Plankalkül: The First High-Level Programming Language and its Implementation

Raúl Rojas, Cüneyt Göktekin, Gerald Friedland, Mike Krüger

{rojas, goekteki, fland, krueger}@inf.fu-berlin.de

Freie Universität Berlin
Institut für Informatik

Olaf Langmack, Denis Kuniß

{langmack,kuniss}@feinarbeit.de

Feinarbeit®, Berlin

Technical Report B-3/2000
February 2000

Freie Universität Berlin
Department of Mathematics and Computer Science
Takustr. 9, 14195 Berlin, Germany

Plankalkül: The First High-Level Programming Language and its Implementation

Raúl Rojas, Cüneyt Göktekin, Gerald Friedland, Mike Krüger

{rojas, goekteki, fland, krueger} @inf.fu-berlin.de

Freie Universität Berlin
Institut für Informatik

Olaf Langmack, Denis Kunuß

{langmack,kuniss}@feinarbeit.de

Feinarbeit®, Berlin

February 2000

Abstract

This paper presents the first implementation of Plankalkül, the programming notation invented by the German engineer Konrad Zuse in 1945. Plankalkül is a high-level imperative programming language. In Plankalkül, programs are functions which can be called non-recursively in other programs. There are no preliminary variable declarations: the type of a variable is specified when it is used. The main constructs are: variable assignment, arithmetical and logical operations, guarded commands and While loops. Some special list and set processing functions are part of the language definition. Plankalkül uses a two-dimensional layout that defies traditional parsers. This, and some inconsistencies in the original definition, have been the main obstacles for its implementation over the years.
We suppressed some inconsistencies in the language and identified a mighty subset of Plankalkül for which we wrote a syntax directed editor, a parser, and a run-time system. The code was written in Java and runs in every workstation connected to the Internet. The editor generates only syntactically valid programs in Zuse's original two-dimensional layout, even when the programmer is first learning the language. Fifty five years after its conception, the first Plankalkül programs ran in February 2000.

1. Introduction

Konrad Zuse designed the high-level programming language Plankalkül (Calculus of Programs) in 1945, after moving out of Berlin at the end of World War II. Anyone who has had the opportunity to study the original definition of Plankalkül is struck by its modern flavor and powerful constructs – it seems as if it had been created much later than 1945. Most amazing, however, is the fact that at the time that Konrad Zuse was writing his Plankalkül document, the only two working computers in the world were the ENIAC and the Harvard Mark I. None of them used a compiler or a formula translator – the ENIAC had even to be rewired for every different problem.

From 1936 to 1945, Zuse built three programmable computers that embodied the same general computational principles. In the lapse of those nine years, Zuse single-mindedly pursued a clear cut architectural concept. The Z1 (1938), the Z3 (1941) and the Z4 (1945) were all binary "algebraic" floating-point machines, with a memory separated from the processor, and a program stored in punched tape. They were programmed in machine language, like all first American or British computers. By his own account, Zuse very soon realized that his „combinatorics of conditionals" (as he called it) was identical to predicate calculus and he conceived a much more powerful machine, the „logic machine", which would be more general and supersede the algebraic machines he had already built.

Although Zuse applied for a patent for the logic machine, he never really finished its design. The logic computer was truly minimal and similar to a Turing Machine: it consisted of a memory with one-bit words, and a processor capable of executing only the logical operations AND, OR, and NOT on one-bit operands. The machine would be capable of solving any numerical or logical problem and, although this is not widely known, its programming language would be the Plankalkül.

In 1942/43, Zuse started writing a long document which he intended to submit as a Ph.D. dissertation. The draft describes the predicate logic in order „to make it accessible for engineers", and goes into great detail into its implementation with mechanical and electrical relays. Konrad Zuse's planned thesis, never submitted, is in fact one of the first treatises on the systematic construction of computer circuits. He describes how to map logical formulas to relay circuits and vice versa. He considers the problem of minimizing circuits and how to overlap them in order to use less components. He explains clocked circuits and everything else that is be needed to build a computer.

The continuation of this unfinished thesis is the Plankalkül document drafted in 1945. Hindered of working on the Z4, which he moved out of Berlin and to Hinterstein, a small town in the Bavarian Alps, Zuse sat down to summarize how he thought logic machines of the future should be programmed. In modern terminology, the Plankalkül has the following main features:

– it is a high-level imperative programming language
– programs are reusable functions
– functions are not recursive
– only call by value is used in function invocation
– variables are local to functions (programs)
– it is a typed language

– the fundamental data types are arrays and tuples of arrays
– the type of the variables does not need to be declared in a special header
– conditionals are processed using guarded commands
– there is a WHILE construct for iteration
– there is no GOTO construct

The main non-modern feature of Plankalkül is its mixed one-dimensional and two-dimensional layout, which has puzzled many readers of the original document. Variables are written using four lines instead of using parenthesis to enclose indices. It could well be that this special layout has been one of the main obstacles for the construction of a compiler or interpreter for the language in the last five decades.

2. **The Syntax of the Plankalkül**

The original document describing Plankalkül [Zuse 1972] is not free of contradictions and there are also several ambiguities that must be solved before attempting to write a compiler for the language. Therefore, in what follows we have identified a powerful subset of the language, which is computationally complete and free of ambiguities. In defining this subset we were guided by the following principles:

- Historical accuracy. We wrote a syntax-oriented editor that preserves the original two-dimensional structure of the language.
- Simplicity. Zuse left alternative syntactical options open at several points in the definition of the language. We preserved only one option in each case, in order to make the syntax unambiguous, especially regarding data types.
- Induction from examples. When Zuse did not clearly outline the syntax or operational semantics of language constructs, we inferred them from the numerous examples contained in the founding Plankalkül document.
- Regularity. When the syntactical options were ambiguous, we selected one that made the language more regular and "orthogonal".
- Easy implementation. For the first subset of Plankalkül that we defined, we selected only those constructs that are easy to implement in a conventional computer. We left set and predicate logic constructs out of the selected language subset. They can be implemented later using macrodefinitions and a standard Plankalkül library.

We call the subset of Plankalkül obtained applying these principles „Plankalkül 2000“.

2.1 **Plankalkül 2000 - Variables and data types**

Variables are essential to any imperative programming language. In Plankalkül there are three main classes of variables:

- V variables, numbered V0, V1, etc., which are read-only.
- Z variables, numbered Z0, Z1, etc., which can be read and written.
- R variables, numbered R0, R1, etc., which are write-only.

The V variables are used to pass parameters to programs, the Z variables hold intermediate results, and the R variables are used to pass the final result of the program.

Aditionally, there are “loop variables”, which are used in While loops. They are denoted i0,i1, i2, etc., according to the depth of loop nesting, and are of generic numeric type. We will have more to say about these variables later.

All variables have a „structure“ or type. The following data types are possible:

- One bit, denoted as “0”
- *n* bits, where *n* is an integer, denoted “*n*.0”
- Tuples of other types. For example (3.0,4.0) denotes a pair of variables, one of 3 bits, the second of 4 bits. Tuples can have two or more elements.
- *m* times any other type, for example 4.5.0, which denotes an array of five elements, each one of five bits.

Some examples of possible data types are:

- 8.0	a byte
- 16.8.0	a vector of 16 bytes
- (0, 8.0, 16.0)	a triple formed of one bit, 8 bits, and 16 bits
- 32.(0, 8.0, 16.0)	an array of 32 triads with the structure above

It is easy to see that data structures in Plankalkül can be implemented as trees. The last example given above represents a tree with 32 children nodes at the root level. Each child has three children and so forth. It is important to notice that tuples are just another syntactical way to refer to arrays. We need tuples when the data type of each element in the array is different. We use vectors when the data type of each element is the same. In Plankalkül 2000 all variables are vectors or tuples, or combinations of both.

There are no variable declarations at the beginning of a program, instead each variable carries its own type. Variables are generally written using four lines:

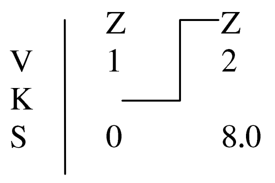
$$\begin{array}{c|l} & Z \\ V & 1 \\ K & \\ S & 5.0 \end{array}$$

This example refers to the variable Z1 of type 5.0 (five bits). The subindex of the variable is written in the “V” line, the component of the variable in the „K“ line, and the type in the „S“ line. The annotations to the left of the vertical line are just a mnemonic device and are not part of the syntax. Variable Z1 is a vector of five bits. If we want to refer to the first bit in the vector we write:

$$\begin{array}{c|l} & Z \\ V & 1 \\ K & 0 \\ S & 0 \end{array}$$

Notice that the components of arrays are numbered starting from zero. Notice also, that the type of the component selected in the example is a single bit. The VKS annotation can be omitted, as we do in the examples that follow.

Component indices can be variable. We can refer to the component of the variable Z1 whose number is stored in variable Z2 as follows:



The connecting line means that the content of variable Z2 (a byte) is used as index for variable Z1. The indexed component is of type “0” (a bit).

2.2. Arithmetical and logical statements

The symbol ⇒ is used to denote value assignment. Variable assignments are read from left to right, like in the following example of a Plankalkül statement:

$$\begin{array}{ccccc} V & + & V & \Rightarrow & Z \\ 0 & & 0 & & 2 \\ 0 & & 2 & & \\ 8.0 & & 8.0 & & 8.0 \end{array}$$

Here, the component 0 of V0 and the component 2 of the same array are added and the result is stored into variable Z2, which is an array of eight bits. The component line of Z2 is left empty, since we want to refer to the whole array of eight bits. Only V and Z variables (and loop variables) can appear in expressions to the left of the assignment symbol. Only Z and R variables to its right.

The four basic arithmetical operations are defined in Plankalkül. We use the symbols +, -, × and / to denote addition, subtraction, multiplication and division. Since each variable „carries" its type, the programmer has to be careful of writing only valid arithmetical operations, otherwise a run-time error will result. We adopt the convention that arithmetical and logical operations are only valid for arguments of the same type (generically, *n*.0). The result has also the same type as the arguments. We use two's complement arithmetic to perform the operations.

There are logical operators for conjunction, disjunction and negation, denoted with the symbols ∧, ∨, and ¬. The conjunction of two bits, for example, can be written as:

```
Z   ∧   Z     ⇒    Z
0       1          2
                   1
0       0          0
```

Here, we compute the conjunction of two variables Z0 and Z1 (single bits) and store the result in variable Z2, component 1. Variable Z2 is therefore an array of bits and we are selecting only one of its components.

Negation is expressed in Plankalkül by writing a dash above the name of a variable or an expression. For implementation convenience we will use instead the unary operator ¬ to denote negation. Two other logical operators are defined in Plankalkül: the identity operator ~ and the XOR operator /~.

Constants are written in Plankalkül in the first line of the tabular notation, like in:

```
Z    +   2     ⇒    Z
0                   2

8.0                 8.0
```

We will always assume that the type of a constant is the type of the other variable argument or of the result (when two constants are combined).

There are no arithmetical operations for tuples, but tuples can be assigned to tuples with the same number of elements.

2.3 Guarded commands

There is a construct in the Plankalkül which could be interpreted in other high-level programming languages as an IF-THEN statement. It corresponds to guarded commands in some modern languages.

The symbol $\rightarrow$ is used to denote conditional execution, it separates a logical expression and a statement. The statement to the right of the arrow is executed only if the logical value to the left of the arrow is true (that is, a 1).

For example, the statement:

$$\begin{array}{lllllllll}
Z & \wedge & Z & \rightarrow & V & + & V & \Rightarrow & Z \\
0 & & 1 & & 0 & & 0 & & 3 \\
 & & & & 0 & & 2 & & \\
0 & & 0 & & 8.0 & & 8.0 & & 8.0
\end{array}$$

means that if the conjunction of the two bits stored in Z0 and Z1 is true, the addition will be performed, and the result will be stored in Z3. Notice that the arrow symbol binds more strongly than the assignment symbol, and the logical and arithmetical operation symbols, more strongly than the arrow. Brackets can be used to disambiguate expressions, like in the example below:

$$\begin{array}{lllllllll}
(Z & \wedge & Z) & \rightarrow & (V & + & V\,) & \Rightarrow & Z \\
0 & & 1 & & 0 & & 0 & & 3 \\
 & & & & 0 & & 2 & & \\
0 & & 0 & & 8.0 & & 8.0 & & 8.0
\end{array}$$

Notice that brackets open and close in the upper line.

Statements are written using a line. A block of statements is marked as a unit by enclosing it in square brackets, which are as large as needed, for example:

$$\left[\begin{array}{lllll}
Z & + & 2 & \Rightarrow & Z \\
0 & & & & 2 \\
 & & & & \\
8.0 & & & & 8.0 \\
 & & & & \\
Z & \times & Z & \Rightarrow & Z \\
2 & & 1 & & 3 \\
 & & & & \\
8.0 & & 8.0 & & 8.0
\end{array}\right]$$

This is a block of two instructions, an addition and a multiplication.

Conditions can be tested with the operators =, >, <, which are used to check if the first argument is equal, larger, or smaller that the second. Any two structures can be tested for equality, but only structures which can interpreted as numbers (*n* bits) can be tested using the other two operators. We can store the larger of two numbers Z1 and Z2 in Z3 using the following instructions:

Z ⇒ Z
1 3

8.0 8.0

Z < Z → Z ⇒ Z
1 2 2 3

8.0 8.0 8.0 8.0

2.4 Iterations

There is a kind of "While" statement which is useful for performing iterations. The syntax of the construct is

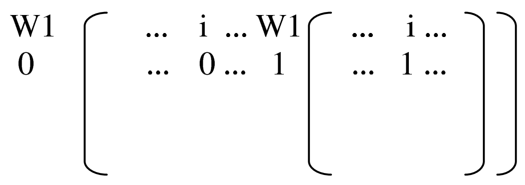
W [Block]

where [Block] denotes a block of statements. In general, an iterative construct has the form:

$$W \left(\begin{array}{l} C1 \rightarrow S1 \\ C2 \rightarrow S2 \\ \ldots \\ Cn \rightarrow Sn \end{array} \right)$$

The block is executed repeatedly until all conditions C1, C2, etc., tested inside the block, fail in a single run. The statements S1, S2, etc. are executed according to the truth value of their respective conditionals.

The construct W0(num) preceding a block of instructions can be interpreted as a "FOR" operation: the block of instructions is executed "num" times. If we want to have access to a loop variable containing the current iteration number, the construct W1(num) is used. A loop variable *i* runs from 0 to num-1. The loop variable is a special variable with an unspecified default numeric type and can be accessed only inside the block following the W1 declaration. If nested loops are used, they are numbered using the index row and their loop variables use also these numbers.



In the example above the first loop has index 0, the second index 1. The loop variables are i0 and i1. They can only be used within the scope of the respective While loops.

Zuse defined a built-in function which is very helpful when processing arrays. The function N applied to a variable yields the number of components of the variable as result. See below for an example of its application.

2.5 Linearized form of the Plankalkül

In order to simplify the rest of the paper, we adopt a linearized form of the Plankalkül in which variables are written as in the following examples:

V0[1 :5.0]	Variable V0, component 1, of type 5.0
Z1[5.3 :9.0]	Variable Z0, component 3 of component 5, of type 9.0

Some special symbols are written using ASCII characters or combinations thereof. Conjunction, disjunction and negation are expressed using the characters "&", "|" and "!". Assignment is expressed using "=>" and the conditional arrow is written "->".
The conditional expression written as the last example in section 2.3 can be simplified by writing:

```
Z0:0 & Z1:0 -> V0[0 :8.0] + V1[2 :8.0] => Z3[:8.0]
```

This is much more convenient than the four line syntax of the Plankalkül draft.

We will use square brackets to enclose blocks of instructions, and the semicolon as separator between statements to write more than one statement in each line.

An illuminating example is to compute the sum of the bytes stored in an array V0 of type 16.8.0. The following instructions would accomplish this task:

```
0 => Z1[:8.0]
W1(16) [ Z1[:8.0] + V0[i :8.0] => Z1[:8.0] ]
```

Notice that we don’t write the type of the constant 16 and the type of the loop variable. They are generic numeric variables.

2.6 Functions and function calls

Programs in Plankalkül are functions which can be called from other programs. Every program is assigned a unique number. The declaration of input and output variables is done in the “Randauszug” (“boundary summary”, i.e. the program interface), which, for example, has the following form:

$$\begin{array}{llllll} \mathrm{P3} & \mathrm{R}\,(\mathrm{V}, & \mathrm{V}\,) & \Rightarrow & (\mathrm{R}\,, & \mathrm{R}\,) \\ & 0 & 1 & & 0 & 1 \\ & 8.0 & 8.0 & & 4.0 & 4.0 \end{array}$$

In this example program P3 is defined with two input variables V0 and V1, each of 8 bits, and two output variables R0 and R1, each of four bits.

The number of input and output variables in the Randauszug is variable. Zuse always numbered the input variables from 0 to *n*-1 and the output variables from 0 to *m*-1, where *n* and *m* are the number of input and output variables respectively. The identifier for the program in the example above is just an “R”. When this program is called as function, we write “R3” with the two variable arguments enclosed in parenthesis. Notice that in the Randauszug, the input and output variables are written without the component row, that is, they can only be used as *complete* variables.

Functions can also have a symbolic identifier. For example, if we want to define a function to select the maximum of two bytes, we could write the following Randauszug

$$\begin{array}{lllll} \mathrm{P5} & \mathrm{Max}\,(\mathrm{V}, & \mathrm{V}\,) & \Rightarrow & \mathrm{R} \\ & 0 & 1 & & 0 \\ & 8.0 & 8.0 & & 8.0 \end{array}$$

followed by the appropriate block of instructions. We will write the Randauszug in the following linearized form:

P5 Max (V0:8.0,V1:8.0) ⇒ R0:8.0

When a function is called with a subindex we select the component of the output we are interested in:

$$\begin{array}{lll} \mathrm{R3} & (\mathrm{Z}\,, & \mathrm{Z}) \\ 0 & 1 & 3 \\ & & \\ 4.0 & 8.0 & 8.0 \end{array}$$

This is a call to program P3 above which computes two result variables, R0 and R1. In the call we select variable R0 of type 4.0 from the result tuple. In linearized form we write for this call: R3(Z1:8.0, Z3:8.0)[0]:4.0 ⇒ Z4:4.0.

Finally, although Zuse did not signal the end of a program with any special keyword, we will write END at the end of every program.

2.7 Input and Output

Zuse did not define any primitive instructions for input and output. He seems to have considered this type of instructions machine specific and they do not belong to the main language constructs. In our implementation of the language we did not define any input/output instructions. The user can inspect and modify the state of the variables stored in memory by opening a memory window, which is specific for every program, since Plankalkül variables are local.

The example below shows a program which computes the maximum of three variables by calling the function max.

```
P1  max3 (V0[:8.0],V1[:8.0],V2[:8.0]) => R0[:8.0]

        max(V0[:8.0],V1[:8.0]) => Z1[:8.0]
        max(Z1[:8.0],V2[:8.0]) => R0[:8.0]
END

P2  max  (V0[:8.0],V1[:8.0]) => R0[:8.0]

        V0[:8.0] => Z1[:8.0]
        (Z1[:8.0] < V1[:8.0]) -> V1[:8.0] => Z1[:8.0]
        Z1[:8.0] => R0[:8.0]
END
```

2.8 Implementation problems

There are several syntactical aspects of Plankalkül which must be dealt with in any implementation. The programmer writes the type of its variables every time they are used and this can lead to inconsistencies. We decided that any variable in a program has a unique type, which cannot be "casted" into another type. This leaves open the possibility of writing the type of a variable only once and use type inference at any other point in the program where it is used. However, type inference is made a little more difficult because we can refer to a component of a variable before we refer to the variable proper. For example, a reference to V0[1.1]:8.0 can appear in a statement before a reference to V0:16.8.8.0. Therefore, the type inference routine has to look at the whole program before deciding on the type of a variable. We did not implement type inference in the first implementation of Plankalkül 2000, so that the programmer has to write the type of any variable or component that is used.

We decided to refer to components of variable calls by using square brackets in the linearized form of the Plankalkül. We decided also, that the type of the result must be always written. Zuse did not usually write the type of the result of a variable call, since it can be inferred from the definition of the language. In the first implementation of Plankalkül 2000 we always write the type of variables or variable calls.

3. **The editor**

The two-dimensional syntax of Plankalkül is very difficult to deal with using a conventional editor. Therefore, we developed a syntax directed editor which allows the user to write a program by selecting options from menus. Fig.1 shows the start window of the editor. By clicking on the "Plan" keyword it is possible to obtain a layout for a new program, which includes the program number, the Randauszug and the instructions. Fig. 2 shows the state of the window after several selection steps. The user has selected program number 1 from a menu and 4 and 3 variables for the input and output in the Randauszug. The user can now select an instruction from the options below ("Befehl"), a new block of instructions ("Block") or end the sequence of instructions ("FIN").

Fig.1: Start window of the syntax directed editor

Basically, the options offered by the menus are the only ones valid under the defined grammar. One can think of this syntax-directed editor as one which only allows the user to develop the Backus-Naur form for the programming language we are considering. In this respect, this editor is similar to those written by Teitelbaum and Reps [1981], by Arefi et al. [1990], and others. The user is restricted to stay within the boundaries of the Plankalkül grammar and cannot write invalid programs. Any program written with this editor is grammatically correct, although, of course, it can be semantically incorrect.
The syntax directed editor produces a linearized version of the Plankalkül program. In this way, the programmer can directly write his program in linear form using any kind of editor, or he can use this two-dimensional editor to produce the linear code. Although both possibilities are open to the programmer, the syntax directed editor should convey to the user the "look and feel" of writing Plankalkül programs in the original syntax.

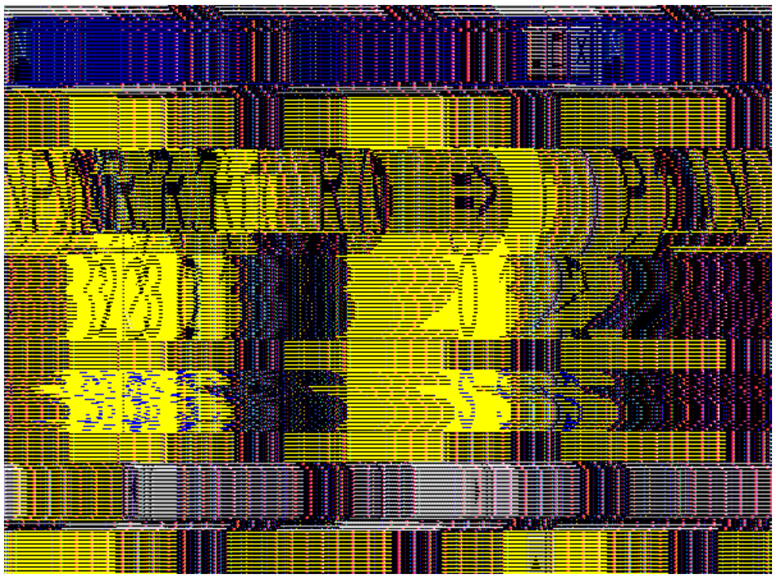


Figure 2: State of the editor after some selections

The editor is written in Java and will be installed as an Applet at our Web site in the future.

4. **The parser**

The syntax of Plankalkül 2000 is summarized in Appendix A. We wrote a parser for this syntax using the public domain version of the Cocktail compiler generator system [Grosch, Emmelman, 1990]. The figure below shows the structure of the whole Plankalkül system: the syntax directed editor transforms the two-dimensional code into the linearized version of Plankalkül described in this document. The parser then transforms this code to a simpler textual representation of the program which we call the "intermediate code". This intermediate code is then interpreted by the run-time system. This allows the user to set the values of variables through an interactive user interface.

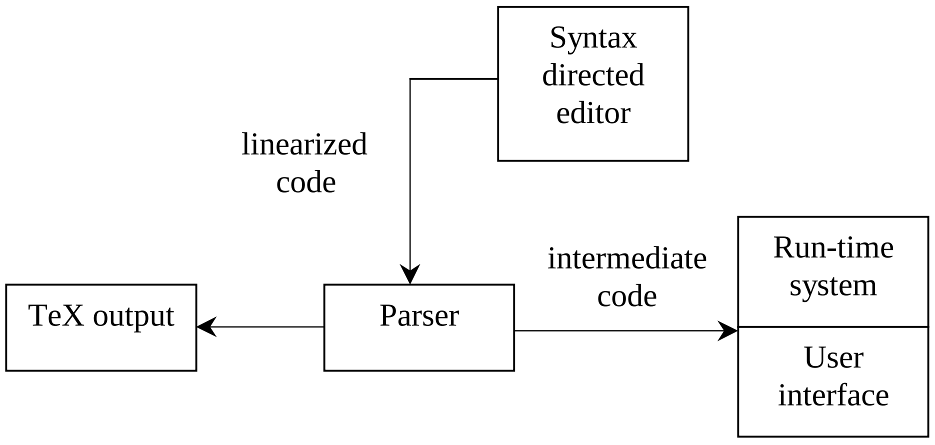


Figure 3: Strutcture of the Plankalkül System

The parser produces not only the intermediate code for the run-time system, but also TeX code that can be interpreted and sent to a PostScript printer.

5. **The run-time system**

The run-time system was written in Java. When the system starts, a window shows the contents of the memory variables. This can be changed interactively by the user. Figure 4 is an example of the state of the memory after a run of a sorting program. The first row in the window shows an array of five numbers, each of 8 bits. The ones are shown as full circles (the lowest order bits are written to the left). The decimal equivalent is written below each element of the array. The last row shows the result of the sorting routine. The rows in the middle are intermediate (Z) variables.

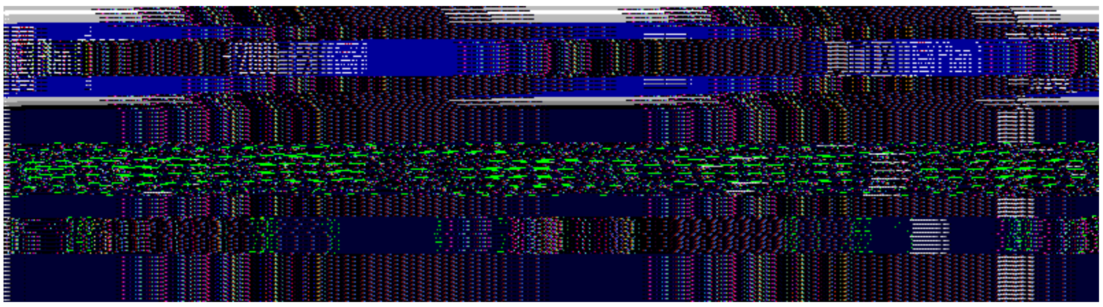


Figure 4: The result (last row) of sorting the V variables (first row)

Before the program starts, the user can modify the values of the V variables by clicking on the individual bits. After the program runs, the user can inspect the result variables. The original definition of Plankalkül does not include any input-output instructions. Zuse left this part of the language undefined. Curiously, this is also the case with modern programming languages in which input and output routines are defined in a special system library.

6. Example programs

In this section we provide some examples of Plankalkül 2000 programs written as linear code. All these programs have been parsed and executed by our system.

Program P1 assigns the conjunction of two input variables to the result variable.

```
P1    R(V0[:0],V1[:0]) => R0[:0]

      V0[:0] & V1[:0] => R0[:0]

END
```

Program P2 computes the expression a+b*c.

```
P2    R(V0[:16.0],V1[:16.0]) => R0[:16.0]
      V0[:16.0] + V1[:16.0] * V1[:16.0] => R0[:16.0]
END
```

A variation of the program above (to test syntactic alternatives).

```
P3    R(V0[:16.0],V1[:16.0]) => R0[:16.0]
      (V0[:16.0] + V1[:16.0]) * V1[:16.0] => R0[:16.0]
END
```

Another variation.

```
P4    R(V0[:16.0],V1[:16.0]) => R0[:16.0]
      (V0[:16.0] * 6)+(V1[:16.0]*V1[:16.0]) => R0[:16.0]
END
```

Program P5 computes the factorial of 5 (the generic type is 32.0):

```
P5    R(V0[:32.0]) => R0[:32.0]
      1 => Z0[:32.0]
      W1 (5)      [
                  i * Z0[:32.0] => Z0[:32.0]
                  ]
      Z0[:32.0] => R0[:32.0]
END
```

Program P6 sorts 16 numbers using insertion sort.

```
P6 sort (V0[:6.8.0]) => R0[:6.8.0]

W1[0](4)[
         V0[i0:8.0] => Z0[i0:8.0]
                  1 => Z4[:32.0]
         W1[1](i0)
          [
            (V0[i0:8.0] < Z0[i1:8.0]) & (Z4[:32.0]=1) ->
            [
             i0-i1 => Z1[:32.0]
             W1[2](Z1[:32.0])
               [
               i0 - i2 - 1 => Z3[:32.0]
               i0 - i2     => Z2[:32.0]
               Z0[Z3[:32.0]:8.0] => Z0[Z2[:32.0]:8.0]
               ]
            V0[i0:8.0] => Z0[i1:8.0]
            0 => Z4[:32.0]
            ]
          ]
        ]
END
```

7. Conclusions

We have described in this paper the architecture of a compiler for a subset of the Plankalkül. The subset of Plankalkül that we have selected for this implementation is computationally universal. The user can write programs using a conventional editor and the linearized version of the language, or he/she can use the syntax-directed editor written in Java. This editor produces only syntactically valid programs.

Our future work will be concentrated in linking the individual system modules in a transparent way, so that the system appears as a monolithic piece of software. We will write also a front-end for the rest of the programming constructs of Plankalkül. Set and predicate logic instructions will be compiled into Plankalkül 2000 instructions. We will then be able to run the chess playing programs written by Konrad Zuse in 1945.

Appendix A

Syntax of the Plankalkül

In the following we adopt the following conventions: a vertical bar ("|") separates optional syntactical elements, {expr}* means that expr can be concatenated zero or more times, all identifiers with a defined rule can be expanded, any other characters are included literally in the expanded expressions.

```
--------- Symbols

digit ::= 0 | 1 | 2 | ... | 9
digits ::=digit {digit}*
letter ::= a | b | ...| A | B | .... | Z
type-letter ::= a | b | ...h | j | ... | A | B | .... | Z       \\ i.e. without the "i"
identifier ::= letter {letter | digit}*
pos-constant ::= digits
neg-constant ::= - digits
constant ::= pos-constant | neg-constant
dot ::= "."
comma ::= ","

---------- Data types

simple-type ::= 0
tuple-type ::= (type, type {comma type}*)
type ::= simple-type | tuple-type | digits dot type
var-type ::=  type-letter dot type
all-type ::= type | var-type

---------- Variables

v-variable ::=  V digits [component: type] | V digits[: all-type]
z-variable ::=   Z digits [component: type] | Z digits[: all-type]
r-variable ::=   R digits [component: type] | R digits[: all-type]

loop-var ::= "i" | "i" digits
loop-expr ::= loop-var | loop-var + pos-constant | loop-var – pos-constant}

type-var ::= type-letter
type-expr ::= type-var | type-var + pos-constant | type-var – pos-constant  | type-expr
+ type-expr | type-expr – type-expr

component ::= digits | v-variable |  z-variable | loop-expr | type-expr | component dot
component
```

```
---------- Function call

zv-call-arg ::= v-variable | z-variable | call | constant | loop-var | type-var

call-all ::=   R  digits [:type] ( zv-call-arg {,zv-call-arg}*)|
              identifier  [:type] ( zv-call-arg {,zv-call-arg}*)
call-one ::=  R digits  [component : type] ( zv-call-arg {,zv-call-arg}*) |
              identifier [component : type] ( zv-call-arg {,zv-call-arg}*)
call ::= call-all | call-one

---------- Arithmetical operations
arith-argument-left ::= v-variable | z-variable | constant | loop-var | type-var | call |
arith-operation | (arith-operation)
arith-argument-right ::= v-variable | z-variable | pos-constant | (neg-constant) | loop-
var | type-var | call | arith-operation | (arith-operation)
arith-argument ::= arith-argument-left | arith-argument-right
arith-operation ::= arith-argument-left {+ | - | × | / } arith-argument-right

---------- Logical operations

log-constant ::= + | -
condition ::=   arith-argument =  arith-argument |
                arith-argument >  arith-argument |
                arith-argument < arith-argument  |
                zv-tuple = zv-tuple
pos-literal ::= v-variable | z-variable | log-constant | call | condition | (condition)
neg-literal ::= !v-variable | !z-variable | !call  | ! (condition)
logic-argument ::= pos-literal | neg-literal | logic-operation | (logic-operation)
logic-binary ::= logic-argument { "|" | & | ~ | /~} logic-argument
logic-operation ::= pos-literal | neg-literal | logic-binary | !(logic-binary)

---------- Assignment

assignment0 ::= arith-argument  => {z-variable | r-variable}
assignment1 ::= logic-argument => {z-variable | r-variable}
assignment2 ::= zv-tuple => zr-tuple
assignment3 ::= zv-tuple => {z-variable | r-variable}

zv-tuple ::= ( zv-arg, zv-arg {comma zv-arg}*)
zv-arg ::= v-variable | z-variable | constant | call | loop-var | type-var | zv-tuple
zr-tuple ::=  ( zr-arg, zr-arg {comma zr-arg}*)
zr-arg ::=  r-variable | z-variable | zr-tuple

assignment ::= assignment0 | assignment1 | assignment2 | assignment 3
```

```
---------- IF-THEN

if-then ::= logic-argument ->  statement

---------- While

block ::= [ statement{; statement}*]
while ::= w block | w  [digits]  block | w1 (arith-arg) block | w1[digits] (arith-arg)
block

---------- Statements

built-ins ::= FIN | FIN digits
statement ::= assignment | if-then | while | block | built-ins


---------- Programs

program ::= P digits    randauszug    {statement }*  END


---------- Randauszug


randauszug ::= identifier  v-tuple => r-tuple

v-tuple ::= v-variable | (v-variable {, v-variable}*)
r-tuple ::=  r-variable | (r-variable {,  r-variable}*)

// The variables are numbered sequentially, starting with 0


// constant, indices, N(), have generic type
```

Appendix B

Plankalkül 2000: Intermediate Code

The run-time system receives an array of strings with the following meaning:

Line	*Meaning*	*Contents*
0	Program number	integer ≥ 0
1	Program identifier	string
2	number of V variables	integer ≥ 0
.	TypID's	see below
.	number of Z variables	integer ≥ 0
.	TypID's	see below
.	number of R variables	integer ≥ 0
.	TypID's	see below
.	number of loop variables	integer ≥ 0
.	<Plan>	see below
.	END	»END« marks the end of a program

Type identifiers are sorted in the order of the variables, a new type in every line. The leaves of the type tree must be explicitly marked using a "1", for example as in 8.1, 4.2.1 or 3.2(2 3.1,4.5.1)

```
TypID            ::= <numl>[.<numl>]*|
      [<numl>][.<numl>]*('_'<num>'_'<numl>[.<numl>]*{,<TypID>})

numl                   ::= <num> | <type-letter>

type-letter    ::= a|b|...h|j|...|z


<Plan>         ::= [<planline>]*
<planline>     ::= <statement>'\n'
<statement>    ::= <assignment> | <if-then> | <while> |
                         <w1> | <wx> | <wd> | <built-ins>
<assignment>   ::= ='_'<term>'_'<factor>
<if-then>      ::= ?'_'<term>{'_'<statement>}'_'$
<while>        ::= W'_'{<statement>'_'}M
<wd>           ::= WD'_'<num>'_'{<statement>'_'}DW
<w1>           ::= W1'_'<term>'_'{<statement>'_'}1W
<wx>           ::= WX'_'<num>'_'<term>'_'{<statement>'_'}XW
<built-ins>    ::= FIN'_'<num>
<call>         ::= <call-r>  |  <call-i>
<call-r>       ::= C'_'R'_'<num>'_'{<term>'_'}'_'<typ>'_'-|<num0>
<call-i>       ::= C'_'I'_'<string>'_'{<term>'_'}'_'<typ>'_'-|<num0>
<term>         ::= <factor>| <op>'_'<factor>['_'<factor>]| <call>|
                 ('_'<num>'_'<term>{'_','_'<term>}'_')
<factor>       ::= <var>|<index>|<typevar>|<const>|<term>
<var>          ::= V|Z|R'_'<num0>'_'<component>
<const>        ::= K'_'[-]<num0>
<index>        ::= I'_'<num0>
```

```
<typevar>       ::= T'_'<letter>
<component>     ::= <factor>['_'<factor>]*'_'.
<op>            ::= +|*|-|/|==|<|>|!|&|'|'|x|nx
<string>        ::= {<letter>|<num0>}
<letter>        ::= a|b|...|z|A|B|...|Z
<num>           ::= natural number
<num0>          ::= natural number or 0
```

References

Arefi, F., Ch. Hughes, and D. Workman, “Automatically Generating Visual Syntax-Directed Editors”, *Communications of the ACM*, Vol. 33, N. 3, March 1990, 349-360.

Grosch, J., *Generators for High-Speed Front-Ends*, LNCS, 371, Springer-Verlag, 1988, pp. 81-92.

Grosch, J. and H. Emmelmann, "A Tool Box for Compiler Construction", LNCS, 477, Springer-Verlag, 1990, pp. 106-116

Rojas, R. (Hrsg.), *Die Rechenmaschinen von Konrad Zuse*, Springer-Verlag, 1998.

Teitelbaum, T., and T. Reps, “The Cornell Program Synthesizer: A Syntax Directed Programming Environment”, *Communications of the ACM*, Vol. 24, N. 9, Sept. 1981, pp.563-573.

Zuse, Konrad, “Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relaisschaltungen” , unpublished manuscript, Zuse Papers 045/018, 1943.

Zuse, Konrad, *Der Plankalkül,* Berichte der Gesellschaft für Mathematik und Datenverarbeitung, Nr. 63, Sankt Augustin, 1972.