Zürich, 2. 2. 2005 / 15. 6. 2005

# Good Ideas, Through the Looking Glass

Niklaus Wirth

**Abstract**

An entire potpourri of ideas is listed from the past decades of Computer Science and Computer Technology. Widely acclaimed at their time, many have lost in splendor and brilliance under today's critical scrutiny. We try to find reasons. Some of the ideas are almost forgotten. But we believe that they are worth recalling, not the least because one must try to learn from the past, be it for the sake of progress, intellectual stimulation, or fun.

**Contents**

## 1. Introduction

The history of Computing has been driven by many good and original ideas. Not only a few of them turned out to be less brilliant than they appeared at the beginning. In many cases, and this is typical for technology, their importance was reduced by changes in the technological environment. Often also commercial factors influenced the importance of a good idea. And some ideas simply turned out to be less effective and less glorious, when reviewed in retrospect, or seen after proper analysis. Other ideas turned out to be reincarnations of ideas invented earlier and then forgotten, perhaps because they were ahead of their time, perhaps because they had not flattered current fashions and trends. And some ideas were reinvented, although they had already been bad in their first round.

This led me to the idea of collecting a number of such good ideas which turned out to be less than brilliant in retrospect. The motivation to do so has been triggered by a recent talk of Charles Thacker about *Obsolete ideas* – ideas deteriorating by aging. I also rediscovered an article by Don Knuth titled *The dangers of computer-science theory*. Thacker's talk was delivered behind the Chinese Wall, Knuth's in Romania in 1970 behind the Iron Curtain, both safe places against damaging "Western Critique". Particularly Knuth's document with its tongue in cheek attitude encouraged me to write down this collection of stories. No claim is made for its completeness.

The collection starts with a few unsuccessful concepts from the area of computer technology, and then proceeds with questionable ideas in computer architecture. Some of them can well be justified in the context of their time, but are obsolete in the present. Then follows a collection of ideas from the realm of programming languages, particularly from the early proponents. And finally, we mention miscellaneous topics from programming paradigms to software (compiler) technology.

## 2. Hardware Technology

Speed has always been the prevalent and almost exclusive concern of computer engineers. Refining existing techniques has been one alley to pursue this goal, looking for alternative solutions the other. Some of these searches were unsuccessful, and here are a few of them.

During the times when magnetic core memories were dominant, the idea of the *magnetic bubble memory* appeared. As usual, great hopes were connected to it. It was planned to replace all kinds of mechanically rotating devices, which were the primary sources of troubles and unreliability. Although magnetic bubbles would still rotate in a magnetic field within a ferrite material, there would be no mechanically moving part. Like disks, they were a serial device. But they never achieved sufficient capacity, and the progress in disk technology was such that not only capacity, but also the speed of the bubbles became inferior. The idea was quietly buried after a few years of research.

A new technology that kept high hopes alive over decades was that of *cryogenic devices*, particularly in the domain of supercomputers. Ultra high switching speeds were

promised, but the effort to operate large computing equipment at temperature close to absolute zero was prohibitive. The appearance of personal computers working on your table let cryogenic dreams either evaporate or freeze.

Then there was the idea of using *tunnel diodes* in place of transistors as switching and memory elements. The tunnel diode (so named because of its reliance on a quantum effect of electrons passing over an energy barrier without having the necessary energy) has a peculiar characteristic with a negative segment. This allows it to assume two stable states. The tunnel diode is a germanium device and has no counterpart on the basis of silicon. This made it work over a relatively narrow temperature range only. Silicon transistors became faster and cheaper at a rate that let researchers forget the tunnel diode.

The same phenomenal progress in silicon transistor fabrication never let gallium-arsenide transistors live up to the high expectations with which they were introduced. Silican bears the inherent advantage that its own oxyde is an ideal insulator, thus simplifying fabrication processes. Other materials such as gallium-arsenide – the 3-5 technology in general - are now rarely used in computer technology, but have their application in the niche of very high frequency communication.

It now even appears that the once dominant bipolar transistor is counting its days, as field effect transistors become ever faster, smaller and cheaper, because it has become possible to grow extremely thin oxyde layers (gates). However, this does not mean that the bipolar transistor had been a bad idea.

## 3. Computer Architecture

### 3.1. Representation of Numbers

A rewarding area for finding "good" ideas is that of computer architecture. A fundamental issue at the time was the choice of representation of numbers, in particular integers. The key question was the choice of their base. Virtually all early computers, with the exception of Konrad Zuse's, featured base 10, that is, a representation by decimal digits, just as everybody was used to and had learnt in school.

However, it is clear that a binary representation with binary digits is much more economical. An integer $n$ requires $log_{10}(n)$ decimal digits in decimal and $log_2(n)$ binary digits (bits) in binary representation. Because a decimal digit requires 4 bits, decimal representation requires about 20% more storage than binary, showing the clear advantage of the binary form. Yet, the decimal representation was retained for a long time, and even persists today in the form of library modules.

The reason is that people insisted in believing that all computations must be accurate. However, errors occur through rounding, for example after division. The effects of rounding may differ depending of the number representation, and a binary computer may yield different results than a decimal computer. Because traditionally financial transaction – and that is where accuracy matters! – were computed by hand with decimal arthmetic, it was felt that computers should produce the same results in all cases, in other words, commit the same errors.

Although the binary form will in general yield more accurate results, the decimal form remained the preferred form in financial applications, as a decimal result can easily be

hand-checked if required. Although perhaps understandable, this was clearly a conservative idea. It is worth mentioning that until the advent of the IBM System 360 in 1964, which featured both, binary and decimal arithmetic, manufacturers of large computers kept two lines of products, namely binary computers for their scientific customers, and decimal computers for their commercial customers. A costly practice!

In early computers, integers were represented by their magnitude and a separate sign bit. In machines which relied on sequential addition digit by digit, the sign was placed at the low end in order to be read first. When bit parallel processing became possible, the sign was placed at the high end, again in analogy to the notation commonly used on paper. However, using a sign-magnitude representation was a bad idea, because addition requires different circuits for positive and negative numbers. Representing negative integers by their complement was evidently a far superior solution, because addition and subtraction could now be handled by the same circuit. Some designers chose 1's complement, where $-n$ was obtained from $n$ by simply inverting all bits, some chose 2's complement, where $-n$ is obtained by inverting all bits and then adding 1. The former has the drawback of featuring two forms for zero (0…0 and 1…1). This is nasty, particularly if available comparison instructions are inadequate. For example, in the CDC 6000 computers, there existed an instruction testing for zero, recognizing both forms correctly, but an instruction testing the sign bit only, classifying 1…1 as a negative number, making comparisons unnecessarily complicated. This was a case of inadequate design, revealing 1's complement as a bad idea. Today, all computers use 2's complement arithmetic.

| decimal | 2's complement | 1's complement |
|---|---|---|
| 2 | 010 | 010 |
| 1 | 001 | 001 |
| 0 | 000 | 000 or 111 |
| -1 | 111 | 110 |
| -2 | 110 | 101 |

Numbers with fractional parts can be represented by fixed or floating point forms. Today, hardware usually features floating-point arithmetic, that is, a representation of a number $x$ by two integers, an exponent $e$ and a mantissa $m$, such that $x = B^e \times m$. For some time, there was discussion about which exponent base $B$ should be chosen. The Burroughs B5000 introduced $B = 8$, and the IBM 360 used $B = 16$, both in 1964, in contrast to the conventional $B = 2$. The intention was to save space through a smaller exponent range, and to accelerate normalization, because shifts occur in larger steps of 3 (or 4) bit positions only. This, however, turned out to be a bad idea, as it aggravated the effects of rounding. As a consequence, it was possible to find values x and y for the IBM 360, such that, for some small, positive $\varepsilon$, $(x+\varepsilon)\times(y+\varepsilon) < (x \times y)$. Multiplication had lost its monotonicity! Such a multiplication is unreliable and potentially dangerous.

### 3.2. Data addressing

Instructions of the earliest computers consisted simply of an operation code and an absolute address (or a literal value) as parameter. This made self-modification by the program unavoidable. For example, if numbers stored in consecutive memory cells had to be added in a loop, the address of the add instruction had to be modified in each step by adding 1 to it. Although the possibility of program modification at run-time was heralded as one of the great consequences of John von Neumann's profound idea of storing

program and data in the same memory, it quickly turned out to enable a dangerous technique and to constitute an unlimited source of pitfalls. Program code must remain untouched, if the search for errors was not to become a nightmere. Program's self-modification was recognized as an extremely bad idea.

The solution was to introduce another addressing mode which allowed one to treat an address as a piece of variable data rather than (a part of) an instruction in the program, which would better be left untouched. The solution was *indirect addressing* and modifying the directly addressed address (a data word) only. Although this removed the danger of program self-modification, and although it remained a common feature of most computers until the mid 1970s, it should be considered in retrospect as a questionable idea. After all, it required two memory accesses for each data access, and hence caused a considerable slow-down of the computation.

The situation was made worse by the "clever" idea of *multi-level indirection*. The data accessed would indicate with a bit, whether the referenced word was the desired data, or whether it was another (possibly again indirect) address. Such machines (e.g. HP 2116) were easily brought to a standstill by specifying a loop of indirect addresses.

The solution lay in the introduction of *index registers*. To the address constant in the instruction would be added the value stored in an index register. This required the addition of a few index registers (and an adder) to the accumulator of the arithmetic unit. The IBM 360 merged them all into a single register bank, as is now customary.

A peculiar arrangement was used by the CDC 6000 computers: Instructions directly referred to registers only, of which there were 3 banks: 60-bit data (X) registers, 18-bit address (A) registers, and 18-bit index (B) registers. Memory access was implicitly evoked by every reference to an A-register (whose value was modified by adding the value of a B-register). The odd thing was that references to A0 – A5 implied the fetching of the addressed memory location into the corresponding X0 – X5 register, whereas a reference to A6 or A7 implied the storing of X6 or X7. Although this arrangement did not cause any great problems, it is in retrospect fair to classify it as a mediocre idea, because a register number determines the operation performed, i.e. the direction of data transfer. Apart from this, the CDC 6000 featured several excellent ideas, primarily its simplicity. It can truly be called the first RISC machine, although it was designed by S. Cray in 1962, well before this term became coined around 1980.

A much more sophisticated addressing scheme was invented with the Borroughs B5000 machine. We refer to its descriptor scheme, primarily used to denote arrays. A so-called *data descriptor* was essentially an indirect address, but in addition also contained index bounds to be checked at access–time. Although automatic index checking was an excellent and almost visionary facility, the descriptor scheme was a questionable idea, because matrices (multi-dimensional arrays) required a descriptor of an array of descriptors, one for each row (or column) of the matrix. Every n-dimensional matrix access required an n-times indirection. The scheme evidently did not only slow down access due to its indirection, but also required additional rows of descriptors. Nevertheless, the poor idea was adopted by the designers of Java in 1995 and of C# in 2000.

### 3.3. Expression Stacks

The language Algol 60 not only had a profound influence on the development of further programming languages, but - to a much more limited extent – also on computer architecture. This should not be surprising, as language, compiler and computer form an inextricable complex.

The first topic to be mentioned in this connection is the evaluation of expressions which, in Algol, could be of arbitrary complexity, with subexpressions being parenthesized and operators having their individual binding strengths. Results of subexpressions have to be stored temporarily. As an example, take the expression

(a/b) + ((c+d)*(c-d))

It would be evaluated in the following steps yielding temporary results *t1* … *t4*:

t1 := a/b; t2 := c+d; t3 := c-d; t4 := t2*t3; x := t1 + t4

F. L. Bauer and E. W. Dijkstra independently proposed a scheme for the evaluation of arbitrary expressions. They noticed that when evaluating from left to right, obeying priority rules and parentheses, the last item stored is always the first to be needed. It therefore could conveniently be placed in a push-down list (a stack):

t1 := a/b; t2 := c+d; t3 := c-d; t2 := t2*t3; x := t1 + t2

It was a straight forward idea to implement this simple strategy using a register bank, with the addition of an implicit up/down counter holding the index of the top register. Such a stack reduced the number of memory accesses and avoided the explicit identification of individual registers in the instructions. In short, stack computers seemed to be an excellent idea. The scheme was implemented by the English Electric KDF-9 and the Borroughs B-5000 computers. It obviously added to their hardware complexity.

How deep should such a stack be? After all, registers were expensive resources. The B-5000 chose to use 2 registers only, and an automatic pushdown into memory, if more than two intermediate results had to be stored. This seemed reasonable. As Knuth had pointed out in an analysis of many Fortran programs, the overwhelming majority of expressions required only 1 or 2 registers. Still, the idea of an expression stack proved to be rather questionable, particularly after the advent of architectures with register banks in the mid 1960s. Now simplicity of compilation was sacrificed for any gain in execution speed. The stack organization had restricted the use of a scarce resource to a fixed strategy. But now sophisticated compilation algorithms proved to utilize registers in a more economical way, given the flexibility of specifying individual registers in each instruction.

### 3.4. Variable length data

Computers oriented towards the data processing market typically featured decimal arithmetic, and also provisions for variable length data. Both texts (strings) and numbers (decimal digits) are predominant in these applications. The IBM 1401 has been clearly oriented towards string processing, and it was manufactured in very large quantities. It featured not an 8-bit word length – the prevalent character size in those times was 6 bits! - but a 9-bit byte (the word *byte* was not yet common either before 1964). One of the bits in each word served to mark the end of the string or number, and it was called the *stop-*

*bit*. Instructions for moving, adding, comparing and translating would process byte after byte sequentially and terminate when encountering a stop-bit.

This solution of treating variable length data seems to be a fairly bad idea. After all, it "wastes" 11% of storage for stop-bits, whether needed or not. Later computers let this problem be solved by software without hardware support. Typically, either the end of a string is marked by a zero byte, or the first byte indicates the length. Also here, the information about length requires storage, but only for strings, and not for every byte in the store.

### 3.5. Storing return addresses in the code

The subroutine jump instruction, invented by D. Wheeler, deposits the program counter value in a place, from where it is restored upon termination of the subroutine. The question is: "Where is that location to be?" In several computers, particularly minicomputers, but also the CDC 6000 main frame, a jump instruction to location *d* would deposit the return address at *d*, and then continue execution at location *d+1*:

mem[d] := PC+1; PC := d+1

This was definitely a bad idea for at least two reasons. First, it prevented a subroutine to be called recursively. Recursion was introduced by Algol and caused much controversy, because *procedure calls*, as subroutine calls were now called, could no longer be handled in this simple way, because a recursive call would overwrite the return address of the previous call. Hence, the return address had to be fetched from the fixed place dictated by the hardware and redeposited in a place unique to the particular incarnation of the recursive procedure. This "overhead" appeared to be unacceptable to many computer designers as well as users, and hence they resorted to declare recursion as undesirable, useless and forbidden. They did not want to notice that the difficulty arose because of their inadequate call instruction.

The second reason why this solution was a bad idea is that it prevented multiprocessing. Each concurrent process had to use its own copy of the code. This is a simple result from the mistake of not keeping program code and data separated. We leave aside the question of what would happen if an interrupt occurs after depositing PC and before reloading the PC register.

Some later hardware designs, notably the RISC architectures of the 1990s, accommodated recursive procedure calls by introducing specific registers dedicated to stack addressing, and to deposit return addresses relative to their value. We shall return to this topic later, but merely mention here that depositing the return address in one of the general purpose registers (presuming the availability of a register bank) is probably the best idea, because it leaves the freedom of choice to the compiler designer while keeping the basic subroutine instruction as efficient as possible.

### 3.6. Virtual addressing

Just as compiler designers had asked hardware architects to provide features catering to their needs, so did operating system designers put forward their favorite ideas. Such wishes appeared with the advent of multi-processing and time-sharing, the concepts that gave birth to operating systems in general. The guiding idea was to use the processor in

an optimal way, switching it to another program as soon as the one under execution would be blocked by, for example, an input or output operation. The various programs were thereby executed in interleaved pieces, quasi concurrently. As a consequence, requests for memory allocation (and release) occurred in an unpredictable, arbitrary sequence. Yet, individual programs were compiled under the premise of a linear address space, a contiguous memory block. Worse, physical memory would typically not be large enough to accommodate sufficiently many processes to make multi-processing beneficial.

The clever solution out of this dilemma was found in indirect addressing, this time hidden from the programmer. Memory would be subdivided into blocks or pages of fixed length (a power of 2). A (virtual) address would be mapped into a physical address by using a page table. As a consequence, individual pages could be placed anywhere in store and, although spread out, would appear as a contiguous area. Even better, pages not finding their slot in memory could be out placed in the backyard, on large disks. A bit in the respective page table entry would indicate, whether the data were currently on disk or in main memory.

This clever and complex scheme was useful in its time. But it displayed its difficulties and problems. It practically required all modern hardware to feature page tables and address mapping, and to hide the cost of indirect addressing – not to speak of disposing and restoring pages on disk in unpredictable moments – from the unsuspecting user. Even today, most processors use page mapping, and most operating systems work in the multi-user mode. But today it has become a questionable idea, because semiconductor memories have become so large, that the trick of mapping and outplacing is no longer beneficial. Yet, the overhead of indirect addressing and the complex mechanism still remain with us.

Ironically, the mechamism of virtual addressing keeps being used for a purpose for which it had never been intended: Trapping references to non-existant objects, against use of NIL pointers. NIL is represented by 0, and the page at address 0 is never allocated. this dirty trick is a misuse of the heavy virtual addressing scheme, and it should have been solved in a straight-forward way.

### 3.7. Memory protection and user classes

There is one concept which at first sight may still justify the mapping mechanism: *Protection.* Every system allowing several concurrent users must provide a safeguard against mutual interferences. Mutual protection of program and data of one user from the other users must be guaranteed. Such a scheme inherently requires a distinction between users with different rights, called privileges. In principle, two classes suffice, one for the "regular" user programs subjected to access restrictions, and the other for a "supervisor" program with unlimited access rights in order to be able to allocate and mark memory blocks to new user programs and to recycle them after a program had terminated. If a user program tried to access memory beyond its allocated part, presumably because of an error, then it would be trapped and control returned to the supervisor program which, supposedly, was free of errors. Access rights were easily registered in page tables.

On closer inspection one realizes that the need for protection and classes of programs arises from the fact that programs are possibly erroneous in the sense of issuing requests

for memory outside their allocated memory space, or accessing devices that should not be directly manipulated. If all programs were written in a proper programming language, this should not even be possible, because – if the language is correctly implemented – no reference to resources not named in the program would be possible. Protection must be implicit in compiled programs; it should be guaranteed by software.

Readers who doubt the sensibility of this postulate, must be reminded that modern programming systems rely on automatic garbage collection for storage management. A garbage collector requires exactly the same access safety as a multi-user system does. Without this safety, a garbage collector may destroy "accidentally" any information even in a single-user system at any moment. A truly safe system would have to require that all code be produced by correct compilers, and that they and all generated code would be immutable. We notice that this requirement almost completely renounces von Neumann's glorious idea of programs being accessible as data in their common store.

Hence, hardware protection appears as a crutch, covering only a small part of possible transgressions, and dispensable if software were implemented safely. It appears now as having been a good idea at its time that should have been superceded in the meantime.

### 3.8. Complex Instruction Sets

Early computers featured small sets of simple instructions, because they had to operate with a minimal amount of expensive circuitry. With hardware getting cheaper, the temptation rose to incorporate instructions of a more complicated nature, such as conditional jumps with three targets, instructions that incremented, compared, and conditionally branched all in one, or complex move and translate operations. With the advent of high-level languages, the desire arose to accommodate certain language constructs with correspondingly tailored instructions. A good example is Algol's **for** statement, or instructions for (recursive) procedure calls. Feature-tailored instructions were a smart idea, because they contributed to code density, which was important at a time when memory was a scarce resource, consisting of 64 KByte or less.

This trend had set in as early as 1963. The Burroughs B5000 machine not only accommodated many complicated features of Algol – more about it later – but it combined a scientific computer with a character string machine, it included two computers with different instruction sets. Such an extravagance had become possible with the technique of microprograms stored in fast read-only memories. This feature also made the idea of a computer family feasible: The IBM Series 360 consisted of a set of computers, all with the same instruction set and architecture, at least as far as it was visible to a programmer. However, internally the individuals differed vastly. The low-end machines were microprogrammed, the genuine hardware executing a short microprogram interpreting the instruction code. The high-end machines, however, implemented all instructions directly. This technology continued with single-chip microprocessors like Intel's 8086, Motorola's 68000, and National's 32000.

The NS processor is a fine example featuring a complex instructions set (CISC). Congealing frequent instruction patterns into a single instruction improved code density by a significant factor and reduced the number of memory accesses, increasing execution speed.

The NS processor accommodated, for example, the new concept of module and separate compilation with an appropriate call instruction. Code segments were linked when loaded, the compiler having provided tables with linking information. It is certainly a good idea to minimize the number of linking operations, which replace references to the link tables by adsolute addresses. The scheme, which simplifies the task of the linker, leads to the following storage organization for every module.

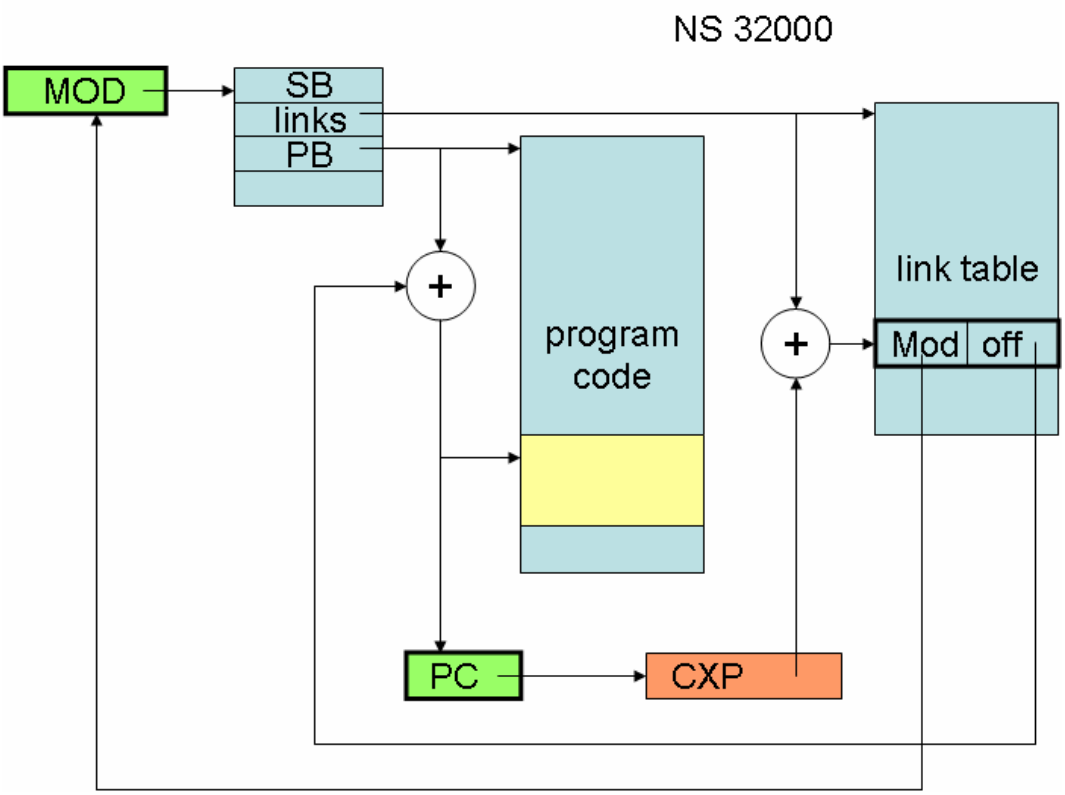


Fig. 1. Module layout in store

A dedicated register MOD points to the descriptor of module M, which contains the procedure P currently being under execution. Register PC is the regular program counter. Register SB contain the address of M's data segment, containing M's static, global variables. All these registers change their values, whenever an external procedure is called. To speed up this process, the processor offers the CXP (call external proceure) in addition to the regular BSR (branch subroutine). Of course, also a paor of corresponding returns is available: RXP, RTS.

Assume now that a procedure P in a module M is to be activated. The CXP instruction's parameter d specifies the entry in the current link table. From this is obtained the address of M's descriptor, and also the offset of P within M's code segment. from the descriptor, then, the address of M's data segment is obtained and loaded into SB. All this with a single, short instruction! However, what had been gained in linking simplicity and in code density, must be paid somewhere, namely by an increased number of indirect references, and, incidentally, by additional hardware, the MOD and SB registers.

A second, similar example was an instruction to check array bounds. It compared an array index against the array's lower and upper bounds, and it caused a trap, if the index did not lie withing the bounds, thereby combining two comparsion and two branch instructions in one.

Several years after our Oberon compiler had been built and released, new, faster versions of the processor appeared. They went with the trend to implement frequent, simple instructions directly by hardware, and to let the complex ones be interpreted by an internal microcode. As a result, those language-oriented instructions became rather slow compared to the simple operations. So I decided to program a new version of the compiler which refrained from using the sophisticated instructions. The result was astonishing! The new code was considerably faster than the old one. It seems that the computer archictect and we as compiler designers had "optimized" in the wrong place.

Indeed, advanced microprocessors in the early 1980s started to compete with old main frames, featuring very complex and irregular instruction sets. In fact, instruction sets had become so complex that most programmers could use only a small fraction of them. Also compilers selected only from a subset of the available instructions, a clear sign that hardware architects had gone overboard. The reaction became manifest in the form of the reduced instruction set computers (RISC), notably the Arm, Mips and Sparc architectures around 1990. They featured a small set of simple instructions, all executing in a single clock cycle, a single addressing mode, and a fairly large, single bank of registers, in short: a highly regular structure. They debunked the CISCs as a bad idea!

### 3.9. Register windows

When a procedure is called, a block of storage for its local variables must be allocated. When the procedure is terminated, this storage is to be released. A most convenient scheme for this purpose is the stack, because release is for free and involves merely the resetting of an address (stack pointer) in a register.

An often used technique for code optimization is to allocate the most frequently accessed variables in registers. Ideally, all variables would reside in registers. However, if only a few in each procedure block would remain in registers, a substantial gain in speed could already be achieved. This led to the idea of *register windows*: Upon call of a procedure, the current set of registers is pushed down, and a new set becomes available for the new block. As it was foreseeable that memory would become faster, larger, and cheaper in the future, more and more of these registers would be placed in real fast storage without the programmer having to adapt his programs. This was the idea of *scalability*. It led to the *Sparc Processor* with the built-in mechanism of register windows: Of the entire stack of registers, only those on top – in the window – are accessible. Every procedure call opens a new window, every termination pops it off the stack.

This seemed like a rather clever idea. But it turned out to be a questionable one, although the Sparc architecture survived as one of the successful examples of RISCs, and it was later copied by Intel's Itanium processor. The reason why it must be considered as questionable, is that at any moment only the top registers, those in the most recently opened window, are under direct use and are therefore frequently accessed. The others

are quasi dormant, yet consume a scare, expensive resource. Thus, register windows imply a poor utilization of the most expensive resource.

## 4. Programming Language Features

A fertile ground for controvercial ideas is the subject of programming languages. Here, some of the ideas were not only questionable, but known to be bad from the outset. We try to avoid discussing the merits and atrocities of individual languages. We will rather concentrate our attention to individual concepts and constructs that have possibly appeared in several languages. We shall mostly base our discussion on features proposed by Algol in 1960 and by some of its successors [1].

Before starting to list individual features, it seems necessary to explain on which basis to assess them. Most people consider a programming language merely as a code with the sole purpose of constructing software to be "run" by computers. We consider a language as a model of computation and programs as formal texts amenable to mathematical reasoning. The model must be defined in such a way that its semantics are defined without reference to an underlying mechanism, be it physical or abstract. This evidently lets a complex set of features and facilities explained in large volumes of manuals appear as a patently bad idea. Actually, a language is not so much characterized by what it allows to program, but more so by what it prevents from being expressed. Quoting the late E. W. Dijkstra, the programmer's most difficult, daily task is to not mess things up. It seems to me that the first and noble duty of a language is to help in this eternal struggle.

### 4.1. Notation and Syntax

It has become fashionable to regard notation as a secondary issue depending purely on personal taste. This may partly be true; yet the choice of notation should not be considered an arbitrary matter. It has consequences, and it reveals the character of a language.

A notorious example for a bad idea was the choice of the equal sign to denote assignment. It goes back to Fortran in 1957 and has blindly been copied by armies of language designers. Why is it a bad idea? Because it overthrows a century old tradition to let "=" denote a comparison for equality, a predicate which is either true or false. But Fortran made it to mean assignment, the *enforcing* of equality. In this case, the operands are on unequal footing: The left operand (a variable) is to be made equal to the right operand (an expression). x = y does not mean the same thing as y = x. Algol corrected this mistake by the simple solution: Let assignment be denoted by ":=".

Perhaps this may appear as nitpicking to programmers who got used to the equal sign meaning assignment. But mixing up assignment and comparison is a truly bad idea, because it requires that another symbol be used for what traditionally was expressed by the equal sign. Comparison for equality became denoted by the two characters "==" (first in C). This is a consequence of the ugly kind, and it gave rise to similar bad ideas using "++", "--", "&&" etc.

Some of these operators exert side-effects (in C, C++, Java, and C#), a notorious source of programming mistakes. It might be acceptable to let, for example, ++ denote incrementation by 1, if it would not also denote the incremented value (or the value to be

incremented?), thereby allowing expressions with side-effects. The heart of the trouble lies in the elimination of the fundamental distinction between *statement* and *expression*. The former is an instruction, and it is *executed*; the latter stands for a value to be computed, and it is *evaluated*.

The ugliness of a construct usually appears in combination with other language features. In C, we may write, for example, x+++++y, a riddle rather than an expression, and a challenge for a sophisticated parser! Guess what? Is its value is equal to ++x+++y+1? Or is the following correct?

x+++++y+1==++x+++y  x+++y++==x+++++y+1

One is tempted to postulate a new algebra! It is indeed absolutely surprising with which eqanimity this notational monster was accepted by the world-wide programmer's community.

A similar break with established convention was the postulation of operators being right-associative in the language APL in 1962. *x+y+z* now suddenly stood for *x+(y+z),* and *x-y-z* for *x-y+z*. What a treacherous pitfall!

A case of unfortunate syntax rather than merely poor choice of a symbol was Algol's conditional statement. It was offered in two forms, S0, S1 being statements:

**if** b **then** S0
**if** b **then** S0 **else** S1

This definition has given rise to an inherent ambiguity and became known as the *dangling else* problem.. For example, the statement

**if** b0 **then if** b1 **then** S0 **else** S1

can be interpreted in two ways, namely

**if** b0 **then** (**if** b1 **then** S0 **else** S1)
**if** b0 **then** (**if** b1 **then** S0) **else** S1

possibly leading to quite different results. The next example appears even graver:

**if** b0 **then for** i := 1 **step** 1 **until** 100 **do if** b1 **then** S0 **else** S1

because it can be parsed in two ways, yielding quite different computations:

**if** b0 **then** [**for** i := 1 **step** 1 **until** 100 **do if** b1 **then** S0 **else** S1]
**if** b0 **then** [**for** i := 1 **step** 1 **until** 100 **do if** b1 **then** S0] **else** S1

The remedy, however, is quite simple: Use an explicit **end** symbol in every construct that is recursive and begins with an explicit start symbol, like **if, while, for, case**:

**if** b **then** S0 **end**
**if** b **then** S0 **else** S1 **end**

### 4.2. The GO TO statement

Which other feature could serve better as a starting point for a list of bad ideas? The **go to** statement has been the villain of many critics. It is the direct counterpart in languages to the jump in instruction sets. It can be used to construct conditional as well as repeated

statements. But it also allows to construct any maze or mess of program flow. It defies any regular structure, and makes structured reasoning about such programs difficult if not impossible.

Let us explain why the **go to** statement became the prototype of a bad idea in programming languages. The primary tools in our struggle to comprehend and control complex objects are structure and abstraction. An overly complex object is broken up into parts. The specification of each part abstracts from aspects that are irrelevant for the whole, whose relevance is local to the object itself. Hence, the design of the whole can proceed with knowledge limited to the object's specifications, to its interface.

As a corollary, a language must allow, encourage, or even enforce formulation of programs as properly nested structures, in which properties of the whole can be derived from properties of the parts. Consider, for example, the specification of a repetition R of a statement S. It follows that S appears as a part of R. We show two possible forms:

R0: **while** b **do** S **end**
R1: **repeat** S **until** b

The key behind proper nesting is that known properties of S can be used to derive properties of R. For example, given that a condition (assertion) P is left valid (invariant) under execution of S, we conclude that P is also left invariant when execution of S is repeated. This is formally expressed by Hoare's rules

{P & b} **S** {P} implies {P} **R0** {P & ¬b}
{P} **S** {P} implies {P} **R1** {P & b}

If, however, S contains a **go to** statement, no such assertion is possible about S, and therefore neither any deduction about the effect of R. This is a great loss. Practice has indeed shown that large programs without **go to** are much easier to understand, and that it is very much easier to give any guarantees about their properties.

Enough has been said and written about this non-feature to convince almost everyone that it is a primary example of a bad idea. The designer of Pascal retained the **goto** statement (as well as the if statement without closing end symbol). Apparently he lacked the courage to break with convention and made wrong concessions to traditionalists. But that was in 1968. By now, almost everybody has understood the problem, but apparently not the designers of the latest commercial programming languages, such as C#.

### 4.3. Switches

If a feature is a bad idea, then features built on top of it are even worse ideas. This rule can well be demonstrated by the switch concept. A switch is essentially an array of labels. Assuming, for example, labels L1, … L5, a switch declaration in Algol may look as follows:

**switch** S := L1, L2, **if** x < 5 **then** L3 **else** L4, L5

Now the apparently simple statement ***goto*** *S[i]* is equivalent to

**if** i = 1 **then goto** L1 **else**
**if** i = 2 **then goto** L2 **else**
**if** i = 3 **then**

```
    if x < 5 then goto L3 else goto L4 else
if i = 4 then goto L5
```

If the **goto** is suitable for programming a mess, the switch makes it impossible to avoid it.

A most suitable replacement of the switch was proposed by C.A.R.Hoare in 1965: The case statement. This construct displays a proper structure with component statements to be selected according to the value i:

```
case i of
      1: S1 | 2: S2 | …….. | n: Sn
end
```

However, modern programming language designers chose to ignore this elegant solution in favor of a formulation that is a bastard between the Algol switch and a structured case statement:

```
switch (i) {
    case 1: S1; break;
    case 2: S2; break;
    … ;
    case n: Sn; break; }
```

Either the break symbol denotes a separation between consecutive statements *Si*, or it acts as a **goto** to the end of the switch construct. In the first case, it is superfluous, in the second a **goto** in disguise. A bad concept in a bad notation! The example stems from C.

### 4.4. Algol's complicated for statement

Algol's designers recognized that certain frequent cases of repetition would better be expressed by a conciser form than in combination with **goto** statements. They introduced the **for** statement, which is particularly convenient in use with arrays, as for example in

```
for i := 1 step 1 until n do a[i] := 0
```

If we forgive the rather unfortunate choice of the words *step* and *until*, this seems a wonderful idea. Unfortunately, the good idea was infected with a bad idea, the idea of imaginative generality. The sequence of values to be assumed by the control variable i can be specified as a list:

```
for i := 2, 3, 5, 7, 11 do a[i] := 0
```

Furthermore, these elements could be general expressions:

```
for i := x, x+1, y-5, x*(y+z) do a[i] := 0
```

Not enough, also different forms of list elements were to be allowed:

```
for i := x-3, x step 1 until y, y+7, z while z < 20 do a[i] := 0
```

Naturally, clever minds would quickly concoct pathological cases, demonstrating the absurdity of the concept:

```
for i := 1 step 1 until i do a[i] := 0
for i := 1 step i until i do i := - i
```

The generality of Algol's **for** statement should have been a warning signal to all future designers to always keep the primary purpose of a construct in mind, and to be weary of exaggerated generality and complexity, which may easily become counter-productive.

### 4.5. Own variables

Algol had introduced the concept of *locality*. Every procedure spans its own scope, implying that identifiers declared in the procedure would be local and invisible outside the procedure. This was probably the most significant innovation introduced by Algol. Yet, it turned out to be not quite right in certain situations. Specifically, when entering a procedure, all local variables are fresh variables. This implies that new storage must be allocated upon entry and released upon exit of the procedure. So far, so good. However, in some cases it might be desirable to have the value of a variable upon exit remembered when the procedure is reentered the next time. Evidently, its storage could then not be released. Algol provided for this option by simply letting the variable be declared as **own**. A popular example for this case is a procedure for generating pseudo-random numbers:

**real procedure** random; **own real** x; **begin** x := (x*a + b) **mod** c; random := x **end**

Already this simple example exhibits the crux buried in this concept: How is the initial seed assigned to the variable *x* owned by *random*? Any solution turns out to be rather cumbersome. A further unresolved question is the meaning of **own** in the case of recursion. Evidently, the problem lay deeper. The simple introduction of the symbol **own** had evidently created more problems than it had solved.

The problem lay in Algol's clever notion of tying together the concepts of scopes of identifiers with that of lifetime of objects. An identifier became visible when the identified object came into existence (at block entry), and became invisible, when it ceased to exist (at block exit). The close connection between visibility and existence had to be broken up. This was done later (in Mesa, Modula, Ada) by introducing the concept of *module* (or package). A module's variables are global, but accessible only from procedures declared within the module. Hence, when a procedure (local to and exported from the module) is terminated, the module's variables continue to exist, and when the procedure is called again, the old values reappear. The module hides the variables (information hiding), which are initialized either when the module is loaded or by an explicit call of an initialization procedure.

This example displays the problems that appear when different concepts – here information hiding and liveness – are represented by a single language construct. The solution is to use separate constructs to express the separate concepts, the module for information hiding, the procedure for liveness of data. A similar case of an unfortunate marriage is given by some object-oriented languages, which tie together (1) objects with pointers, and (2) modules with types, called classes.

### 4.6. Algol's name parameter

Algol introduced procedures and parameters in a much greater generality than was known in older languages (Fortran). In particular, parameters were seen as in traditional mathematics of functions, where the actual parameter *textually* replaces the formal parameter [3]. For example, given the declaration

**real procedure** square(x); **real** x; square := x * x

the call *square(a)* is to be literally interpreted as *a*a*, and *square(sin(a)*cos(b))* as *sin(a)*cos(b) * sin(a)*cos(b)*. This requires the evaluation of sine and cosine twice, which in all likelihood was not the intention of the programmer. In order to prevent this frequent, misleading case, a second kind of parameter was postulated in addition to the above *name parameter*: the *value parameter*. It meant that a local variable (x') is to be allocated that is initialized with the value of the actual parameter (x). With

**real procedure** square(x); **value** x; **real** x; square := x * x

the above call was to be interpreted as

x' := sin(a) * cos(b); square := x' * x'

avoiding the wasteful double evaluation of the actual parameter. The name parameter is indeed a most flexible device, as is demonstrated by the following examples.

Given the declaration

```
real procedure sum(k, x, n); integer k, n; real x;
begin real s; s := 0;
    for k := 1 step 1 until n do s := x + s;
    sum := s
end
```

Now the sum $a_1 + a_2 + \ldots + a_{100}$ is written simply as *sum(i, a[i], 100)*, the inner product of vectors *a* and *b* as *sum(i, a[i]*b[i], 100)* and the harmonic function as *sum(i, 1/i, n)*. But generality, as elegant and sophisticated as it may appear, has its price. A little reflection reveals that every name parameter must be implemented as an anonymous, parameterless procedure. Will the unsuspecting programmer gladly pay for the hidden overhead? Of course, a cleverly designed compiler might "optimize" certain cases. The smart designer will rejoyce about the challenge presented by such a wonderful feature!

The reader will perhaps ask at this point: "Why all this fuss?" Perhaps he will come forward with the suggestion to simply drop the name parameter from the language. However, this measure would be too drastic and therefore unacceptable. It would, for example, preclude assignments to parameters in order to pass results back to the caller. The suggestion, however, led to the replacement of the name parameter by the reference parameter in later languages such as Algol W, Pascal, Ada, etc. For today, the message is this: Be skeptical towards overly sophisticated features and facilities. At the very least, their cost to the user must be known before a language is released, published, and propagated. This cost must be commensurate with the advantages gained by the feature.

### 4.7. Incomplete parameter specifications

This issue is almost a continuation of the previous one, as it concerns parameters. It so happened that the language failed to require complete type specifications for parameters. It now almost appears as an oversight, but one with grave consequences. Algol permitted, for example, the following declaration:

**real procedure** f(x, y); f := (x-y)/(x+y)

Given variables

**integer** k; **real** u, v, w

the following calls are possible:

u := f(2.7, 3.14); v := f(f(u+v), 10); w := f(k, 2)

So far, so (more or less) good. More problematic is the following:

**real procedure** g; g := u*v + w;
u := f(g, g)

Apparently, one cannot deduce from the declaration of *f*, whether a variable, an expression, or a function will be substituted for a given formal parameter, nor whether the function's parameters are of the correct number and types. When such a formal parameter is accessed, a test at run-time must first determine, whether the corresponding actual parameter is a variable, an expression, or a function. This is an overhead unacceptable in most practical applications.

The story might end here, being of academic interest only, that is, without consequences. But it does not. After all, the challenge was issued to fulfill the postulated language rules, and quite obviously the question arose, whether hardware support might provide the remedy. The Burroughs Corporation went farthest along this path with its B5000 computer. We have already mentioned the introduction of data descriptors to access arrays, and now we follow with the program descriptor for procedures. Data and program descriptors differed from data words by their tag field values. If a program descriptor was accessed by a regular load instruction, not only a simple memory access was performed, but a procedure call. In short, what cannot be decided by inspection of the program, i.e. at compile-time, has to be done at run-time. Proper hardware support was the last resort. It was found in an instruction that can either be a simple data fetch or a complex procedure call, depending on a tag in the accessed word.

But was this a good or a bad idea? Rather the latter, not only because the mentioned computer was complicated, expensive, and therefore less than successful in the market, but because it is not wise to include very complex instructions, whose benefit is marginal. After all, the language feature that led to these complications must be considered a mistake, a design flaw, and competent programmers would provide complete parameter specifications anyway in the interest of program clarity. The urge to obey the Algol specification to its last letter was an idea of questionable wisdom.

In closing this topic, we present a short procedure declaration, simply to show how a combination of features, apparently innocent in isolation, can turn into a puzzle.

```
procedure P(Boolean b; procedure q);
begin integer x;
    procedure Q; x := x+1;
    x := 0;
    if b then P(¬b, Q) else q;
    write(x)
end
```

Which values will be written by calling P(**true**, P)? Is it 0, 1, or 1, 0?

### 4.8. Loopholes

One of the worst features ever is the *loophole*. This author makes this statement with a certain amount of tongue in cheek and uneasiness, because he infected his languages Pascal, Modula, and even Oberon with this deadly virus.

The loophole lets the programmer breach the type checking by the compiler. It is a way to say: "Don't interfere, as I am smarter than the rules". Loopholes take many forms. The most common are explicit type transfer function, such as

```
x := LOOPHOLE[i, REAL]        Mesa
x := REAL(i)                  Modula
x := SYSTEM.VAL(REAL, i)      Oberon
```

But they can also be disguised as absolute address specifications, or by variant records as in Pascal. In the examples above, the internal representation of integer *i* is to be interpreted as a floating-point (real) number. This can only be done with knowledge about number representation, which should not be necessary when dealing with the abstraction level provided by the language. In Pascal and Modula [4], loopholes were at least honestly displayed, and in Oberon they are present only through a small number of functions encapsulated in a pseudo-module *System*, which must be imported and is thus explicitly visible in the heading of any module in which such low-level facilities are used. This may sound like an excuse, but the loophole is nevertheless a bad idea.

Why, then, was it introduced? The reason was the desire to implement total systems in one and the same (system programming) language. For example, a storage manager must be able to look at storage as a flat array of locations without data types. It must be able to allocate blocks and to recycle them, independent of any type constraints. Another example where a loophole is required is the device driver. In earlier computers, devices were accessed by special instructions. Later, devices were instead assigned specific memory addresses. They were "memory-mapped". Thus arose the idea to let absolute addresses be specified for certain variables (Modula). But this is a facility that can be abused in many clandestine and tricky ways.

Evidently, the "normal" user will never need to program a storage manager nor a device driver, and hence has no need for those loopholes. However – and this is what really makes the loophole a bad idea – also the "normal" programmer has the loophole at his disposal. Experience showed that he will not shy away from using it, but rather enthusiastically grab the loophole as a wonderful feature and use it wherever possible. This is particularly so, if manuals caution against its use!

It remains to say, that the presence of a loophole facility usually points to a deficiency in the language proper, it reveals that certain things could not be expressed that might be important. An example of this kind is the type *address* in Modula, which had to be used to program data structures with different types of elements. This was made impossible by the strict, static typing strategy, which demanded that every pointer was statically associated with a fixed type, and could only reference such objects. Knowing that pointers were addresses, the loophole in the form of an innocent looking type transfer function would make it possible to let pointer variables point to objects of any type. Of course, the drawback was that no compiler could check the correctness of such

assignments. The type checking system was overruled, and might as well not have existed. Remember: A chain is only as strong as its weakest member.

The clean solution given in Oberon [6] is the concept of *type extension*, in object-oriented languages called *inheritance*. Now it became possible to declare a pointer as referencing a given type, and it would be able to point to any type which was an extension of the given type. This made it possible to construct inhomogeneous data structures, and to use them with the security of a reliable type checking system. An implementation must check at run-time, if and only if it is not possible to check at compile-time,.

Programs expressed in languages of the 1960s were full of loopholes. They made these programs utterly error-prone. But there was no alternative. The fact that a language like Oberon lets you program entire systems from scratch without use of loopholes (except in the storage manager and device drivers) marks the most significant progress in language design over 40 years.

## 5. Miscellaneus techniques

The last section of bad ideas stems from the wide area of software practice, or rather from the narrower area of the author's experiences. Some of the latter were made 40 years ago, but what can be learnt from them is still as valid today as it was then. Some reflect more recent practices and trends, mostly supported by the abundant availability of hardware power.

### 5.1. Syntax analysis

The 1960s were the decade of syntax analysis. The definition of Algol by a formal syntax provided the necessary underpinnings to turn language definition and compiler construction into a field of scientific merit. It established the concept of the syntax-driven compiler, and it gave rise to many activities for automatic syntax analysis on a mathematically rigorous basis. The notions of top-down and bottom-up principles, of the recursive descent technique, of measures for symbol look-ahead and backtracking were created. This was also accompanied with efforts to define language semantics more rigorously by piggybacking semantic rules onto corresponding syntax rules.

As it happens with new fields of endeavor, research went rather beyond the needs of the first hour. More and more powerful parser generators were developed, which managed to handle ever and ever more general and complex grammars. Albeit this was an intellectual achievement, the consequence was less positive. It led language designers to believe that no matter what syntactic construct they postulated, automatic tools could surely detect ambiguities, and some powerful parser would certainly cope with it. A misled idea! No such tool would give any indication how that syntax could be improved. Not only had designers ignored the issue of efficiency, but also the fact that a language serves the human reader, and not only the automatic parser. If a language poses difficulties to parsers, it surely does so for the human reader too. Many languages would be clearer and cleaner, had their designers been forced to use a simple parsing method.

I can strongly support this statement by my own experiences. After having contributed in the 1960s to the development of parsers for precedence grammars, and having used them for the implementation of the languages Euler and Algol W, I decided to switch to the

simplest parsing method for Pascal, the method of top-down, recursive descent. The experience was most encouraging, and I stuck to it up to this day with great satisfaction.

The drawback, if one wants to call so this advantage, is that considerably more careful thought has to go into the design of the syntax prior to publication and any effort of implementation. This additional effort will be more than compensated in the later use of both language and compiler.

### 5.2. Extensible languages

The phantasies of computer scientists in the 1960s knew no bounds. Spurned by the success of automatic syntax analysis and parser generation, some proposed the idea of the flexible, or at least extensible language. The notion was that a program would be preceded by syntactic rules which would then guide the general parser while parsing the subsequent program. A step further: The syntax rules would not only precede the program, but they could be interspersed anywhere throughout the text. For example, if someone wished to use a particularly fancy private form of **for** statement, he could do so elegantly, even specifying different variants for the same concept in different sections of the same program. The concept that languages serve to communicate between humans had been completely blended out, as apparently everyone could now define his own language on the fly. The high hopes, however, were soon damped by the difficulties encountered when trying to specify, what these private constructions should mean. As a consequence, the intreaguing idea of extensible languages faded away rather quickly.

### 5.3. Nested procedures and Dijkstra's display

Local variables for every procedure! That was one of the greatest inventions of Algol. The idea to declare other procedures along with variables local to procedures was only natural and straight-forward. It was a natural consequence for any mathematically trained mind. But sometimes even straight-forward concepts cause substantial complications for their implementation. The nesting of procedures is such an example. The problem lies in the addressing of variables (or objects more generally). To understand it, we need some preliminaries.

Let us assume the following constellation of three nested procedures P, Q, and R:

```
procedure P;
begin integer i;
    procedure Q;
    begin integer j;
        procedure R;
        begin integer k; (*P, Q, R, i, j, k visible here*)
        end of R;
        (*P, Q, R, i, j visible here*)
    end of Q;
    (*P, Q, i visible here*)
end of P
```

How are i, j, and k addressed in the body of R? The possibility of recursion prevents static addressing. Every procedure call establishes a frame for its own local variables.

They are addressed relative to the frame's base address, preferably located in a register. Typically, these frames are placed in a stack (the procedure stack in contrast to the expression stack), and the frames are linked. Therefore, every access is preceded by descending along this link chain by a number of steps which is given by the difference in nesting levels between the accessed variable and the accessing procedure. In the example above, in the body of R the base of k is found by descending 0 steps, the base of j by descending 1 step, and the base of i by descending 2 steps in the link.

This simple scheme is, unfortunately, wrong. In addition to the mentioned *dynamic link* also a *static link* must be maintained. The static link always points to the block of the static environment of a procedure. Even if this chain is usually quite short, variable access is slowed down, if previously a linked chain has to be traversed. E. W. Dijkstra therefore proposed to circumvent the traversal of a list by mapping the links into an array of contiguous memory cells (or registers) called *the display*, and to use the block level as index.

The crux of the scheme lies in the fact that the display has to be updated every time a procedure is called and, unfortunately, sometimes also when it is terminated. Such updates may well involve several registers, as the following pathological example shows:

```
procedure A(procedure X);
begin integer x;
    procedure B;
    begin integer y;
        procedure C;
        begin integer z; X
        end of C;
        C
    end of B;
    B
end of A;

procedure P;
begin integer i;
    procedure Q;
    begin integer j;
        procedure R;
        begin integer k;
        end of R;
        A(R)
    end of Q;
    Q
end of P
```

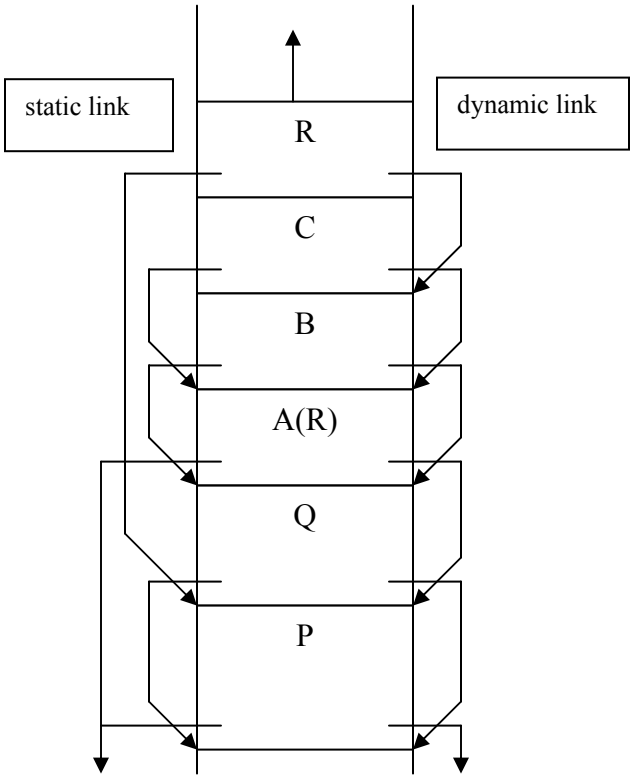


Fig. 2. Procedure frame stack with links

A call of P will evoke the sequence of subsequent activations of P, Q, A, B, C, R. Within R only the blocks of R, Q and P are accessible with respective variables k, j, i. Upon exit from R visibility switches back to blocks C, B, A with respective variables z, y, x. It turned out that the good idea had aggravated rather than solved the problem.

It is indeed difficult to estimate in such complex situations, whether the introduction of a display makes programs more efficient due to faster memory access (no chain traversals), or whether it slows them down because of the overhead of display updates. This strongly depends on the frequency of variable access versus procedure call. In 1970 we had implemented Pascal for the CDC 6000 computer, and it occurred to the author that indeed procedure calls became slower and code longer due to the presence of a display. Therefore, we tried to do without display and found that the compiler itself had become faster and shorter. The display was a disputable idea.

However, as so often with "optimizations", the benefit varies among different programs. Obviously, the display does *not* affect at all programs without nested procedures. Indeed, we have here an optimization that seldom can be applied, as intermediate level variables are relatively rare. The Burroughs Algol compiler did not cater for such variables. All accesses had to be either global or strictly local. This was probably a good idea, also with respect to programming clarity.

The main lesson here is that when implementing an optimization facility, one must first find out whether it is worth while. Usually, it is worth while only for frequently used constructs.

### 5.4. Tree-structured symbol tables

Compilers construct symbol tables. They are built up while processing declarations, and they are searched during the processing of statements. In languages that allow nested scopes, every scope is represented by its own table.

Traditionally these tables are binary trees in order to allow fast searching. Having also quietly followed this long-standing tradition, the author dared to doubt the benefit of trees when implementing the Oberon compiler. As soon as doubts occur, one is quickly convinced that tree structures are not worth while for local scopes. In the majority of cases, procedures contain a dozen or even fewer local variables. The use of a linked linear list is then both simpler and more effective.

In programs 30 and 40 years ago, most variables were declared globally. So perhaps a tree structure was justified for the global scope. In the meantime, however, skeptizism against the value of global variables had been on the rise. Modern programs do not feature many globals, and hence also in this place a tree structured table is hardly recommendable.

Modern programming systems are compositions of many modules, each of which probably contains some globals (mostly procedures), but not hundreds. The many globals of early programs have become distributed over many modules and are referenced not by a single identifier, but by a name combination M.x defining the initial search path.

The use of sophisticated data structures for symbol tables had evidently been a poor idea. Once we had even considered balanced trees!

### 5.5. Using wrong tools

Using the wrong tools is obviously an intrinsically bad idea. The trouble is that often one discovers a tool's inadequacy only after having invested a substantial amount of effort to

build and understand it, and the tool thus having become “valuable”. This happened to the author and his team when implementing the first Pascal compiler in 1969.

The tools available for writing programs were an assembler, a Fortran and an Algol compiler. The latter was so poorly implemented that we did not dare rely on it, and work with assembler code was considered dishonorable. There remained only Fortran.

Hence our naïve plans were to construct a compiler for a substantial subset of Pascal using Fortran, and when completed, to translate it into Pascal. Afterwards, the classical bootstrapping technique would be employed to complete, refine, and improve the compiler.

This plan, however, crashed in the face of reality. When step one was completed after about one man-year’s labor, it turned out that translation of the Fortran code into Pascal was quite impossible. That program was so much determined by Fortran’s features, or rather its lack of any, that there was no other option than to write the compiler afresh. Fortran did not feature pointers and records, and therefore symbol tables had to be squeezed into the unnatural form of arrays. Fortran did not feature recursive subroutines. Hence the complicated table-driven bottom-up parsing technique had to be used with syntax represented by arrays and matrices. In short, the advantages of Pascal could only be employed by a completely rewritten and restructured, new compiler.

This “incident” revealed, that the apparently easiest way is not always the right way, but that difficulties also have their benefits: This new compiler, written in Pascal, could not be tested during development, because no Pascal compiler was available yet. The whole program for compiling at least a very substantial subset of Pascal, had to be written without feedback from testing. This was an exercise that was extremely healthy, and it would be even more so today, in the era of quick trial and interactive error correction. After we believed that the compiler was “complete”, one member of the team was banned to his home to translate the program into a syntax-sugared, low-level language, for which a compiler was available. He returned after two weeks of intensive labor, and a few days later the first test programs were compiled correctly by the compiler written in Pascal. The exercise of conscientious programming proved to have been extremely valuable. Never contain programs so few bugs, as when no debugging tools are available!

Thereafter, new versions, handling more and more of Pascal’s constructs, and producing more and more refined code, could be obtained by bootstrapping. Immediately after the first bootstrap, we discarded the translation written in the auxiliary language. Its character had been much alike that of the ominous low-level language C, published a year later. After this experience, it was hard to understand that the software engineering community did not recognize the benefits of adopting a high-level, type-safe language instead of C.

### 5.6. Wizards

We have discussed the great leaps forward in parsing technology originating in the 1960s. The results are ever-present since those years. Hardly anybody now constructs a parser by hand. Instead, one buys a parser generator and feeds it the desired syntax.

This brings us to the topic of automatic tools, now being called *wizards*. The idea is that they are to be considered as black boxes, and that the user would not have to understand their innards, as they were optimally designed and laid out by experts. The idea is that

they automate simple routine tasks, relieving computer users from bothering about them. Wizards supposedly help you – and this is the key – without your asking, as a faithful, devoted servant.

Although it would be unwise to launch a crusade against wonderful wizards, this author's experiences with wizards were largely unfortunate. He found it impossible to avoid confronting them in text editors. Worst are those wizards that constantly interfere with one's writing, automatically indenting and numbering lines when not desired, capitalizing certain letters and words at specific places, combining sequences of characters into some special symbol, ☺, automatically converting a sequence of minus signs into a solid line, etc. If at least they could easily be deactivated, but typically they are obstinate and immortal like devils. So much for clever *software for dummies*: a bad idea!

# 6. Programming paradigms

## 6.1. Functional programming

Functional languages had their origin in Lisp [2]. They have undergone a significant amount of development and change, and they have been used to implement small and large software systems. This author has always maintained a critical attitude towards such efforts. Why?

What is, or what characterizes a functional language? It has always appeared that it was their form, the fact that the entire program consists of function evaluations, nested, recursive, parametric, etc. Hence the term *functional*. However, the core of the idea is that functions inherently have *no state*. This implies that there are *no variables* and no assignments. The place of variables is taken by immutable function parameters, variables in the sense of mathematics. As a consequence, freshly computed values cannot be reassigned to the same variable, overwriting its old value. This is why repetion must be expressed with recursion. A data structure can at best be extended, but no change is possible in its old part. This yields an extremely high degree of storage recycling; a garbage collector is the necessary ingredient. An implementation without automatic garbage collection is unthinkable.

To postulate a state-less model of computation on top of a machinery whose most eminent characteristic is state, seems to be an odd idea, to say the least. The gap between model and machinery is wide, and therefore costly to bridge. No hardware support feature can wash this fact aside: It remains a bad idea for practice. This has in due time also been recognized by the protagonists of functional languages. They have introduced state (and variables) in various tricky ways. The purely functional character has thereby been compromised and sacrificed. The old terminology has become deceiving.

Looking back at the subject of functional programming, it appears that its truly relevant contribution was certainly not its lack of state, but rather its enforcement of clearly nested structures, and of the use of strictly local objects. This discipline can, of course, also be practiced using conventional, imperative languages, which have subscribed to the notions of nested structures, functions and recursion long ago. Of course, functional programming implies much more than avoiding **goto** statements. It also implies restriction to local variables, perhaps with the exception of very few global state variables. It probably also considers the nesting of procedures as undesirable. The B5000

computer apparently has been right, after all, in restricting access to strictly local and strictly global variables.

Are functional languages thus a category of their own merely due to terminology? Are their functions functions by form only, but not by substance? Or is the substance of the functional paradigm expressed by simply saying: “No side-effects”?

Many years ago, and with increasing frequency, it is claimed that functional languages are the best vehicle to introduce parallelism. It would be more to the point to say: To facilitate compilers to detect opportunities for parallelizing a program. After all, it is relatively easy to determine which parts of an expression may be evaluated concurrently. More important is that parameters of a called function may be evaluated concurrently, provided, of course, that side-effects are banned (which cannot occur in a truly functional language). As this may be true and perhaps of marginal benefit, this writer believes that a more effective way to let a system make good use of parallelism is provided by object-orientation, each object representing its own behaviour in the form of a “private” process.

### 6.2. Logic programming

Another instance of programming paradigm that has received wide attention is that of logic programming. Actually, there is only a single well-known language representing this paradigm: Prolog. Its principal idea is that the specification of actions, such as assignment to variables, is replaced by the specification of predicates on states. If one or several of a predicate’s parameters are left unspecified, the system searches for all possible argument values satisfying the predicate. This implies the existence of a search engine looking for solutions of logic statements. This mechanism is complicated, often time-consuming, and sometimes inherently unable to proceed without intervention. This, however, requires that the user must support the system by providing hints (cuts), and therefore must understand what is going on, must understand the process of logic inference, the very thing that he had been promised to be able to ignore.

One must suspect that an interesting intellectual exercise was sold to the public by raising great expectations. The community was in desperate need for ways to produce better, more reliable software, and was glad to hear of a possible panacea. But the promises never materialized. We sadly recall the exaggerated hopes that fueled the project of the Japanese Fifth Generation Computer, Prolog’s inference machines. Large amounts of resources were sunk into it. That was an unwise and now forgotten idea.

### 6.3. Object-oriented programming

In contrast to functional and logic programming, object-oriented programming (OOP) rests on the same priciples as conventional, procedural programming. Its character is imperative. A process is described as a sequence of transformations of a state. The novelty is the partitioning of a global state into individual *objects*, and the association of the state transformers (called *methods*) with the object itself. The objects are seen as the actors, causing other objects to alter their state by sending *messages* to them. The description of an object template is called a *class* definition.

This paradigm closely reflects the structure of systems “in the real world”, and it is therefore well suited to model complex systems with complex behaviour. Not

surprisingly, oop has its origins in the field of system simulation (Simula, Dahl and Nygaard, 1966). Its success in the field of software system design speaks for itself. Its career started with the language Smalltalk [5] and continued with Object-Pascal, C++, Eiffel, Oberon, Java, C#. The original Smalltalk implementation provided a convincing example of its suitability. It was the first to feature windows, menues, buttons and icons, perfect examples of (visible) objects in the sense outlined above. These examples were the carriers to success and wide acceptance. The direct modelling of actors diminished the importance of proving program correctness analytically, because the original specification is one of behaviour, rather than a static input-output relationship.

Nevertheless, the careful observer may wonder, where the core of the new paradigm would hide, what was the essential difference to the traditional view of programming. After all, the old cornerstones of procedural programming reappear, albeit embedded in a new terminology: Objects are records, classes are types, methods are procedures, and sending a method is equivalent to calling a procedure. True, records now consist of data fields and, in addition, methods; and true, the feature called *inheritance* allows the construction of heterogeneous data structures, useful also without object-orientation. Was this change of terminology expressing an essential paradigm shift, or was it a vehicle for gaining attention, a "sales trick"?

## 7. Concluding remarks

A collection of ideas has been presented, stemming from a wide spectrum of computing science, and having been widely acclaimed at their time. For various reasons a closer inspection reveals certain weaknesses. Some of the ideas are hardly relevant now, due to changes and advances in the underlying technology, some others have been clever ideas solving a local problem no longer important, and some have received attention and success for various non-technical reasons.

We believe that we can learn not only from bad ideas and past mistakes, but even more from good ideas, analysing them from the distance of time. This collection of topics may appear as accidental, and it is certainly incomplete. Also, it is written from an individual's perspective, with the feeling that Computing Science would benefit from more frequent analysis, critique, particularly self-critique. After all, thorough self-critique is the hallmark of any subject claiming to be a science.

### References

1. P. Naur (Ed). Report on the Algorithmic Language ALGOL 60. *Comm. ACM 3* (May 1960), 299-314.
2. J. McCarthy. Recursive Functions of symbolic Expressions and their Computation by Machine. *Comm. ACM 5,* (1962)
3. D. E. Knuth. The Remaining Trouble Spots in ALGOL 60. *Comm. ACM 10* (Oct. 1967), 611-618.
4. N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.
5. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

6. N. Wirth. The Programming Language Oberon. *Software – Practice and Experience, 18,* (July 1988), 671-691.

**Author's address:**

Niklaus Wirth, Langacherstr. 4, CH-8127 **Forch** Switzerland wirth@inf.ethz.ch