copies of software. However, the Internet also creates an opportunity for a technological solution to the piracy problem. With the Internet, many kinds of businesses in addition to software companies have a need for a secure, nonreplicable way to transfer digital files, such as cash equivalent transactions or digital signatures.

In April 2000, **Xerox** and **Microsoft** announced that they were creating a company to develop software that would allow the electronic distribution of copyrighted material such as software, music, videos, or documents while protecting it against unauthorized copying. Several other companies, including IBM and **AT&T**, are also investing in research to develop a reliable means of encoding digital files so that they cannot readily be duplicated. Given the difficulty of enforcing the current law, an effective technological solution that would substantially reduce the dollars lost to piracy would be a great boon to the software industry.

FURTHER READING

1999 Global Software Piracy Report: A Study Conducted by International Planning and Research Corp. for the Business Software Alliance and the Software and Information Industry Association. West Chester, Penn.: International Planning and Research Corporation, May 1999.

Keet, Ernest E. *Preventing Piracy: A Business Guide to Software Protection.* Reading, Mass.: Addison-Wesley, 1985.

Software and Information Industry Association. *SPA Anti-piracy, a Division of SIIA.* Washington, D.C.: SIIA, 2000. *http://www.siia.net/piracy/default.asp*

Tetzlaff, David. "Yo-ho-ho and a Server of Warez: Internet Software Piracy and the New Global Information Economy." Talk delivered at the World Wide Web and Contemporary Cultural Theory Conference, Drake University, Des Moines, Iowa, 7 Nov. 1998. *http://www.drake.edu/swiss/webconference/tetzlaff.html*

—Luanne Johnson

PGP See Pretty Good Privacy.

Plankalkül

Plankalkül (*calculus of programs*) was the first high-level programming language ever conceived. It was designed by **Konrad Zuse** (1910–95), the German inventor, between 1943 and 1945, a time when the first computers were being built in the United States, United Kingdom, and Germany. It represents one of the major contributions to the history of ideas in the computer field, although it was never implemented for any kind of machine.

Plankalkül corresponds to Zuse's mature conception of how to build a computer and how to allocate the total computing work to the **hardware** and **software** of a machine. Zuse called the first computers he constructed (the Z1, Z2, Z3, and Z4) *algebraic machines*, in contrast to *logistic machines*. The first were specially built to handle scientific computations, the latter could deal not only with scientific but also with symbolic processing. Zuse's *logistic machine* was never built, but its design called for a one-bit word memory and a processor that could compute only the basic logic operations AND, OR, and NOT. It was a sort of minimal machine. Since the memory consisted of a long chain of bits, they could be grouped in any desired form to represent numbers, characters, arrays, and so on.

Plankalkül was to be the software counterpart of the logistic machine. Complex structures could be built from elementary ones, the simplest being a single bit. Sequences of instructions could be grouped into subroutines and functions, so that the user had only to deal with a very abstract instruction set that masked the complexity of the underlying hardware. Plankalkül exploited the concept of modularity, so important today in computer science, almost in an extremist way: Several layers of software would make the hardware transparent for the programmer. The hardware itself was able to execute only the absolutely minimal instruction set.

In Plankalkül, the programmer uses variables to perform computations. The notation is such that intermediate results are labeled Z1, Z2, Z3, and so on. Input variables are labeled V1, V2, V3, and so on, and results are labeled R1, R2, R3, and so on. To describe a variable and its type, Zuse used the *row notation*:

	Z
V	1
K	2
S	5.0

These four lines define the variable Z1 (note that the index is written in the next line, the V line), with *structure* 5.o, that is, five times structure "o," which represents a single bit. The K line tells us which component is being referred to. In this case we refer to the second bit of the five-bit field Z1. Therefore, the notation is two-dimensional, although it could be compressed on a single line. In a modern programming language, we would write Z1[2]. There are no separate variable declarations; any variable can be used in any part of the program and its type is written together with the name.

The type of a variable could be selected in a very flexible way. The only primitive type was "o" (a bit). A group of n bits was denoted as n.o, a group of m n-bit numbers as $m.n$.o, and so on. Any kind of primitive data type (characters, integers, reals), as well as vectors and matrices, could be defined in this way. A data type could be abbreviated using another letter, and this letter could be used as a building block for another composite type.

Variable assignment was to be done as in modern programming languages: The new value overwrites the old value of a variable. There are several operations that are also used in ways similar to other programming languages (addition, subtraction, etc.). The addition of two variables V1 and V2 (eight bits each) can be stored in an intermediate variable Z1 using the following piece of code:

	V	+	V	⇒	Z
V	1		2		1
K	1		3		1
S	5.8.0		5.8.0		5.8.0

In Pascal, we would just write Z1[1]:=V1[1]+V2[3]. Note that the variables V1, V2, and Z1 have the same type: an array of five numbers of eight bits. The programmer has to see to it that the assignments refer to variables of the same type, since there is no type checking.

Arrays of objects can be indexed by using an auxiliary variable. The use of the index variables is shown using a line:

	V		V	⇒	Z
V	1	⌐	2		1
K	⌙		2		1
S	5.8.o		5.8.0		8.0

In this example, the second component of the array V2 contains the index for the array V1. The number is copied to the first component of Z1. In Pascal we would write Z1[1]:=V1[V2[2]].

Boolean operations produce results that are single bits. The zero is interpreted as FALSE and the 1 as TRUE. Boolean results can be used in conditional instructions. Plankalkül could work with conditional instructions of the If-Then-Else type, which would be written as guarded instructions of the form A → B. If the guard A is true, the command B is executed. Blocks of instructions could be written in Plankalkül by separating each instruction with a vertical line or by writing the instructions one under the other. A block is enclosed in parentheses. A block counts later as a single instruction and can be made part of another block.

There is also an iterative operator W, which repeats the execution of a sequence of instructions until all guards in the body of the loop fail:

W	A	→	B
	C	→	D
	E	→	F

Here, the scope of the W covers the three guarded instructions, which form a block. The loop is repeated if any of the guards are true. Execution of the loop is terminated when the three guards A, C, and E fail within the same iteration.

The elementary Boolean and arithmetic operations, guarded commands, and the W control structure formed the basis of Plankalkül. Other control structures and commands could be built using them. There was, for example, a W1 control structure that would correspond to the FOR command in a modern programming language—that is, an iteration that is performed a certain number of times. There were also other more specialized constructions that employ quantors (there exists an x such that, for all x, etc.) but they

could be expressed also using the basic elements mentioned above. Zuse never built a compiler or interpreter for Plankalkül, but it seems that he was well aware that the more complex portions of Plankalkül could be written using the basic commands.

Subroutines and functions could be written in Plankalkül. A declaration was put in front of the code to make it clear which variables were the arguments and which the results. This declaration was the *boundary summary* (*Randauszug*) of the procedure. It was also possible to give operators as arguments. A subroutine could be written, for example, that received as argument the operator "+" or the operator "×," so that the same general code could be compiled with a different operator in the body of the routine. One complication of this scheme was the absence of a clear distinction between local and global variables. Most of Zuse's draft of 1945 deals only with global variables, but he also indicates that variables in different programs can have the same name but refer to different memory localities. However, subroutines could also be used as functions: Kla(x), in an example given by Zuse, was a function that checks if a character x is an opening parenthesis and returns a Boolean value.

Although Zuse published some small papers about the Plankalkül and tried to make it known in Germany, the language never was implemented. The main obstacles were its ambitious scope, the large variety of instructions that it contained, its modular architecture, which called for incremental compilation, and the availability of dynamical structures and functionals. Also, some aspects of the semantics are not quite clear and the absence of type checking would have made it extremely difficult to debug. A practical implementation of Plankalkül would certainly require a major revision of Zuse's draft of 1945. However, Plankalkül was way ahead of its time and many of the concepts on which it was based were only rediscovered much later. In the case of Plankalkül, Konrad Zuse suffered the same fate as **Charles Babbage** (1791–1871) and the **Analytical Engine**: Babbage had the right concepts but the wrong hardware. After 1945, many more years would be needed until programming languages could achieve the level of sophistication of Plankalkül.

FURTHER READING

Zuse, Konrad. *Der Plankalkül.* Technical Report 63. Bonn, Germany: Gesellschaft für Mathematik und Datenverarbeitung, 1972.

—*Raúl Rojas*

PL/1

Programming Language 1 (PL/1) was developed by **IBM** in the early 1960s. It was first released as an application development language for the System/360 operating system in 1964. IBM promoted PL/1 as a general-purpose language—a successor and replacement for **Fortran** and **COBOL**.

PL/1 was designed to be an ideal match for structured programming, a programming paradigm based on hierarchical decomposition that was to exert great influence on programming practices throughout the 1970s and early 1980s. Each of the control flow constructs used in structure code design are represented directly by PL/1 statements: loops, conditionals, and case selection. PL/1 also supports the GOTO statement, although use of GOTO was discouraged in many PL/1 programming texts.

Because it was intended for a wide variety of programming tasks, PL/1 is an extensive language. It is block structured and supports packages, procedures, and functions. It also supports many different data types and structures: numeric types, arrays, records, character strings, bit strings, and references. Most of the PL/1 concepts were drawn from other languages: block structure and recursive subroutines from **ALGOL**, common blocks and parameter transmission from Fortran, and formatted I/O and records from COBOL. The PL/1 features are more comprehensive and flexible than those in the original languages. For example, PL/1 supports a very extensive complement of record- and text-oriented input and output facilities, with provision for Fortran-style formats and COBOL-style picture specifications. Like **APL**, also developed at IBM, PL/1 supports applying arithmetic operations to entire arrays. In this case, however, the facilities in PL/1 fall far short of those in APL.

New concepts introduced in or substantially refined by PL/1 include type-parameterized (generic)