 \quad 0 \mid 0$$

$$Z \vee V \Rightarrow Z \mid Z \Rightarrow \text{FIN}$$
$$0 \quad 1 \mid 0$$

$$Z \vee V \Rightarrow R$$
$$0 \quad 2 \quad 0$$

The FIN^2 construct causes program execution to begin anew. This may be repeated until another statement with the simple FIN terminates the repetition. Thus, by the combination of FIN and FIN^2 with program equations that compute jump conditions, programs may execute repeatedly in the REPEAT ... UNTIL fashion. As the Plankalkül allows for all variants of incrementing or decrementing iteration counters, one can also program FOR-type loops—an example of this can be found in Zuse.[5] In the iterative execution of programs, the (varying) argument values with which the repetitions are carried out are obtained through indexing. Since variable identifiers in the Plankalkül are always index numbers, all that needs to be done is to use index parameters computed inside the program.

Scalar Data Types

The Plankalkül comprises a large variety of standard data types, yet it also allows the declaration of user-defined types. There exist, for example, the following predefined scalar types:

- complex numbers
- real numbers
- integer numbers
- nonnegative integers
- negative integers
- rational numbers
- nonnegative rational numbers
- binary numbers

Recall that in the Plankalkül the only primitive scalar type is the single bit. Integer numbers, for instance, are bit vectors and, thus, already structures.

Standard types have fixed symbolic denotators. The denotation can be refined in order to specify idiosyncrasies of the binary representation. For example, one can distinguish between an integer representation by absolute value and sign or by two's complement. A variety of arithmetic functions are defined on the numerical standard types, corresponding with what can be found in any better high-level programming language, including its mathematical function library.

Notation	Structure
S0	Single boolean value
S1.n (= nxS0)	Bit vector with n elements
nxS1.4	Integer with n digits in BCD representation
mxnxS1.32	mxn matrix consisting of m vectors of n 32-bit integers
σ, τ	Unspecified structures, to be defined on program execution
□x σ	List of variable length (□) with undefined elements (σ)
□x2 σ	List of pairs of equal elements
□x(σ,τ)	List of pairs of different elements

Fig. 2. Examples of structure specifications.

User-defined types can be anything the user wants. Zuse[4] illustrates this by the example:

- persons
- age
- gender
- marital status
- other data pertaining to a person.

Another set of Zuse's examples is:

- the fields of a chessboard
- the pieces of the chess game, including a definition of the way they are allowed to move
- the edges of a graph or
- whatever else the application may demand.

The price the programmer has to pay for the ability to define arbitrary types is that for each type introduced, not only its definition but also its binary representation must be specified. The binary representation is always a bit string; its interpretation is determined by the specification the user provides as part of the type definition.

All operations of user-defined types are operations on the binary representation, that is, expressions of proposition calculus or predicate logic. This allows the programmer to define subsets of the set of elements of a type according to propositions of predicate calculus as illustrated by the simple example shown in Fig. 3.

Data Structure Types

I mentioned above that a data structure type of the Plankalkül consists of a set of scalar values plus a binary pattern specifying their structuring.

All data structure types have as a basis the binary tree whose nodes are readily addressed by a bit string. Other structures, e.g., lists and arrays, must be mapped onto binary trees. Consequently, the user sees, in the case of an array, the tree that represents it. This is the price to pay for the generality of constructing arbitrary data structures in the form of trees.

Lists and lists of pairs are the most important structures in the Plankalkül. The latter allow the representation of generalized graphs, that is, arbitrary relations.[6] A relation is defined by listing all its pairs. This approach is appropriate, e.g., for representing geometric structures. Zuse[4] treats various examples of graph theoretical problems.

The elements of a data structure may be of any given type. For example, element types may be:

- binary values
- integer numbers
- real numbers
- lists
- lists of pairs
- vectors

Higher dimensional arrays, e.g., matrices, may be defined as vectors of vectors etc. As in APL, structures can be dynamically created by the execution of appropriate program equations, in the Plankalkül called "structure equations."

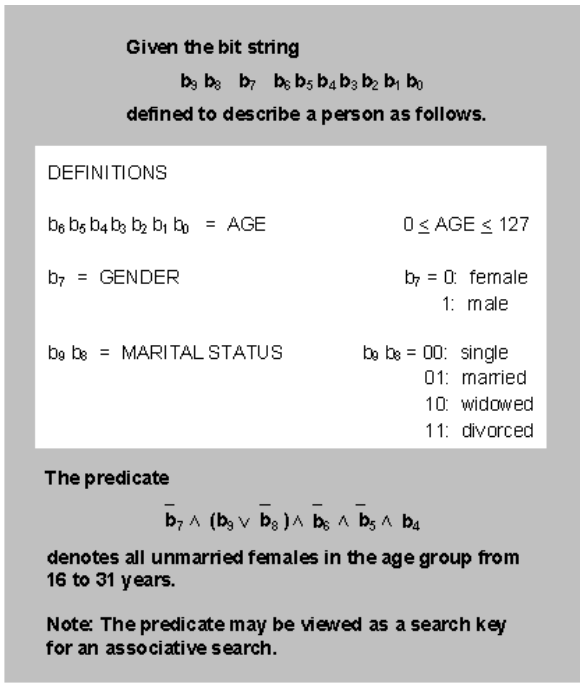


Fig. 3. Example of a user-defined data structure.

Lists are dynamic, i.e., may grow and shrink. An *empty variable* symbol is provided to be used as a placeholder for dynamically created elements. List operations exist for the following:

- creating a sublist containing all elements that satisfy a given predicate
- obtaining the number of elements of a list
- obtaining the next element
- obtaining the first or last element
- obtaining the smallest or largest element
- adding an element to the head or tail of a list
- concatenating two lists.

Hence, the type *list* is a true data structure type that encompasses a number of functions applicable to list objects. Arrays, on the other hand, exist only in the rudimentary form as found in the later high-level languages. One can construct arrays, but the only function defined in them is indexing, allowing for the selection of array elements. Zuse had planned to add genuine array types, e.g., matrices with matrix operations, but postponed it in favor of the more intricate relational structures he needed for his chess programs. He considered the addition of array types as trivial.

Plankalkül and the Evolution of High-Level Languages

Intention of the Plankalkül

It is fair to assume that the development of the Plankalkül was strongly influenced by Zuse's experience with his early machines, Z1 through Z3, which neither supported program loops nor had a stack. Thus, they were not "von Neumann machines" in the above definition. The Z4 was Zuse's first machine that had branch instructions.

Zuse emphasized, on many occasions, that from the beginning, he had very clearly seen the possibility of storing not only data but also instructions in the memory of his computer, that is, execute programs out of store. However, in his early designs, he considered memory—the most expensive part of the machine—too precious to be "wasted" on storing code rather than data. Consequently, programs were stored in a separate read-only memory realized in the form of punched celluloid film (he obtained this as scrap from movie studios). This decision implied a loop-free, functionally oriented programming style. He mechanized this in the Plankalkül in the form of an operator language.

He considered memory ... too precious to be "wasted" on storing code rather than data.

Of course, when Zuse developed the Plankalkül, he had already come to realize that many important algorithms were iterative by nature. This motivated him to provide a mechanism that allowed the programmer to embed sequences of program equations in a program or subprogram that could be repeatedly executed. APL, although much richer in types (*n*-dimensional arrays and operations of linear algebra), is based on the same concept.

Zuse was often asked what motivated him in 1936 to start developing computing automatons. His standard response—which always earned him some laughs—was "laziness." He then used to point out that, as a civil engineer, he had to do tedious and boring numerical computations, which made him want a machine to do them for him. This representation earned him points with journalists, but it is at best only partly true. Though he considered numerical computations to become the "bread-and-butter" applications of his machines, his heart was in their use for solving sophisticated logical problems—the epitome being chess playing. Had his focus in 1945 been on numerical problems, e.g., solving systems of linear equations (which he had to do all the time when working in the aircraft industry), he would probably have come up with a language centered on array operations. Instead, he de-

vised a language providing the expressive means to formulate problems of propositional calculus, predicate calculus, and set theory in a procedural manner.

To demonstrate the kind of applications—besides chess playing—Zuse had in mind when he designed the Plankalkül, consider a simple example of list processing. Given a unidirectional graph, the task is to write a subprogram that yields the number of edges leaving a given node. As in other programming languages, there exist two approaches to graph representation[4]:

1) for a graph with N nodes, construct a Boolean $N \times N$ matrix and mark each edge with a "1"
2) generate the list of relation pairs (the source and destination node of an edge).

Since the Plankalkül comprises the data type *list of pairs* as well as a variety of list operations, the second approach is quite straightforward.

The subprogram, let it be called U5.1, is of the general form: $R(V_0, V_{1.0}) \Rightarrow R$, where V_0 denotes the list of pairs, V_1 a single pair, and $V_{1.0}$ the first element of that pair. The operation to be performed is to count the number of pairs in V_0 in which $V_{1.0}$ occurs as the first element. In the Plankalkül, this is readily performed by using the THOSE WHICH operator[4] and the operator N(L), which yields the number of elements in a list L.[4] The THOSE WHICH operator, $\hat{x}$, has the form: $\hat{x}R\square(V_0) \Rightarrow R$, yielding as a result the list of elements of V_0 for which the predicate $R\square$ is true. This leads to the following program that can be read almost like a mathematical formula:

$\underline{U5.1}$

	$R(V$	$V)$	$\Rightarrow$	R
V	0	1		0
K		0		
S	$m \times 2\sigma$	σ		$1.n$

V	$N[\hat{x}[$	v	$\in$	V	$\wedge$	v	$=$	V	$]] \Rightarrow R$
				0				1	
K								0	0
S		2σ		$m \times 2\sigma$		σ		σ	$1.n$

Lack of Impact of the Plankalkül

Zuse and other authors[7] see the Plankalkül as a forerunner of algorithmic programming languages such as Fortran and Algol. This view is based on some major characteristics the Plankalkül has in common with those languages:

- the notion of variables, including variable declaration and assignment
- the notion of subroutines of the function type
- the conditional or repetitive execution of subprograms or programs

It is not surprising that, in the early years of high-level language development, the Plankalkül had so little impact for the simple reason that it was hardly known. However, Zuse more than once expressed his disappointment that the Plankalkül had so little, if any, impact on the later development of programming languages and that his priority in the invention of the above concepts was not duly acknowledged.

Because of the forced exodus of its elite of artists and scientists during the Third Reich, Germany lost its traditional role as a leader in science and as a place that would attract foreign scientists. Consequently, the German language had ceased to be a major science language. The postwar conditions in Germany hindered the spread of scientific publications for years. Therefore, Zuse's monograph on the Plankalkül was hardly known in Germany, let alone outside the country. It took three decades for the report[4] to be published in English. The same report contains the draft of the PhD thesis Zuse wrote in 1944 but did not have a chance to submit to a university.

In an addendum to the monograph written in 1972, Zuse expressed his disappointment that even his German colleagues involved in the development of Algol did not give him due credit for the programming concepts proposed in the Plankalkül, although they knew about it.[4]

I deem it questionable that the creators of Algol were intentionally denying Zuse the credit he deserved as the inventor of important programming concepts; rather, I give them the benefit of the doubt that this came from a lack of understanding. What the creators of Algol wanted to, and did, accomplish was to raise programming to a higher level of abstraction. They may have viewed the explicit manipulation of binary structures as the basis of all operations in the Plankalkül, whose rationale they may not have appreciated, as unnecessarily low level. Although they must have recognized the semantic similarities between the Plankalkül and some concepts of their language, the crudeness (in their view) of the former as an unstructured operator language without safe mechanisms for data encapsulation and scope control may have led them to discard it as too exotic and low level to be credited as a forerunner of what they considered a good high-level algorithmic programming language, the more so as for them "algorithmic" meant primarily numerical operations.

To back this conjecture, note Zuse's intentions with the Plankalkül. I will try to show that the Plankalkül intentionally presented a "non von Neumann" model of computing, similar to what a decade later Kenneth E. Iverson had in mind with the creation of APL.[8] To this end, I first must define the term "von Neumann language." It will then become obvious why the protagonists of the later "von Neumann type" programming languages had little appreciation for Zuse's endeavors, as little as they had for the APL concepts.[9]

More than a decade after the Plankalkül was conceived, Zuse built a computer company that, in the late 1950s and early 1960s, became the main supplier of German universities with an affordable electronic digital computer, the Z22. Originally, the drum machine Z22 had to be programmed at the machine level (this author belongs to the generation who originally learned digital programming on a Z22, i.e., the hard way). Zuse[4] writes that at that time he considered implementing the Plankalkül on his computers, however, he came to realize that this would overtax the financial resources of his company. At a time (around 1959) when compilers were still practically unknown and the only programming facility for the Z22 was some kind of assembler language called "Freiburger Code," the Plankalkül would have made a sensational high-level programming system.

Plankalkül and "von Neumann Languages"

A programming language is the realization of a certain programming model that, in turn, may be defined by the properties of an abstract machine. Languages of the Algol type have as their underlying abstract machine the "von Neumann machine," given in the specific form of a stack machine. This leads to the following characteristics of a von Neumann language:

- The data objects of the language are strictly scalar memory objects; there exist no true data structure types. Variable names are symbolic addresses of memory objects, to the extent that there exists the explicit type "memory address" (pointer).
- A computation consists of the step-wise (sequential) transformation of the states of memory objects. Consequently, the language offers only scalar operations.
- There may be procedures (program blocks) nested to any arbitrary depth.

Array declarations serve the purpose of having the compiler reserve a contiguous memory space for the storage of a homogeneous, ordered set of scalar data; consequently, the elements of the set can be accessed through computed addresses (indices). Besides indexing, there exist no array operations such as inner or outer products etc. The lack of array operations or other data structure types is reflected in the application program by the occurrence of loops in which the scalar array elements are transformed one at a time.

It was not before the mid-1970s that even the Algol school finally came to realize that the lack of comprehensive data structure types is a disadvantage.[10] This shortcoming weighs particularly severely under the aspect of parallel processing, for which von Neumann languages offer no expressive means. To execute a "von Neumann program" in parallel, it must first be "parallelized." At the present state-of-the-art, the programmer is burdened with this task. In the future, with the development of parallel programming languages such as High Performance Fortran[11] and others, this task will be supported by or even totally automatically performed by the compiler.[12]

The above characterization of a von Neumann language immediately reveals a significant difference from the Plankalkül, as the latter features the notion of data structure types, thus favoring a largely loop-free programming style.

Plankalkül and "non von Neumann Languages"

Well-known examples for programming languages that deviate significantly from the von Neumann model are APL, Lisp, and SETL. As mentioned above, APL is based on multidimensional dynamic arrays. Lisp has dynamic list structures, built with the CDR-CON mechanism as its primary data type. SETL is based on sets in the set-theoretical definition; consequently, its functions include set operations such as union, intersection, etc.

The development of APL was undertaken with the declared goal of creating a new, more powerful algebra.[8] To this end, Iverson combined the operations of linear algebra with the procedural concept of state (a concept alien to mathematical algebra). The powerful array operations of APL allow for a loop-free programming of algorithms that in a von Neumann language must be executed iteratively. Therefore, the APL types provide an appropriate basis for parallel programming and, consequently, can be found in Fortran90, the modern version of a data-parallel Fortran.

Zuse's Plankalkül, with its emphasis on logical and relational operations, comes closer in intent to a set-oriented language such as SETL (albeit on a lower level of abstraction). The relationship of the Plankalkül with Lisp, on the other hand, is superficial. Though both languages comprise the data type *list*, the representation of list structures and their intended use in the Plankalkül is rather different from that of Lisp.

Conclusion

I hope to have shown that with the development of the Plankalkül in 1943–1945, Zuse took a broader view of computing than, a decade later, John Backus with Fortran or Bauer and Naur with Algol-60. In motivation and aims, Zuse's Plankalkül came much closer to Iverson's APL or Ted Codd's concepts of a relational algebra.

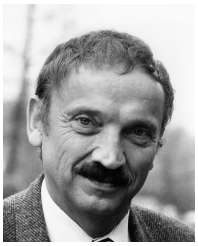
For the development of computer science, it certainly was a loss that under the given circumstances, the Plankalkül remained practically unknown. Had Zuse's monograph become as widely known as the report of Burks et al.,[3] some developments such as the relational data base, logical programming,[13] or the creation of standardized forms of knowledge representation in artificial intelligence might have evolved sooner.

Acknowledgments

The author is greatly indebted to Stefan Jaehnichen, J.A.N. Lee, Niklaus Wirth, Michael Williams, and the reviewers for very valuable discussions and suggestions.

References

[1] K. Zuse, *Der Computer—Mein Lebenswerk*. Munich: Verlag Moderne Industrie, 1970.
[2] B. Randell, ed., *The Origins of Digital Computers*. Berlin-Heidelberg-New York: Springer-Verlag, 1973.
[3] A.W. Burks, H.H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," A.H. Taub, ed., *Collected Works of John von Neumann*, vol. 5. New York: Macmillan, 1963, pp. 34-79.
[4] K. Zuse, *The Plankalkül*, GMD Report no. 175, 2nd ed. Munich-Vienna: R. Oldenbourg-Verlag, 1989.
[5] D.E. Knuth and T.L. Pardo, "The Early Development of Programming Languages," N. Metropolis, J. Howlett, and G.-C. Rota, eds., *A History of Computing in the Twentieth Century*. New York: Academic Press, 1980, pp. 197-208.
[6] D.E. Knuth, *The Art of Computer Programming*, vol. 1. Reading, Mass.: Addison-Wesley, 1975, 2nd ed.
[7] F.L. Bauer and H. Wössner, "The Plankalkül of Konrad Zuse, a Forerunner of Today's Programming Languages," *Elektronische Rechenanlagen*, 1972, H.2.
[8] K.E. Iverson, *A Programming Language*. New York: J. Wiley & Sons, 1962.
[9] W.W. Dijkstra, "The Humble Programmer," *Comm. ACM*, vol. 15, no. 10, pp. 859-866, 1972.
[10] E.W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.
[11] "High Performance Forum: High Performance Fortran Language Specification," *Scientific Programming*, vol. 2, pp. 1-170, 1993.
[12] W.K. Giloi, M. Kessler, and A. Schramm, "PROMOTER: A High-Level, Object-Parallel Programming Language," Sahni, Prasanna, and Bhatkar, eds., *Proc. Int'l Conf. High Performance Computing*. New Delhi: McGraw-Hill, 1995, pp. 661-666.
[13] R. Kowalski, "Predicate Logic as Programming Language," *Proc. IFIP Congress*, 1974.



Wolfgang K. Giloi received his Diploma (1957) and PhD (1960) in electrical engineering from the University of Stuttgart. From 1960 to 1964, he was development engineer at AEG-Telefunken and general manager of its Analog/Hybrid Computer Department. From 1965 to 1970, he was a professor of electrical engineering at the Technical University of Berlin and department director at the Heinrich-Hertz-Institute (1966 to 1973). From 1968 to 1969, he a visiting professor at MIT. From 1971 to 1977, he was a professor of computer science at the University of Minnesota. Since 1978 he has been a professor of computer science at the Technical University of Berlin and (since 1983) director of the Research Institute for Computer Architecture and Software Technology of the German National Research Center for Information Technology. He was appointed as an adjunct professor of computer science at UCLA (1988), honorary professor at Shanghai Jiao Tong University (1990), and member of the Academy of Science of Berlin-Brandenburg (1994). Dr. Giloi is the architect of two of the world's most powerful massively parallel computers, Suprenum (completed 1989) and Manna (1993). He was awarded the Ring of Honor of the German Society of Professional Engineers (VDI), the IFIP Silver Core, and the Cross of Merit First Class of the Federal Republic of Germany. Since 1991, he has been an IEEE Fellow and has served since 1992 on the Board of Governors of the IEEE Computer Society. From the 1960s onward, he had enjoyed a close personal relationship with Konrad Zuse.

The author can be contacted at
GMD Research Institute for Computer Architecture and Software Technology
Rudower Chaussee 5
12489 Berlin, Germany
e-mail: w.giloi@computer.org