Good Ideas, through the Looking Glass

Niklaus Wirth

Given that thorough self-critique is the hallmark of any subject claiming to be a science, computing science cannot help but benefit from a retrospective analysis and evaluation.

Computing's history has been driven by many good and original ideas, but a few turned out to be less brilliant than they first appeared. In many cases, changes in the technological environment reduced their importance. Often, commercial factors also influenced a good idea's importance. Some ideas simply turned out to be less effective and glorious when reviewed in retrospect or after proper analysis. Others were reincarnations of ideas invented earlier and then forgotten, perhaps because they were ahead of their time, perhaps because they had not exemplified current fashions and trends. And some ideas were reinvented, although they had already been found wanting in their first incarnation.

This led me to the idea of collecting good ideas that looked less than brilliant in retrospect. A recent talk by Charles Thacker about obsolete ideas—those that deteriorate by aging—motivated this work. I also rediscovered an article by Don Knuth titled "The Dangers of Computer Science Theory." Thacker delivered his talk in faraway China, Knuth from behind the Iron Curtain in Romania in 1970, both places safe from the damaging "Western critique." Knuth's document in particular, with its tongue-in-cheek attitude, encouraged me to write these stories.

HARDWARE TECHNOLOGY

Speed has always been computer engineers' prevalent concern. Refining existing techniques has been one alley for pursuing this goal, looking for alternative solutions the other. Although presented as most promising, the following searches ultimately proved unsuccessful.

Magnetic bubble memory

Back when magnetic core memories dominated, the idea of magnetic bubble memory appeared. As usual, its advocates attached great hopes to this concept, planning for it to replace all kinds of mechanically rotating devices, 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 rapid progress in disk technology made both the bubbles' capacity and speed inferior, so developers quietly buried the idea after a few years' research.

Cryogenics

Cryogenic devices offered a new technology that kept high hopes alive for decades, particularly in the supercomputer domain. These devices promised ultrahigh switching speeds, but the effort to operate large computing equipment at temperatures close to absolute zero proved prohibitive. The appearance of personal computers let cryogenic dreams either freeze or evaporate.

Tunnel diodes

Some developers proposed using tunnel diodes—so named because of their reliance on a quantum effect of electrons passing over an energy barrier without having the necessary energy—in place of transistors as switching and memory elements.

The tunnel diode has a peculiar characteristic with a negative segment. This lets it assume two stable states. A germanium device, the tunnel diode has no silicon-based counterpart. This made it work over only a rela-

Published by the IEEE Computer Society
0018-9162/06/$20.00 © 2006 IEEE

tively narrow temperature range. Silicon transistors became faster and cheaper at a rate that let researchers forget the tunnel diode.

COMPUTER ARCHITECTURE

This topic provides a rewarding area for finding good ideas. A fundamental issue has been representing numbers, particularly integers.

Representing numbers

Here, the key question has been the choice of the numbers' base. Virtually all early computers featured base 10—a representation by decimal digits, just as everybody learned in school.

However, a binary representation with binary digits is clearly more economical. An integer n requires $\log_{10}(n)$ decimal digits, but only $\log_2(n)$ binary digits (bits). Because a decimal digit requires four bits, decimal representation requires about 20 percent more storage than binary, which shows the binary form's clear advantage. Yet, developers retained the decimal representation for a long time, and it persists today in library module form. They did so because they continued to believe that all computations must be accurate.

However, errors occur through rounding, after division for example. The effects of rounding can differ depending on the number representation, and a binary computer can yield different results than a decimal computer. Because financial transactions—where accuracy matters most—traditionally were computed by hand with decimal arithmetic, developers felt that computers should produce the same results in all cases—and thus commit the same errors.

The binary form will generally yield more accurate results, but the decimal form remained the preferred option in financial applications, as a decimal result can easily be hand-checked if required.

Although perhaps understandable, this was clearly a conservative idea. Consider that until the advent of the IBM System/360 in 1964, which featured both binary and decimal arithmetic, manufacturers of large computers offered two lines of products: binary computers for their scientific customers and decimal computers for their commercial customers—a costly practice.

Early computers represented integers by their magnitude and a separate sign bit. In machines that relied on sequential addition, digit by digit, to be read first, the system placed the sign at the low end. When bit parallel processing became possible, the sign moved to the high end, again in analogy to the commonly used paper notation. 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 evidently provided a far superior solution, because the same circuit could now handle both addition and subtraction. Some designers chose 1's complement, where $\neg n$ was obtained from n by simply inverting all bits. Some chose 2's complement, where $\neg n$ 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.

Table 1. Using 2's complement arithmetic to avoid ambiguous results.

Decimal	2's complement	1's complement
2	010	010
1	001	001
0	000	000 or 111
–1	111	110
–2	110	101

For example, the CDC 6000 computers had an instruction that tested for zero, recognizing both forms correctly, but also an instruction that tested the sign bit only, classifying 1…1 as a negative number, making comparisons unnecessarily complicated. This case of inadequate design reveals 1's complement as a bad idea. Today, all computers use 2's complement arithmetic. The different forms are shown in Table 1.

Fixed or floating-point forms can represent numbers with fractional parts. 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, developers argued about which exponent base B they should choose. 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 ε, $(x+\varepsilon) \times (y+\varepsilon) < (x \times y)$. Multiplication had lost its monotonicity. Such a multiplication is unreliable and potentially dangerous.

Data addressing

The earliest computers' instructions consisted simply of an operation code and an absolute address or literal value as the 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 program had to modify the address of the add instruction in each step by adding 1 to it.

Although developers heralded the possibility of program modification at runtime as one of the great consequences of John von Neumann's profound idea of

storing program and data in the same memory, it turned out to enable a dangerous technique and to constitute an unlimited source of pitfalls. Program code must remain untouched to avoid having the search for errors become a nightmare. Developers soon recognized that program self-modification was a bad idea.

They avoided these pitfalls by introducing another addressing mode that treated an address as a piece of variable data rather than as part of a program instruction, which would be better left untouched. The solution involved indirect addressing and modifying only the directly addressed address, a data word.

Although this feature removed the danger of program self-modification and remained common on most computers until the mid-1970s, it should be considered a questionable idea in retrospect. After all, it required two memory accesses for each data access, which caused a computation slowdown.

The "clever" idea of multilevel indirection made this situation worse. The data accessed would indicate with a bit whether the referenced word was the desired data or another—possibly again indirect—address. Such machines could be brought to a standstill by specifying a loop of indirect addresses.

The Algol 60 language influenced the development of both programming languages and computer architecture.

The solution lay in introducing index registers. The value stored in an index register would be added to the address constant in the instruction. This required adding a few index registers and an adder to the arithmetic unit's accumulator. The IBM 360 merged them all into a single register bank, as is now customary.

The CDC 6000 computers used a peculiar arrangement: Instructions directly referred to registers only, of which there were three 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 a B-register's value. The odd thing was that references to A0-A5 implied fetching the addressed memory location into the corresponding X0-X5 register, whereas a reference to A6 or A7 implied storing X6 or X7.

Although this arrangement did not cause any great problems, we can retrospectively classify it as a mediocre idea because a register number determines the operation performed and thus the data transfer's direction. Apart from this, the CDC 6000 featured several excellent ideas, primarily its simplicity. Although Seymour Cray designed it in 1962, well before the term was coined, the CDC 6000 can truly be called the first reduced-instruction-set computer (RISC).

The Burroughs B5000 machine introduced a more sophisticated addressing scheme—its descriptor scheme, primarily used to denote arrays. A so-called *data descriptor* was essentially an indirect address, but it also contained index bounds to be checked at access time.

Although automatic index checking was an excellent and almost visionary facility, the descriptor scheme proved a questionable idea because matrices (multidimensional 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 not only slowed access due to its indirection, it also required additional rows of descriptors. Nevertheless, Java's designers adopted this idea in 1995, as did C#'s designers in 2000.

Expression stacks

The Algol 60 language had a profound influence on the development of further programming languages and, to a more limited extent, on computer architecture. This should not be surprising given that language, compiler, and computer form an inextricable complex.

The evaluation of expressions could be of arbitrary complexity in Algol, with subexpressions being parenthesized and operators having their individual binding strengths. Results of subexpressions were stored temporarily. F.L. Bauer and E.W. Dijkstra independently proposed a scheme for evaluating arbitrary expressions. They noticed that when evaluating from left to right, obeying priority rules and parentheses, the last item stored is always the first needed. It therefore could be placed in a push-down list, or stack.

Implementing this simple strategy using a register bank was straightforward, with the addition of an implicit up/down counter holding the top register's index. Such a stack reduced memory accesses and avoided explicitly identifying individual registers in the instructions. In short, stack computers seemed to be an excellent idea, and the English Electric KDF-9 and the Burroughs B5000 computers both implemented the scheme, although it obviously added to their hardware complexity.

Given that registers were expensive resources, the question of how deep the stack should be arose. After all, the B5000 used two registers only, along with an automatic push-down into memory, if more than two intermediate results required storage. This seemed reasonable. As Knuth had pointed out in an analysis of many Fortran programs, the overwhelming majority of expressions required only one or two registers.

Yet, the idea of an expression stack proved questionable, particularly after the advent in the mid-1960s of architectures with register banks. These architectures sacrificed compilation simplicity for any gain in execution speed. The stack organization restricted use of a scarce resource to a fixed strategy. But sophisticated compilation algorithms use registers more economically, given

the flexibility of specifying individual registers in each instruction.

Storing return addresses in the code

The subroutine jump instruction, invented by D. Wheeler, deposits the program counter value to be restored when the subroutine terminates. The challenge involves choosing the place to deposit the value. In several computers, particularly minicomputers but also the CDC 6000 mainframe, a jump instruction to location *d* would deposit the return address at *d*, then continue execution at location *d*+1:

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

This was a bad idea for at least two reasons. First, it prevented calling a subroutine recursively. Algol introduced recursion and caused much controversy because these procedure calls could no longer be handled in this simple way given that a recursive call would overwrite the previous call's return address. Hence, the return address had to be fetched from the fixed place dictated by the hardware and redeposited in a place unique to the recursive procedure's particular incarnation. This overhead was unacceptable to many computer designers and users, forcing them to declare recursion undesirable, useless, and forbidden. They refused to acknowledge that the difficulty arose because of their inadequate call instruction.

Depositing the addresses in a general-purpose register leaves compiler designers freedom of choice while keeping the basic subroutine instruction as efficient as possible.

Second, this solution proved a bad idea because it prevented multiprocessing. Given that program code and data were not kept separate, each concurrent process had to use its own copy of the code.

Some later hardware designs, notably the RISC architectures of the 1990s, accommodated recursive procedure calls by introducing specific registers dedicated to stack addressing and depositing return addresses relative to their value. Depositing the addresses in a general-purpose register, presuming the availability of a register bank, is probably best because it leaves freedom of choice to the compiler designer while keeping the basic subroutine instruction as efficient as possible.

Virtual addressing

Like compiler designers, operating system designers had their own favorite ideas they wanted implemented. Such wishes appeared with the advent of multiprocessing and time-sharing, concepts that gave birth to operating systems in general.

The guiding idea was to use the processor optimally, 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, quasiconcurrently. 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 enough processes to make multiprocessing beneficial.

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

This clever and complex scheme, useful in its time, posed some problems. It practically required all modern hardware to feature page tables and address mapping and to hide the cost of indirect addressing—not to mention 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 multiuser mode. But this has become a questionable idea because semiconductor memories have become so large that mapping and outplacing are no longer beneficial. Yet, the overhead of the complex mechanism of indirect addressing remains with us.

Ironically, virtual addressing is still used for a purpose for which it was never intended: Trapping references to nonexistent objects, against using NIL pointers. NIL is represented by 0, yet the page at address 0 is never allocated. This trick misuses the heavy virtual addressing scheme and should have been resolved.

Complex instruction sets

Early computers featured small sets of simple instructions because they operated with a minimal amount of expensive circuitry. With hardware becoming cheaper, the temptation rose to incorporate more complicated instructions 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, developers sought to accommodate certain language constructs

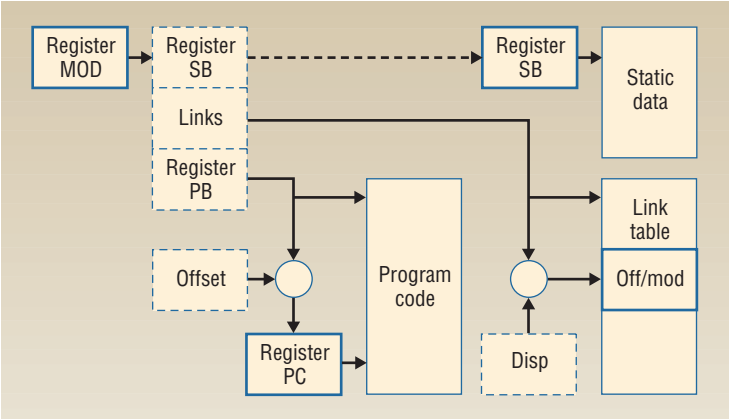

Figure 1. Module storage organization. A dedicated register MOD points to module M's descriptor, which contains the procedure P currently being executed. Register PC is the regular program counter. Register SB contains the address of M's data segment, including M's static, global variables. All these registers change their values whenever the system calls an external procedure.

with correspondingly tailored instructions such as Algol's for statement or instructions for recursive procedure calls. Feature-tailored instructions were a smart idea because they contributed to code density, an important factor when memory was a scarce resource that consisted of 64 Kbytes or less.

This trend set in as early as 1963. The Burroughs B5000 machine not only accommodated many of Algol's complicated features, it combined a scientific computer with a character-string machine and 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 from the programmer's perspective. However, internally the individual machines differed vastly. The low-end machines were microprogrammed, the 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 Semiconductor's 32000.

The NS processor offers a fine example of a complex-instruction-set computer (CISC). Compressing frequent instruction patterns into a single instruction improved code density significantly and reduced memory accesses, increasing execution speed.

The NS processor accommodated, for example, the new concept of modules and separate compilation with an appropriate call instruction. Code segments were linked when loaded, the compiler having provided tables with linking information. Minimizing the number of linking operations, which replace references to the link tables by absolute addresses, is certainly a good idea. The scheme, which simplifies the linker's task, leads to the storage organization for every module, as Figure 1 shows.

A dedicated register MOD points to module M's descriptor, which contains the procedure P currently being executed. Register PC is the regular program counter. Register SB contains the address of M's data segment, including M's static, global variables. All these registers change their values whenever the system calls an external procedure. To speed up this process, the processor offers the call external procedure (CXP) in addition to the regular branch subroutine (BSR). A pair of corresponding returns, RXP and RTS, is also available.

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 the system obtains the address of M's descriptor and also the offset of P within M's code segment. From the descriptor, the system then obtains M's data segment and loads it into SB—all with a single, short instruction. However, what was gained in linking simplicity and code density must be paid for somewhere, namely by an increased number of indirect references and, incidentally, by additional hardware—the MOD and SB registers.

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

Several years after the Oberon compiler had been built and released, new, faster versions of the processor appeared. They went with the trend of implementing frequent, simple instructions directly with hardware and letting an internal microcode interpret the complex instructions. As a result, those language-oriented instructions became rather slow compared to the simple operations. So I programmed a new version of the compiler that didn't use the sophisticated instructions.

This produced astonishing results. The new code executed considerably faster, making it apparent that the computer architect and the compiler designers had optimized in the wrong place.

Indeed, in the early 1980s, advanced microprocessors began to compete with old mainframes, featuring complex and irregular instruction sets. These 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. Around 1990, the reaction became manifest in the form of RISC machines—notably the ARM, MIPS, and Sparc architectures. They featured a small set of simple instruc-

tions, 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. These architectures debunked CISCs as a bad idea.

PROGRAMMING LANGUAGE FEATURES

Programming languages offer a fertile ground for controversial ideas. Some of these were not only questionable but also known to be bad from the outset. The features proposed for Algol in 1960 and for some of its successors[1] provide an excellent example.

Most people consider a programming language merely as code with the sole purpose of constructing software for computers to run. However, a language is a computation model, and programs are formal texts amenable to mathematical reasoning. The model must be defined so that its semantics are delineated without reference to an underlying mechanism, be it physical or abstract.

Choosing the equal sign to denote assignment is a notoriously bad example that goes back to Fortran in 1957.

This makes explaining a complex set of features and facilities in large volumes of manuals appear as a patently bad idea. Actually, a language is characterized not so much by what it lets us program as by what it keeps us from expressing. As E.W. Dijkstra observed, the programmer's most difficult, daily task is to not mess things up. The first and most noble duty of a language is thus to help in this eternal struggle.

Notation and syntax

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

Choosing the equal sign to denote assignment is one notoriously bad example that goes back to Fortran in 1957 and has been copied blindly by armies of language designers since. This bad idea overthrows a century-old tradition to let = denote a comparison for equality, a predicate that is either true or false. But Fortran made this symbol 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. Thus, x = y does not mean the same thing as y = x. Algol corrected this mistake with a simple solution: Let assignment be denoted by :=

This might appear as nitpicking to programmers accustomed to the equal sign meaning assignment. But mixing up assignment and comparison truly is a bad idea because it requires using another symbol for what the equal sign traditionally expresses. Comparison for equality became denoted by the two characters == (first in C). This is an ugly consequence that gave rise to similar bad ideas using ++, --, &&, and so on.

Some of these operators exert side effects in C, C++, Java, and C#, a notorious source of programming mistakes. It might be acceptable to, for example, let ++ denote incrementation by 1, if it would not also denote the incremented value, thereby allowing expressions with side effects. The trouble lies in the elimination of the fundamental distinction between statement and expression. The former is an instruction to be executed, the latter a value to be computed and evaluated.

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

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

We might as well postulate a new algebra. I find absolutely surprising the equanimity with which the programmer community worldwide has accepted this notational monster. In 1962, the postulating of operators being right-associative in the APL language made a similar break with established convention. Now x+y+z suddenly stood for x+(y+z), and x-y-z for x-y+z.

Algol's conditional statement provides a case of unfortunate syntax rather than merely poor symbol choice, offered in two forms, with S0 and S1 being statements:

```
if b then S0
if b then S0 else S1
```

This definition has given rise to an inherent ambiguity and become 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 such as if, while, for, or case:

```
if b then S0 end
if b then S0 else S1 end
```

The goto statement

In the bad-ideas hall of shame, the goto statement has been the villain of many critics. It serves as the direct counterpart in languages to the jump statement in instruction sets and can be used to construct conditional as well as repeated statements. But it also lets programmers construct any maze or mess of program flow, defies any regular structure, and makes structured reasoning about such programs difficult if not impossible.

Our primary tools for comprehending and controlling complex objects are structure and abstraction. We break an overly complex object into parts. The specification of the whole is based on the specifications of the parts. The goto statement became the prototype of a bad programming language idea because it could break the boundaries of the parts and invalidate their specifications.

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. Hoare's rules express this formally as

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

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

Enough has been said and written about this nonfeature to convince almost everyone that it is a primary example of a bad idea. Pascal's designer retained the goto statement as well as the if statement without a 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 understands the problem except for the designers of the latest commercial programming languages, such as C#.

Switches

If a feature is a bad idea, then features built atop it are even worse. This rule can be demonstrated by the switch, which is essentially an array of labels. Assuming, for example, labels L1, … L5, a switch declaration in Algol could 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 encourages a programming mess, the switch makes it impossible to avoid.

C.A.R. Hoare proposed a most suitable replacement of the switch 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 bastard formulation 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, it is a goto in disguise. A bad concept in a bad notation, this example stems from C.

Algol's complicated for statement

Algol's designers recognized that certain frequent cases of repetition would better be expressed by a more concise form than in combination with goto statements. They introduced the for statement, which is particularly convenient in use with arrays, as 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, it was infected with the bad 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
```

Further, these elements could be general expressions:

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

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 quickly concocted pathological cases, demonstrating the concept's absurdity:

```
for i := 1 step 1 until i+1 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 keep the primary purpose of a construct in mind and be wary of exaggerated generality and complexity, which can easily become counterproductive.

Algol's name parameter

Algol introduced procedures and parameters in a much greater generality than known in older languages such as Fortran. In particular, parameters were seen as in traditional mathematics of functions, where the actual parameter textually replaces the formal parameter.[2] 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 programmer's intention. To prevent this frequent, misleading case, Algol's developers postulated a second kind of parameter: the *value parameter*. It meant that a local variable, x′, is to be allocated and then initialized with the value of the actual parameter, x. With

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

the above call would be interpreted as

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

Algol introduced procedures and parameters in a much greater generality than known in older languages such as Fortran.

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

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. Reflection reveals that every name parameter must be implemented as an anonymous, parameterless procedure. To avoid paying for the hidden overhead, a cleverly designed compiler might optimize certain cases. But mostly inefficiency is caused, which could easily have been avoided.

We could 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 to pass results back to the caller. This suggestion, however, led to the replacement of the name parameter by the reference parameter in later languages such as Algol W, Pascal, and Ada.

For today, the message is this: Be skeptical about 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 including the feature.

Loopholes

I consider the loophole one of the worst features ever, even though I infected Pascal, Modula, and even Oberon

with this deadly virus. The loophole lets the programmer breach the compiler's type checking and say "Don't interfere, as I am smarter than the rules." Loopholes take many forms. Most common are explicit type transfer functions, such as

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

But they can also be disguised as absolute address specifications or by variant records, as in Pascal. In the preceding examples, the internal representation of integer *i* is to be interpreted as a floating-point number *x*. This can only be done with knowledge about number representation, which should not be necessary when dealing with the abstraction level the language provides.

The presence of a loophole facility usually points to a deficiency in the language proper, revealing that certain things could not be expressed.

Pascal and Modula[3] at least displayed loopholes honestly. In Oberon, they are present only through a small number of functions encapsulated in a pseudomodule called *SYSTEM*, which must be imported and is thus explicitly visible in the heading of any module that uses such low-level facilities.

This may sound like an excuse, but the loophole is nevertheless a bad idea. It was introduced to implement complete systems in a single programming language. For example, a storage manager must be able to look at storage as a flat array of bytes without data types. It must be able to allocate blocks and recycle them, independent of any type constraints. The device driver provides another example that requires a loophole. Earlier computers used special instructions to access devices. Later, programmers assigned devices specific memory addresses, *memory mapping* them. Thus, the idea arose to let absolute addresses be specified for certain variables, as in Modula. But this facility can be abused in many clandestine ways.

Evidently, the normal user will never need to program either a storage manager or a device driver, and hence has no need for those loopholes. However—and this is what really makes the loophole a bad idea—programmers still have the loophole at their disposal. Experience showed that normal users will not shy away from using the loophole, but rather enthusiastically grab onto it as a wonderful feature that they use wherever possible. This is particularly so if manuals caution against its use.

The presence of a loophole facility usually points to a deficiency in the language proper, revealing that certain things could not be expressed that might be important. For example, the type *ADDRESS* in Modula had to be used to program data structures with different element types. A strict, static typing strategy, which demanded that every pointer be statically associated with a fixed type and could only reference such objects, made this impossible. 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. 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. Oberon[4] introduced the clean solution of type extension, called *inheritance* in object-oriented languages. Now it became possible to declare a pointer as referencing a given type, and it could point to any type that was an extension of the given type. This made it possible to construct inhomogeneous data structures and use them with the security of a reliable type-checking system. An implementation must check at runtime if and only if it is not possible to check at compile time.

Programs expressed in 1960s languages were full of loopholes that made them utterly error-prone. But there was no alternative. The fact that developers can use a language like Oberon to program entire systems from scratch without using loopholes, except in the storage manager and device drivers, marks the most significant progress in language design over the past 40 years.

MISCELLANEOUS TECHNIQUES

These bad ideas stem from the wide area of software practice, or rather from the narrower area of the author's experiences with it, dating back 40 years. Yet the lessons learned then remain valid today. Some reflect more recent practices and trends, mostly supported by the abundant availability of hardware power.

Syntax analysis

The 1960s saw the rise of syntax analysis. Using a formal syntax to define Algol 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 gave rise to many activities for automatic syntax analysis on a mathematically rigorous basis. This work created the notions of top-down and bottom-up principles, the recursive descent technique, and measures for symbol lookahead and backtracking. Accompanying this were efforts to define language semantics more rigorously by piggybacking semantic rules onto corresponding syntax rules.

As happens with new fields of endeavor, research went beyond the needs of the hour. Developers built increasingly powerful parser generators, which managed to handle ever more general and complex grammars. Although

an intellectual achievement, this development had a less positive consequence. 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.

Yet no such tool would give any indication how that syntax could be improved. Designers had ignored both the issue of efficiency and that a language serves the human reader, not just the automatic parser. If a language poses difficulties to parsers, it surely also poses difficulties for the human reader. Many languages would be clearer and cleaner had their designers been forced to use a simple parsing method.

My own experience strongly supports this statement. After contributing to the development of parsers for precedence grammars in the 1960s, and having used them for the implementation of Euler and Algol W, I decided to switch to the simplest parsing method for Pascal: top-down, recursive descent. The experience was most encouraging, and I have stuck to it to this day with great satisfaction.

If a language poses difficulties to parsers, it surely also poses difficulties for the human reader.

The drawback is that considerably more careful thought must go into the syntax's design prior to publication and any implementation effort. This additional effort will be more than compensated for in the later use of both the language and the compiler.

Extensible languages

Computer scientists' fantasies in the 1960s knew no bounds. Spurred by the success of automatic syntax analysis and parser generation, some proposed the idea of the flexible, or at least extensible, language, envisioning that a program would be preceded by syntactic rules that would then guide the general parser while parsing the subsequent program. Further, the syntax rules also could be interspersed anywhere throughout the text. For example, it would be possible to use a particularly fancy private form of the for statement, 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 now it was possible to define a language on the fly. However, the difficulties encountered when trying to specify what these private constructions should mean soon dashed these high hopes. As a consequence, the intriguing idea of extensible languages quickly faded away.

Tree-structured symbol tables

Compilers construct symbol tables. The compiler builds up the tables while processing declarations and searches them while processing statements. In languages that allow nested scopes, every scope is represented by its own table.

Traditionally, these tables are binary trees to allow fast searching. Having followed this long-standing tradition, I dared to doubt the benefit of trees when implementing the Oberon compiler. When doubts occurred, I quickly became convinced that tree structures are not worthwhile for local scopes. In the majority of cases, procedures contain a dozen or even fewer local variables. Using a linked linear list is then both simpler and more effective.

Programs written 30 or 40 years ago declared most variables globally. As there were many of them, the tree structure was justified. In the meantime, however, skepticism about the practice of using global variables has been on the rise. Modern programs do not feature many globals, and hence in this case a tree-structured table is hardly suitable.

Modern programming systems consist of many modules, each of which probably contains some globals, but not hundreds. The many globals of early programs have become distributed over numerous modules and are referenced not by a single identifier, but by a name combination *M.x*, which defines the initial search path.

Using sophisticated data structures for symbol tables was evidently a poor idea. Once, we had even considered balanced trees.

Using the wrong tools

Although using the wrong tools is an obviously bad idea, often we discover a tool's inadequacy only after having invested substantial effort into building and understanding it. Investing this effort, unfortunately, gives the tool perceived value regardless of its functional worth. This happened to my team when we implemented the first Pascal compiler in 1969.

The tools available for writing programs were assembly code, Fortran, and an Algol compiler. The Algol compiler was so poorly implemented we dared not rely on it, and working with assembly code was considered dishonorable. There remained only Fortran.

Hence, our naïve plan was to use Fortran to construct a compiler for a substantial subset of Pascal that, when completed, would be translated into Pascal. Afterward, we would employ the classic bootstrapping technique to complete, refine, and improve the compiler.

This plan crashed in the face of reality. When we completed step one—after about one person-year's labor—it turned out that translating Fortran code into Pascal was impossible. That program was so much determined by Fortran's features, or rather its lack of any, that our only option was to write the compiler afresh. Because Fortran did not feature pointers and records, we had to

squeeze symbol tables into the unnatural form of arrays. Nor did Fortran have recursive subroutines. Hence, we had to use the complicated table-driven bottom-up parsing technique with syntax represented by arrays and matrices. In short, employing the advantages of Pascal required completely rewriting and restructuring the compiler.

This incident revealed that the apparently easiest way is not always the right way. But difficulties also have their benefits: Because no Pascal compiler had yet become available, we had to write the entire program for compiling a substantial subset of Pascal without testing feedback. This proved an extremely healthy exercise and would be even more so today, in the era of quick trial and interactive error correction.

After we believed the compiler to be complete, we banished one member of our team to his home to translate the program into a syntax-sugared, low-level language that had an available compiler. He returned after two weeks of intensive labor, and a few days later the compiler written in Pascal compiled the first test programs.

Never do programs contain so few bugs as when no debugging tools are available.

The exercise of conscientious programming proved extremely valuable. Never do programs contain so few bugs as when no debugging tools are available.

Thereafter, we could use bootstrapping to obtain new versions that handled more of Pascal's constructs and produced increasingly refined code. Immediately after the first bootstrap, we discarded the translation written in the auxiliary language. Its character had been much like that of the ominous low-level language C, published a year later.

After this experience, we had trouble understanding why the software engineering community did not recognize the benefits of adopting a high-level, type-safe language instead of C.

PROGRAMMING PARADIGMS

Several programming paradigms have come into and out of vogue over the past few decades, making contributions of varying importance to the field.

Functional programming

Functional languages had their origin in Lisp.[5] They have undergone a significant amount of development and change and have been used to implement both small and large software systems. I have always maintained a skeptical attitude toward such efforts.

What characterizes functional languages? It has always appeared that it was their form, that the entire program consists of function evaluations—nested, recursive, parametric, and so on. Hence, the term *functional*. However, the core idea is that functions inherently have no state. This implies that there are no variables and no assignments. Immutable function parameters—variables in the mathematical sense—take the place of variables. As a consequence, freshly computed values cannot be reassigned to the same variable, overwriting its old value. This explains why repetition must be expressed by recursion. A data structure can at best be extended, but no change can be made to 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 stateless model of computation atop a machine whose most eminent characteristic is state seems an odd idea at the least. The gap between model and machinery is wide, which makes bridging it costly. No hardware support feature can gloss over this: It remains a bad idea in practice.

The protagonists of functional languages have also recognized this over time. 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 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 its use of strictly local objects. This discipline can also be practiced using conventional, imperative languages, which have subscribed to the notions of nested structures, functions, and recursion for some time.

Functional programming implies much more than avoiding goto statements, however. It also implies restriction to local variables, perhaps with the exception of a few global state variables. It probably also considers the nesting of procedures as undesirable. The B5000 computer apparently was right, after all, in restricting access to strictly local and global variables.

Many years ago, and with increasing frequency, several developers claimed that functional languages are the best vehicle for introducing parallelism—although it would be more to the point to say "to facilitate compilers to detect opportunities for parallelizing a program." After all, determining which parts of an expression can be evaluated concurrently is relatively easy. More important is that parameters of a called function can be evaluated concurrently, provided that side effects are banned—which cannot occur in a truly functional language. As this may be true and perhaps of marginal benefit, object orientation offers a more effective way to let a system make good use of parallelism, with each object representing its own behavior in the form of a private process.

Logic programming

Although logic programming has also received wide attention, only a single well-known language represents this paradigm: Prolog. Principally, Prolog replaces the specification of actions, such as assignment to variables, with 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 to 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, and therefore must understand what is going on and the process of logic inference, the very thing the paradigm's advocates had promised could be ignored.

We must suspect that, because the community desperately desired ways to produce better, more reliable software, its developers were glad to hear of this possible panacea. But the promised advancements never materialized. I sadly recall the exaggerated hopes that fueled the Japanese Fifth-Generation Computer project, Prolog's inference machines. Organizations sank large amounts of resources into this unwise and now largely forgotten idea.

Some developers claimed that functional languages are the best vehicle for introducing parallelism.

Object-oriented programming

In contrast to functional and logic programming, object-oriented programming rests on the same principles as conventional, procedural programming. OOP's character is imperative. It describes a process as a sequence of state transformations. 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 that cause 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 is therefore well suited to model complex systems with complex behavior. Not surprisingly, OOP has its origins in the field of system simulation. Its success in the field of software system design speaks for itself, starting with the language Smalltalk[6] and continuing with Object Pascal, C++, Eiffel, Oberon, Java, and C#.

The original Smalltalk implementation provided a convincing example of its suitability. The first language to feature windows, menus, buttons, and icons, it provides a perfect example of visible objects. The direct modeling of actors diminished the importance of proving program correctness analytically because the original specification is one of behavior, rather than a static input-output relationship.

Nevertheless, we may wonder where the core of the new paradigm would hide and how it would differ essentially from 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. Whether this change in terminology expresses an essential paradigm shift or amounts to no more than a sales trick remains an open question.

Much can be learned from analyzing not only bad ideas and past mistakes but also good ones. Although the collection of topics here may appear accidental, and is certainly incomplete, I wrote it from the perspective that computing science would benefit from more frequent analysis and critique, particularly self-critique. After all, thorough self-critique is the hallmark of any subject claiming to be a science.

References

1. P. Naur, "Report on the Algorithmic Language ALGOL 60," *Comm. ACM*, May 1960, pp. 299-314.
2. D.E. Knuth, "The Remaining Trouble Spots in ALGOL 60," *Comm. ACM*, Oct. 1967, pp. 611-618.
3. N. Wirth, *Programming in Modula-2*, Springer-Verlag, 1982.
4. N. Wirth, "The Programming Language Oberon," *Software—Practice and Experience*, Wiley, 1988, pp. 671-691.
5. J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine," *Comm. ACM*, May 1962, pp. 184-195.
6. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

Niklaus Wirth *is a professor emeritus of informatics at ETH Zurich. His research interests include programming languages, integrated software environments, and hardware design using field-programmable gate arrays. Wirth received a PhD in electrical engineering from the University of California at Berkeley. Contact him at wirth@inf.ethz.ch.*