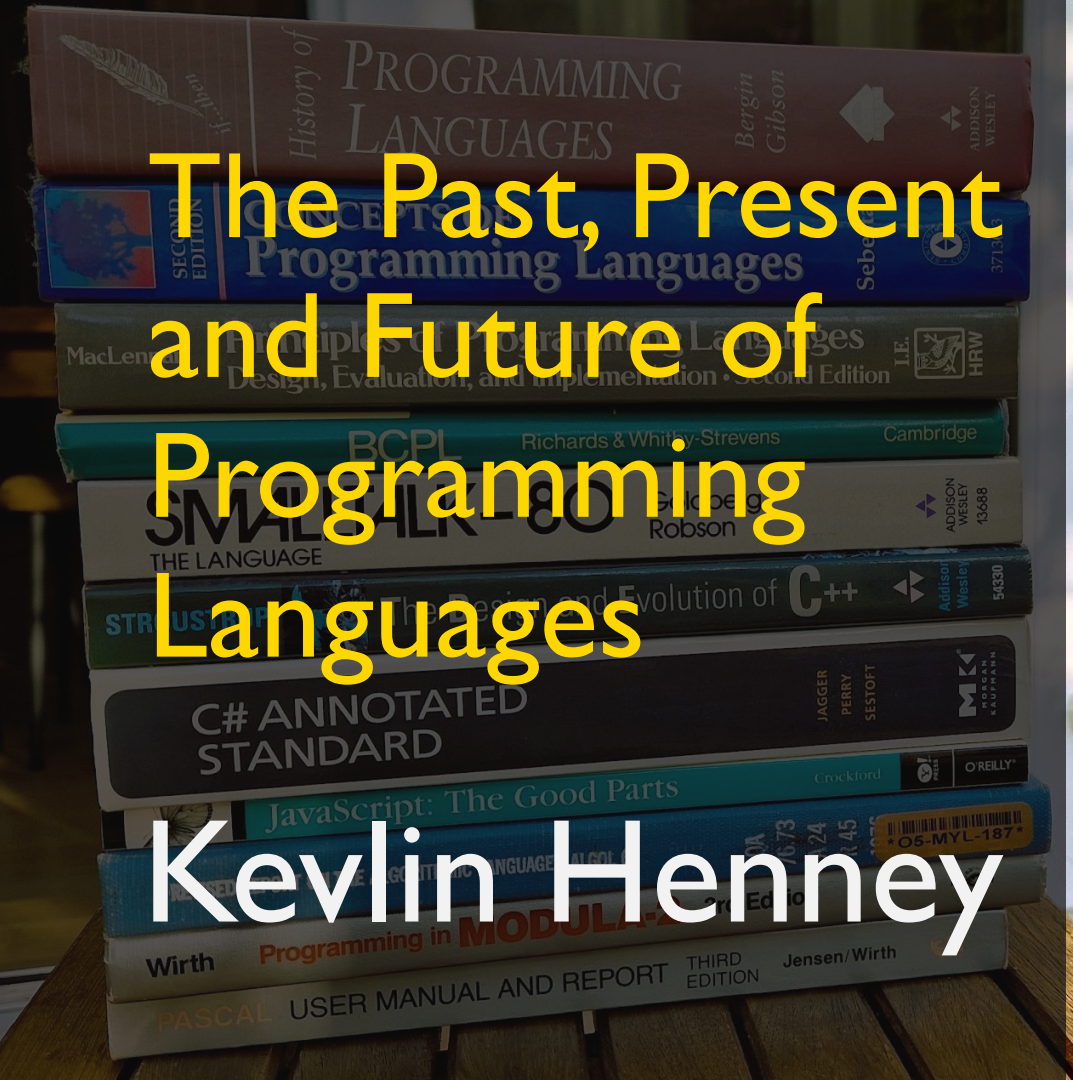
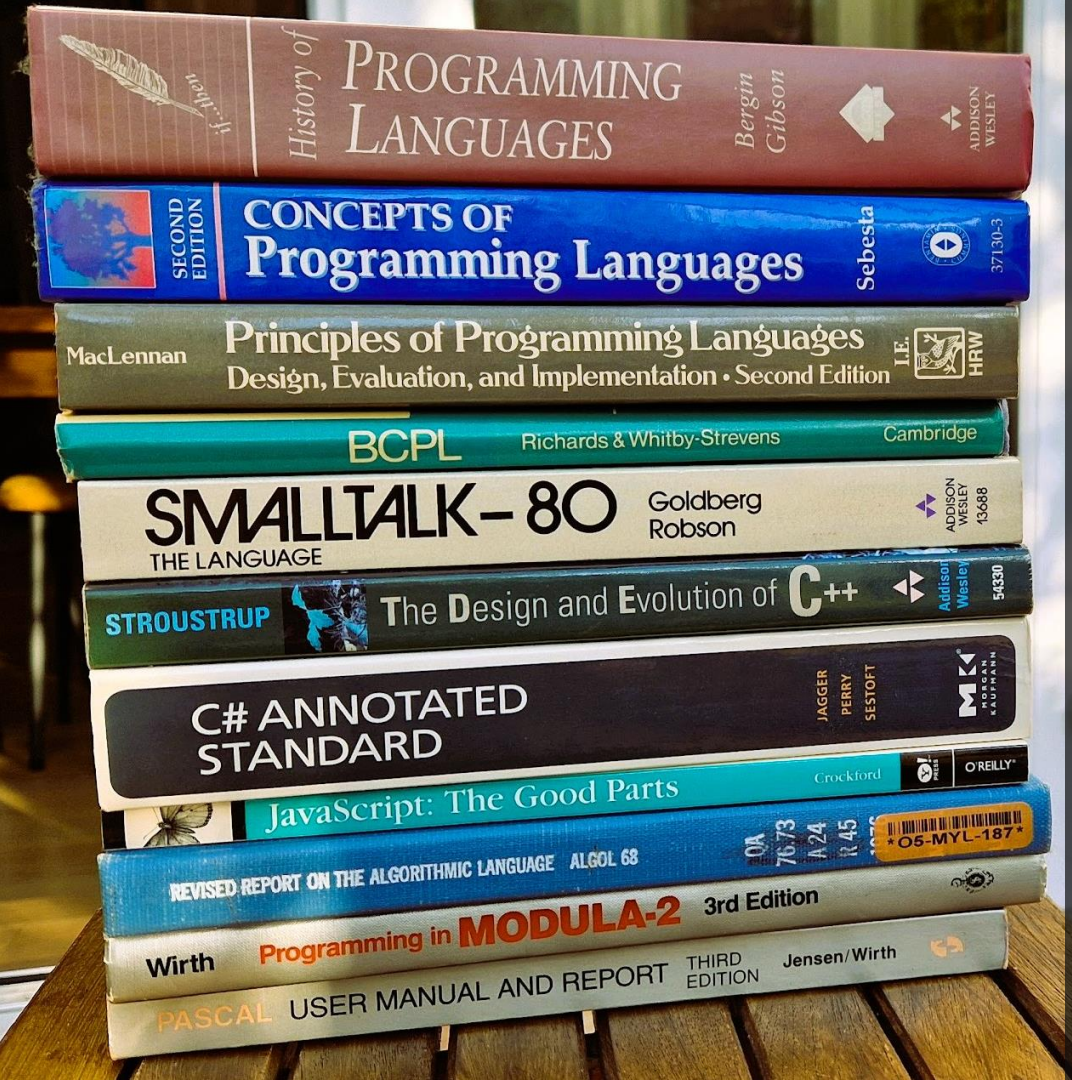
The Past, Present and Future of Programming Languages

Kevlin Henney







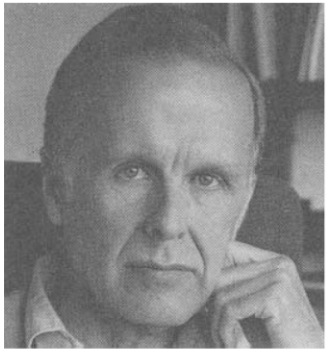
@KevlinHenney
@kevlin.bsky.social
linkedin.com/in/kevlin
@kevlin@mastodon.social
threads.net/@kevlin.henney
instagram.com/kevlin.henney
kevlinhenney.medium.com
kevlin@curbralan.com
about.me/kevlin

The past, present and future walked into a bar. It was a tense situation.


NASA
ARTEMIS

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

© 1978 ACM 0001-0782/78/0800-0613 $00.75

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

The Paradigms of Programming

Robert W. Floyd
Stanford University



Paradigm(pæ·radim, −dəim) . . . [a. F. *paradigme*, ad. L. *paradigma*, a. Gr. παραδειγμα pattern, example, f. παραδεικνυ·ναι to exhibit beside, show side by side. . .]
1. A pattern, exemplar, example.
1752 J. Gill *Trinity* v. 91
The archetype, paradigm, exemplar, and idea, according to which all things were made.
From the Oxford English Dictionary.

Today I want to talk about the paradigms of programming, how they affect our success as designers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formulated by Dijkstra [6], Wirth [27, 29], and Parnas [21], among others, consists of two phases.

In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of simpler subproblems. In programming the solution of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangularized system. This gradual decomposition is continued until the subproblems that arise are simple enough to cope with directly. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the *i*th variable from the *i*th equation. Yet further decomposition would yield a fully detailed algorithm.

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

Programming languages appear to be in trouble.



General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

© 1978 ACM 0001-0782/78/0800-0613 $00.75

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Python
C++
Java
C
C#
JavaScript
Visual Basic
Go
SQL
Fortran

TIOBE Index for September 2024

September Headline: Programming language C at lowest position ever

One of the mothers of all programming languages, C, has dropped one position in the TIOBE index this month. C is currently at position #4, which is its lowest position ever since the start of the TIOBE index in 2001. Is C losing its mojo? C is a relatively small programming language, which generates very performant code. As a consequence, it has been the favorite of embedded systems for a very long time. The only serious drawback of C so far was that it is not designed to scale. Large C programs are hard to maintain because of the lack of object oriented features. Now that embedded systems tend to grow in functionality and thus in code size, and since more and more embedded compilers have good C++ support, there is tendency to switch from C to C++. Recently another threat for C raised its head. The US government wants to put a ban on memory unsafe languages such as C in favor of memory safe languages such as Rust. This is the other tendency I observe. Companies are checking out Rust as an alternative for C. C might lose its mojo, but it will stay in the TIOBE index top 10 for a very long time. Its installed base is incredible and it is part of a zillion of safety-critical systems around the world.

C
Java
Python
C++
C#

TIOBE Index for September 2020

C
Python
Java
C++
C#

TIOBE Index for September 2021

Python
C
Java
C++
C#

TIOBE Index for September 2022

Python
C
C++
Java
C#

TIOBE Index for September 2023

Python
C++
Java
C
C#

TIOBE Index for September 2024

```
top_5 = {"Python",
         "C++",
         "Java",
         "C",
         "C#"}

for ranking in permutations(top_5):
    ...
```

1	Python	20.17%
2	C++	10.75%
3	Java	9.45%
4	C	8.89%
5	C#	6.08%
6	JavaScript	3.92%
7	Visual Basic	2.70%
8	Go	2.35%
9	SQL	1.94%
10	Fortran	1.78%

1 Python
2 C++
3 Java
4 C
5 C#
6 JavaScript
7 Visual Basic
8 Go
9 SQL
10 Fortran

~68%

1	**Python**
2	**C++**
3	**Java**
4	**C**
5	**C#**

~55%

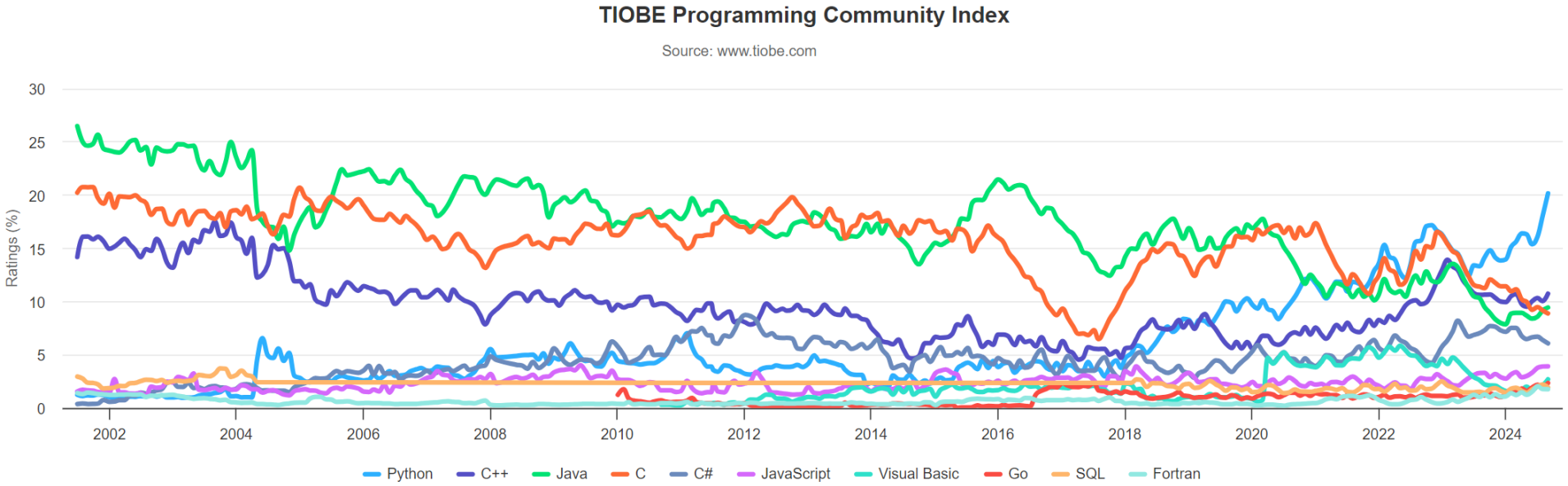
Python
C++
Java
C
C#
JavaScript
Visual Basic
Go
SQL
Fortran

Python
C++
Java
C
C#
JavaScript
Visual Basic
Go
SQL
Fortran

Python
C++
Java
C
C#
JavaScript
Visual Basic
Go
SQL
Fortran

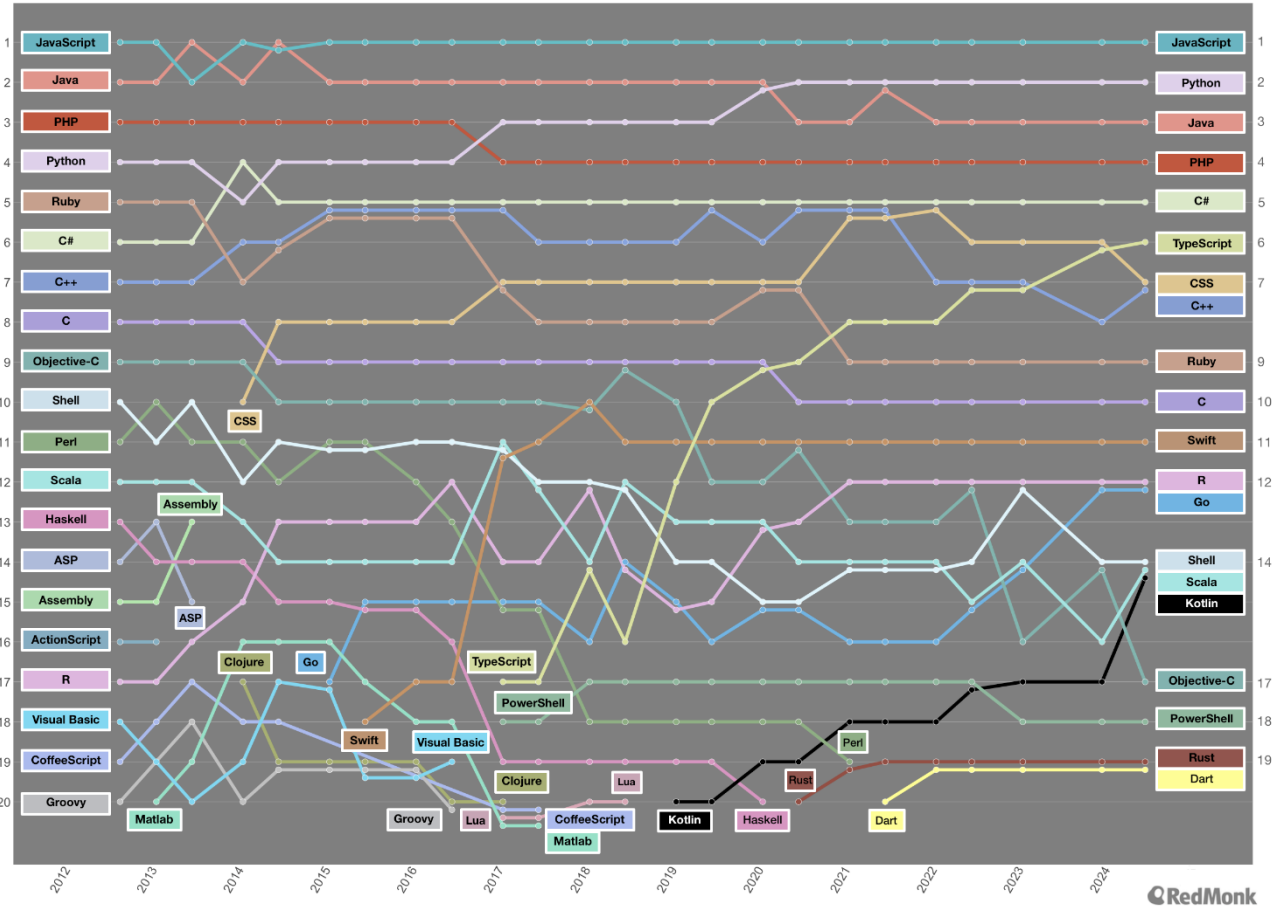
Python
C++
Java
C
C#
JavaScript
Visual Basic
Go
SQL
Fortran
Delphi
MATLAB
PHP
Rust
R
Ruby
Scratch
Kotlin
COBOL
Swift

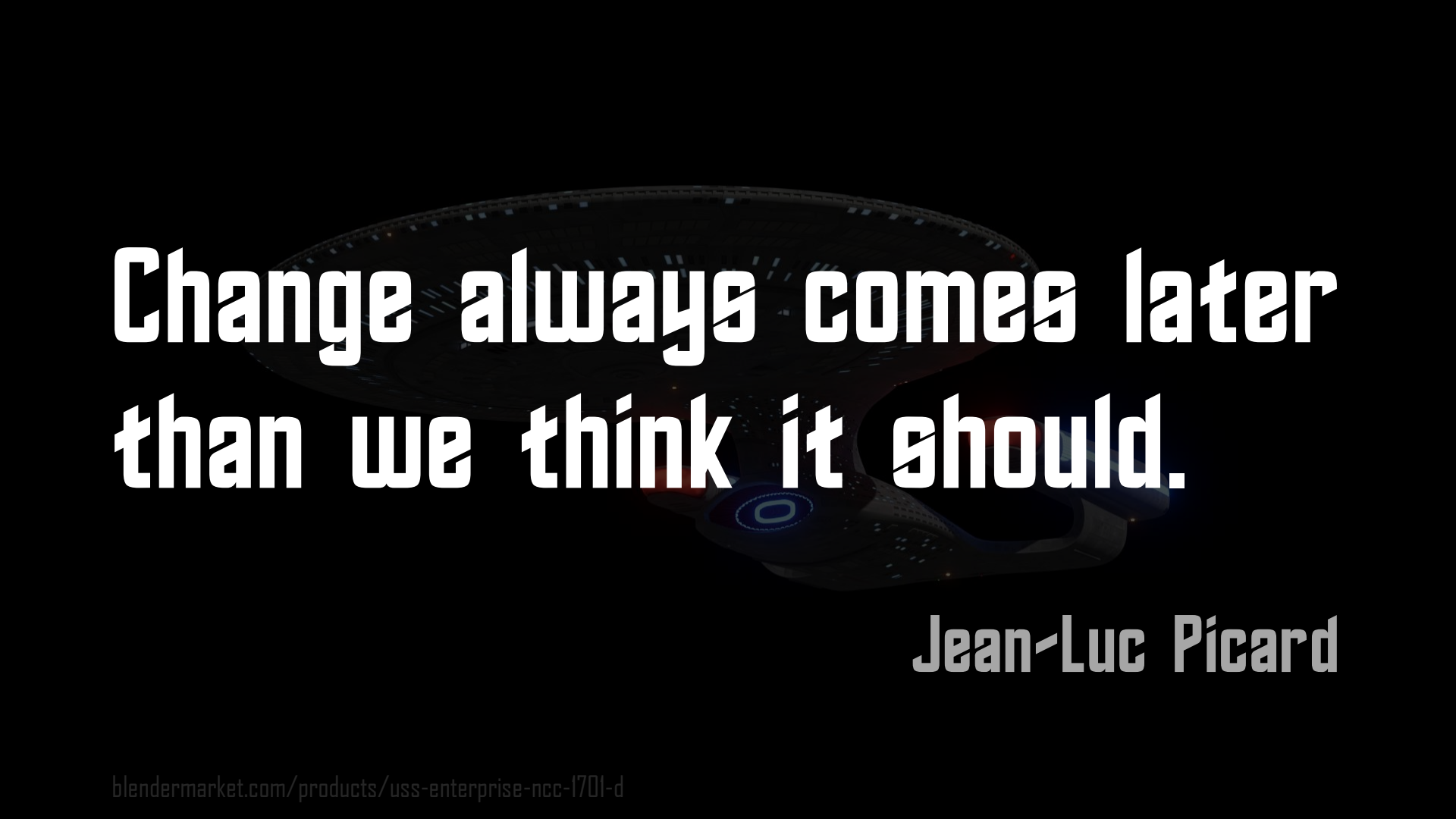
Python
C++
Java
C
C#
JavaScript
Visual Basic
Go
SQL
Fortran
Delphi
MATLAB
PHP
Rust
R
Ruby
Scratch
Kotlin
COBOL
Swift

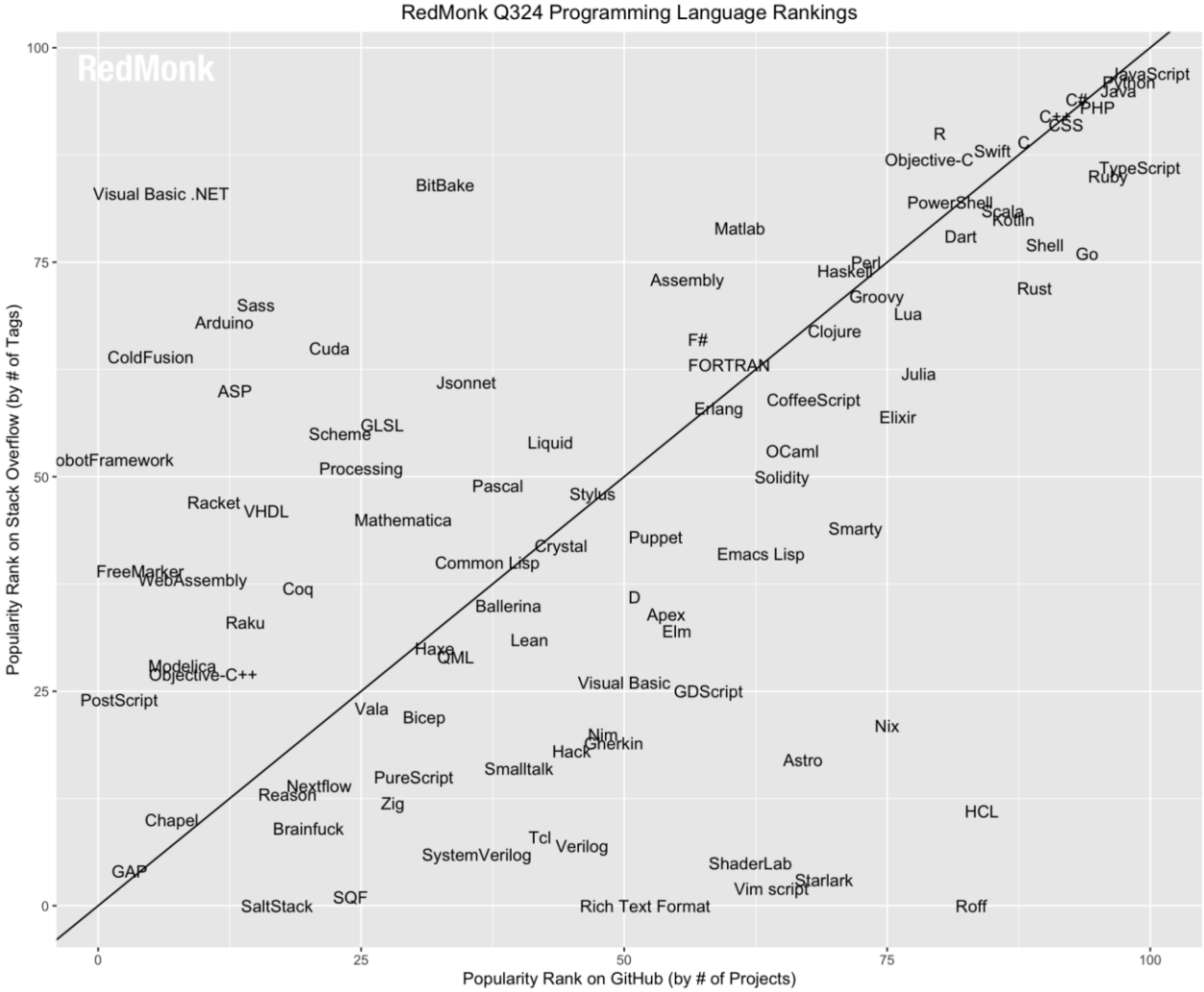

TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
2022
2024
Python
C++
Java
C
C#
JavaScript
Visual Basic
Go
SQL
Fortran

RedMonk Language Rankings

September 2012 - June 2024




Change always comes later than we think it should.
Jean-Luc Picard
blendermarket.com/products/uss-enterprise-ncc-1701-d


RedMonk Q324 Programming Language Rankings
RedMonk
Popularity Rank on Stack Overflow (by # of Tags)
Popularity Rank on GitHub (by # of Projects)
0
25
50
75
100
JavaScript
Python
Java
C#
PHP
C++
CSS
R
Swift
C
Objective-C
TypeScript
Ruby
Visual Basic .NET
BitBake
PowerShell
Scala
Kotlin
Matlab
Dart
Shell
Go
Perl
Haskell
Assembly
Rust
Groovy
Lua
Sass
Arduino
Clojure
F#
Cuda
ColdFusion
FORTRAN
Julia
Jsonnet
ASP
Erlang
CoffeeScript
Elixir
GLSL
Scheme
Liquid
OCaml
obotFramework
Processing
Solidity
Pascal
Stylus
Racket
VHDL
Mathematica
Smarty
Puppet
Crystal
Emacs Lisp
Common Lisp
FreeMarker
WebAssembly
Coq
D
Ballerina
Apex
Elm
Raku
Lean
Haxe
QML
Modelica
Objective-C++
Visual Basic
GDScript
PostScript
Vala
Bicep
Nix
Nim
Gherkin
Hack
Astro
Smalltalk
PureScript
Nextflow
Reason
Zig
HCL
Chapel
Brainfuck
Tcl
Verilog
SystemVerilog
ShaderLab
Starlark
GAP
Vim script
SaltStack
SQF
Rich Text Format
Roff

JavaScript
Python
Java
PHP
C#
TypeScript
CSS
C++
Ruby
C
Swift
Go
R
Shell
Kotlin
Scala
Objective-C
PowerShell
Rust
Dart

JavaScript
Python
Java
PHP
C#
TypeScript
CSS
C++
Ruby
C
Swift
Go
R
Bash
Kotlin
Scala
Objective-C
PowerShell
Rust
Dart

JavaScript
Python
Java
PHP
C#
TypeScript
CSS
C++
Ruby
C
Swift
Go
R
Bash
Kotlin
Scala
Objective-C
PowerShell
Rust
Dart

JavaScript
Python
Java
PHP
C#
TypeScript
CSS
C++
Ruby
C
Swift
Go
R
Bash
Kotlin
Scala
Objective-C
PowerShell
Rust
Dart

IEEE Spectrum

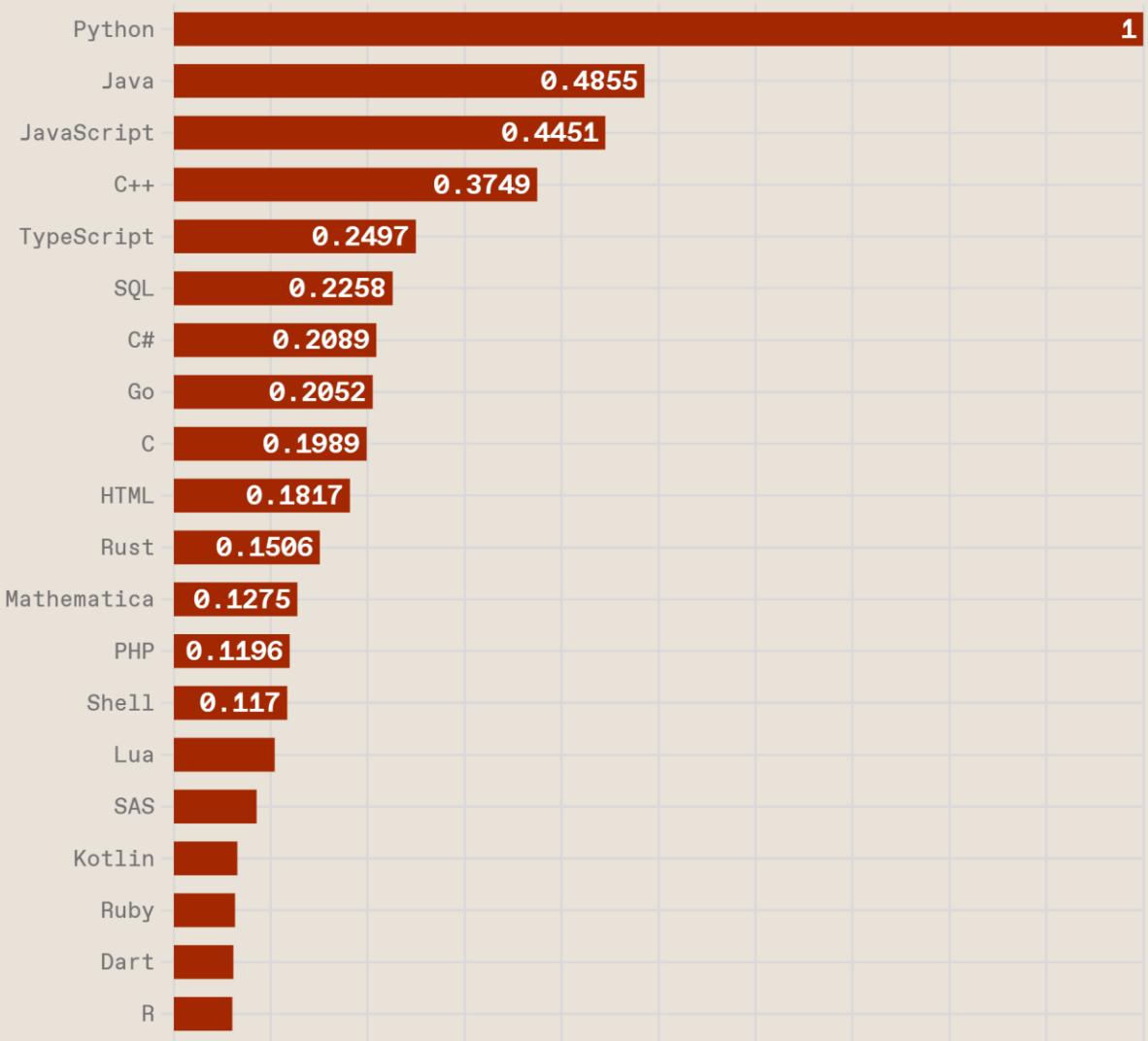
NEWS | COMPUTING

The Top Programming Languages 2024 >

Typescript and Rust are among the rising stars

BY STEPHEN CASS | 22 AUG 2024 | 3 MIN READ |

Stephen Cass is the special projects editor at IEEE Spectrum.


Python
1
Java
0.4855
JavaScript
0.4451
C++
0.3749
TypeScript
0.2497
SQL
0.2258
C#
0.2089
Go
0.2052
C
0.1989
HTML
0.1817
Rust
0.1506
Mathematica
0.1275
PHP
0.1196
Shell
0.117
Lua
SAS
Kotlin
Ruby
Dart
R

Python
Java
JavaScript
C++
TypeScript
SQL
C#
Go
C
HTML
Rust
Mathematica
PHP
Shell
Lua
SAS
Kotlin
Ruby
Dart
R

Python
Java
JavaScript
C++
TypeScript
SQL
C#
Go
C
HTML
Rust
Mathematica
PHP
Bash
Lua
SAS
Kotlin
Ruby
Dart
R

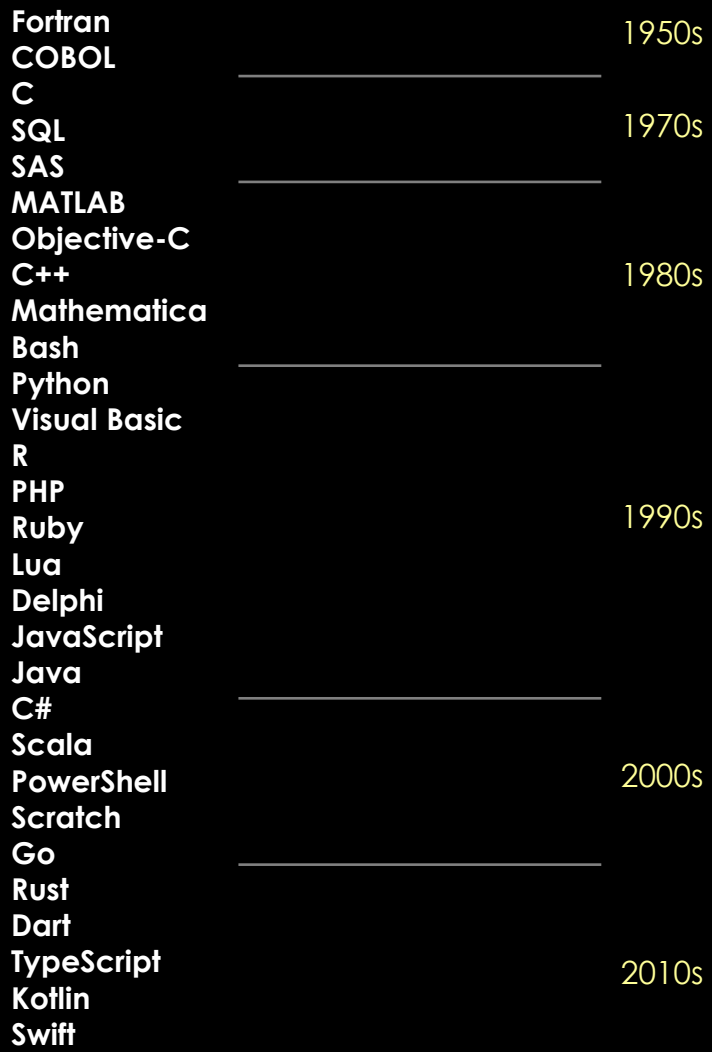
Python
Java
JavaScript
C++
TypeScript
SQL
C#
Go
C
HTML
Rust
Mathematica
PHP
Bash
Lua
SAS
Kotlin
Ruby
Dart
R

Python
Java
JavaScript
C++
TypeScript
SQL
C#
Go
C
HTML
Rust
Mathematica
PHP
Bash
Lua
SAS
Kotlin
Ruby
Dart
R

There are only two kinds of languages: The ones people complain about and the ones nobody uses.

Bjarne Stroustrup

Bash
C
C#
C++
COBOL
Dart
Delphi
Fortran
Go
Java
JavaScript
Kotlin
Lua
MATLAB
Mathematica
Objective-C
PHP
PowerShell
Python
R
Ruby
Rust
SAS
SQL
Scala
Scratch
Swift
TypeScript
Visual Basic


Fortran
COBOL
C
SQL
SAS
MATLAB
Objective-C
C++
Mathematica
Bash
Python
Visual Basic
R
PHP
Ruby
Lua
Delphi
JavaScript
Java
C#
Scala
PowerShell
Scratch
Go
Rust
Dart
TypeScript
Kotlin
Swift
1950s
1970s
1980s
1990s
2000s
2010s

Fortran
COBOL
C
SQL
SAS
MATLAB
Objective-C
C++
Mathematica
Bash
Python
Visual Basic
R
PHP
Ruby
Lua
Delphi
JavaScript
Java
C#
Scala
PowerShell
Scratch
Go
Rust
Dart
TypeScript
Kotlin
Swift

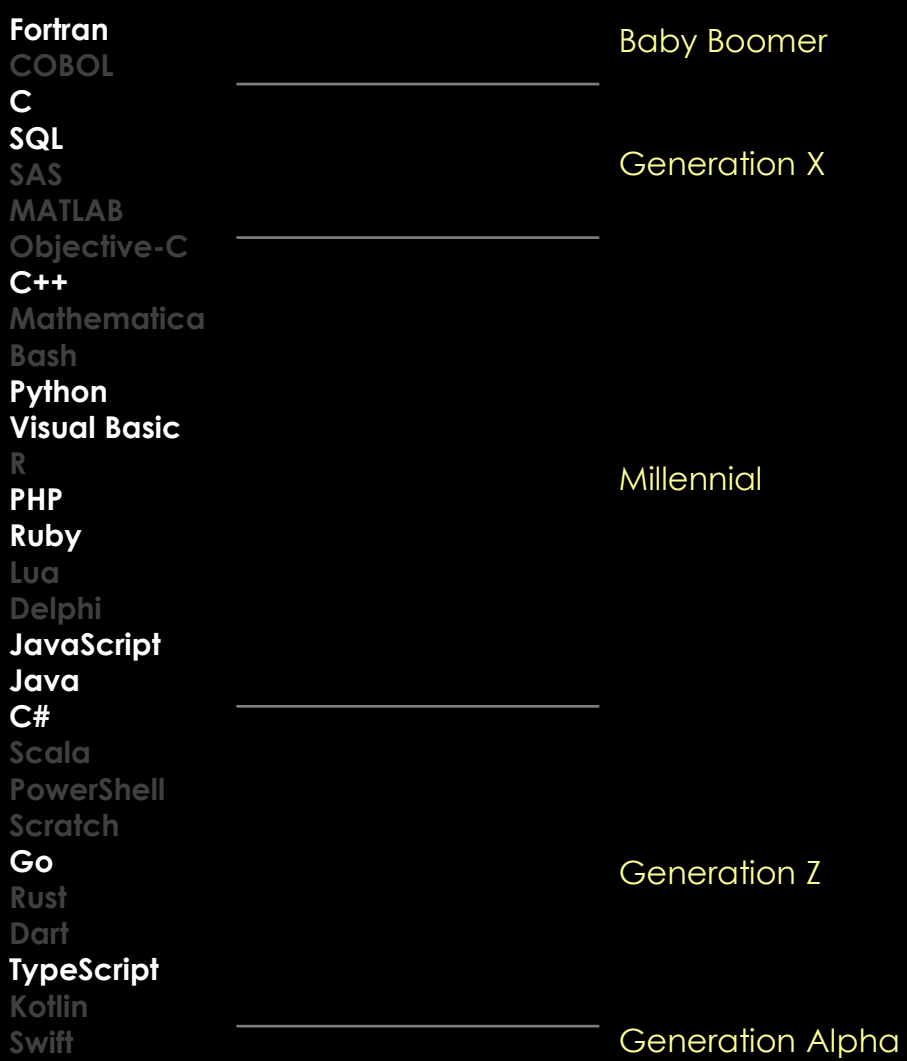
Baby Boomer

Generation X

Millennial

Generation Z

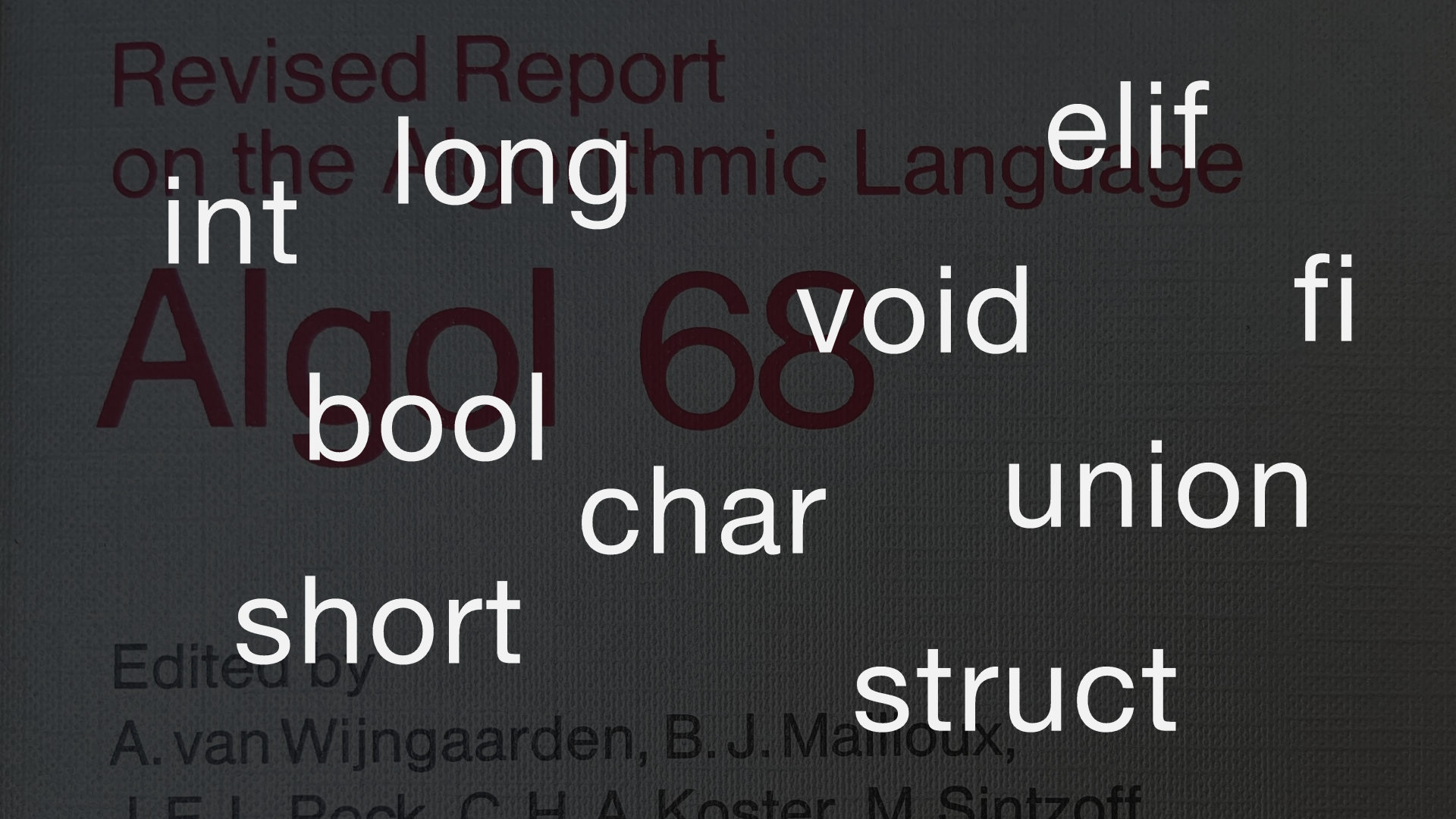
Generation Alpha


Fortran
COBOL
C
SQL
SAS
MATLAB
Objective-C
C++
Mathematica
Bash
Python
Visual Basic
R
PHP
Ruby
Lua
Delphi
JavaScript
Java
C#
Scala
PowerShell
Scratch
Go
Rust
Dart
TypeScript
Kotlin
Swift
Baby Boomer
Generation X
Millennial
Generation Z
Generation Alpha

Revised Report on the Algorithmic Language

Algol 68

Edited by
A. van Wijngaarden, B. J. Mailloux,
J. E. L. Peck, C. H. A. Koster, M. Sintzoff


Revised Report
on the Algorithmic Language
Algol 68
Edited by
A. van Wijngaarden, B. J. Mailloux,
int
long
elif
void
fi
bool
char
union
short
struct

0
1
1
2
3
5
8
13
21
34
55
⋮

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

```
proc fibonacci = (int n) int:
  if n < 2
  then n
  else fibonacci(n - 1) + fibonacci(n - 2)
  fi
```

Are you going to this year's Fibonnaci conference?
It's going to be as big as the last two put together.


WELCOME TO
goto;
copenhagen

A Case against the GO TO Statement.

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

A Case Against the Go To Statement

by Edsger W Dijkstra

EWD215 - 0

A Case against the GO TO Statement.

by Edsger W. Dijkstra

Technological University

Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A*1 **else** *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of($A1, A2, \cdots, An$)), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1, B2 \rightarrow E2, \cdots, Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands



Go To Statement Considered Harmful

by Edsger W Dijkstra

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to perform what is desired, to produce the results for which the program has been written. Yet, once the program has been made, the "making" of the process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process is controlled by a tape recorder. In that case, what do we mean by point 74 of the tape recorder?) We cannot do it by pointing to a place in the text, for the simple reason that the next symbol chosen is dependent upon the input supplied to the program. Normally speaking during processing the latter, part of the program will be performed a number of times, but the total number of times a part of the program will be performed is not known beforehand. The only handle that we have on the process is the current point in the program text to which control has been transferred. The values of variables are meaningless if we do not know how we got there. So we can say that the progress of the process is characterized by the current point in the program text and by the corresponding current values of the variables involved.

The practice of tracing the progress of a process by following the values of variables is a very effective debugging technique. It is also an unpleasant technique because it is so laborious. But whether we like it or not, the debugging of a program cannot be based on any form of reasoning that has the progress of the process as its subject matter.

For that reason I regard the go to statement as harmful in any language in which it is available: the language should not provide it. The reasons for this drastic view are the following.

Let us first consider the go to statement as it occurs in ALGOL 60. Here we have a choice between a textual jump backwards and a textual jump forwards. In connection with the former one can easily express oneself in such a way that a program terminates with certainty in a finite number of steps. With the latter one can do so provided that no jump into the range is made from outside. The remaining go to statements are well disciplined in the sense that they can be understood by a reader who understands the conditional clause and the assignment statement. For the sake of argument suppose that the go to statement in ALGOL 60 had been abolished. Would that have been a great loss? I doubt it.

Then consider the go to statement as it occurs in FORTRAN I. Here I have a very strong feeling that one would have been much better off if it had never been included. For this language the debugging of a program without go to statements would have been a very much easier task than it is now. In FORTRAN the go to statement is indispensable for the formulation of conditional expressions and decision functions. Its use is logically unavoidable.

I now suggest that we should reconsider our past habits and that we should become more conscious of the go to statement's capacity for making a program unclear, so that we can more effectively restrict our use of it.

Let us start our reconsideration by asking ourselves why we sometimes need so acutely a more liberal use of the go to statement than FORTRAN allows us. The reason is undoubtedly that we frequently wish to express a choice between two possibilities and after having performed the one we must decide to perform the next action somewhere else in the program, while after having performed the other we can proceed with the next consecutive statement. The go to statement as such is too primitive; it is too much an invitation to make a mess of one's program.

Conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A*1 **else** *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of (*A*1, *A*2, ···, *An*)), or conditional expressions as introduced by J. McCarthy (*B*1 → *E*1, *B*2 → *E*2, ···, *Bn* → *En*), all express a choice between two or more possibilities. And when we get rid of the go to statement, we must find other means of describing the conditional transfer of control. Fortunately, this is not difficult. When we get rid of the go to statement, we must find other means of describing the conditional transfer of control. Fortunately, this is not difficult.

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can and must avoid that, but prefers after severe self-discipline. The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can and must avoid that, but prefers after severe self-discipline.

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can and must avoid that, but prefers after severe self-discipline.

judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and thereby helps to avoid the need for introducing a large number of labels."

In [2] Böhm and Jacopini seems to have proved the (logical) superfluity of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one is quoted to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

References:

1. Wirth, Niklaus, and Hoare, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. Böhm, Corrado, and Jacopini, Guiseppe. Flow diagrams, Turing machines and languages with only two formation



Go To Statement Considered Harmful*

by Edsger W Dijkstra

*Title by Niklaus Wirth

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to perform satisfactorily and it is the program that has to [illegible]. Yet, in texts about programming, the process is hardly ever given explicit attention.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process is controlled by a tape with instructions, what determines the moment at which the next instruction is fetched from the tape?) Let us now also restrict the number of different kinds of actions that may occur in a process to a few very primitive ones, viz. the assignment statement (for the purpose of this discussion regarded as the description of elementary actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A*1 **else** *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of $(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1, B2 \rightarrow E2, \cdots, Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous because we can describe repetitions with the help of recursive procedures. For reasons of implementation we may wish to express repetitions explicitly and for conceptual reasons of publication we may wish to allow for an explicit repetition clause. We can do so keeping all the advantages of the simple textual index if we augment it by a set of counters, one for each repetition clause, to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition clause execution. Finally, since procedure calls) may be applied recursively, we find that now the dynamic process can be uniquely characterized by the textual index and the associated set of dynamic indices.

The [illegible] of these indices are outside the program's control (they are generated (either by the writer of this program or by the implementation) to express how the program wishes to be executed). They form an independent coordinate system to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, n's value [illegible] the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process conveniently. Usually people try to find explanations in terms of values of the progressing variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

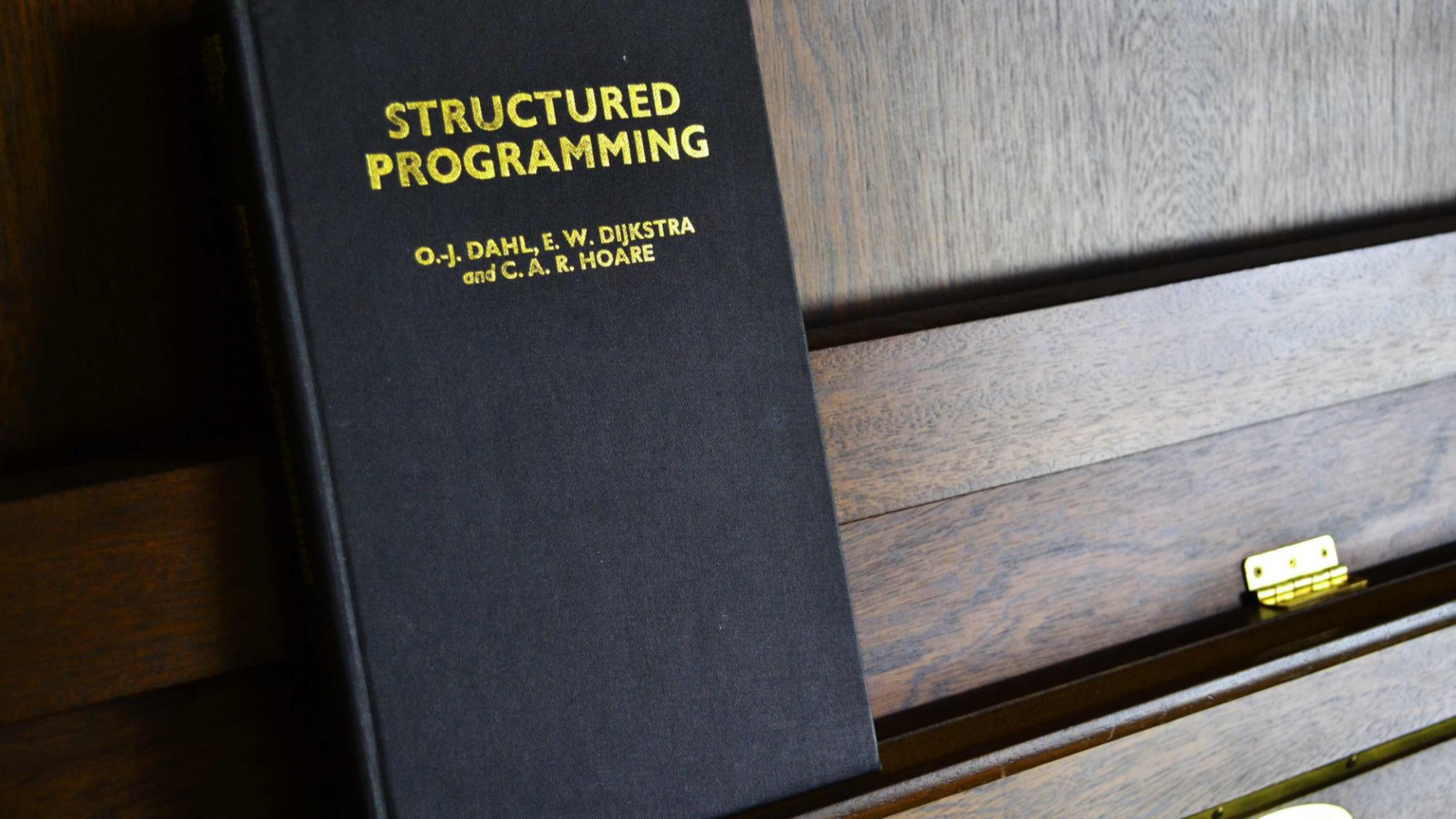
It is hard to end this with a fair acknowledgment. [illegible] judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

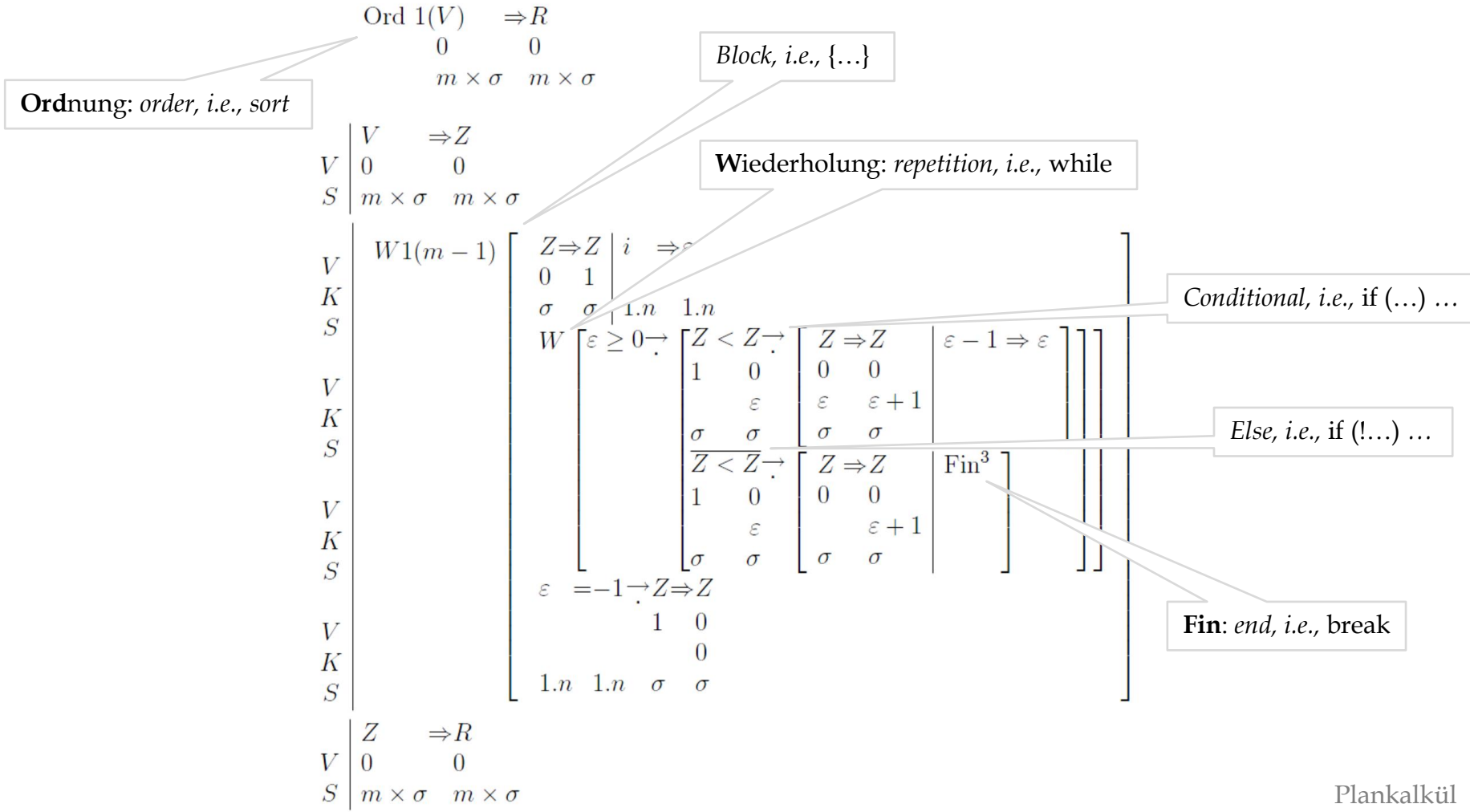
The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and [illegible] the need for introducing a large number [illegible]

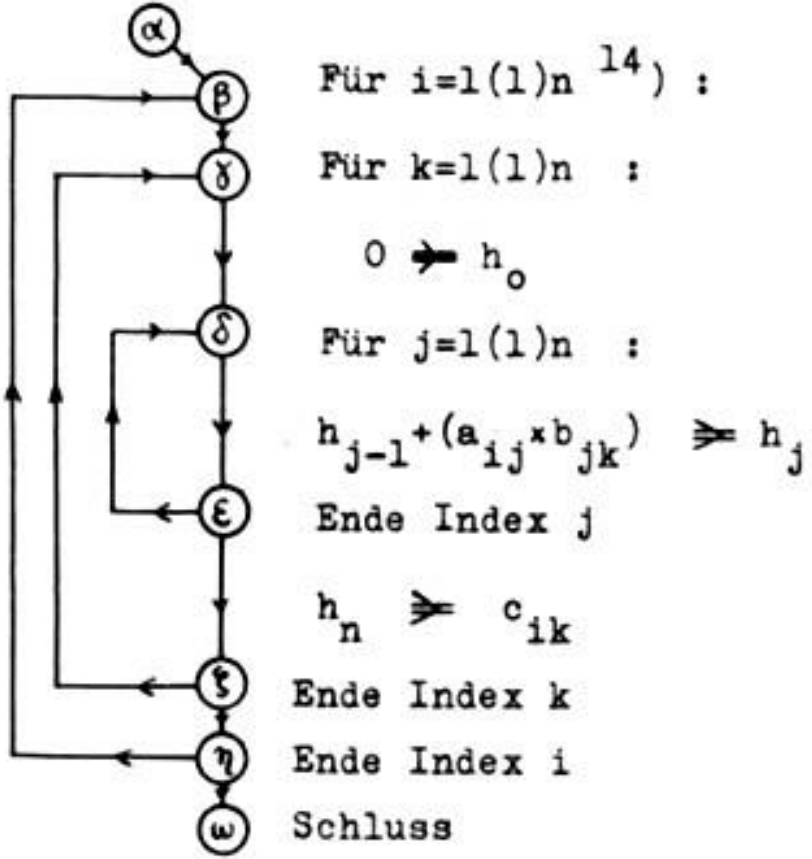
The [illegible] seems to have proved the (logical) superfluity of the **go to** statement. The exercise to translate [illegible] more or less mechanically into a jump-[illegible] be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

References:

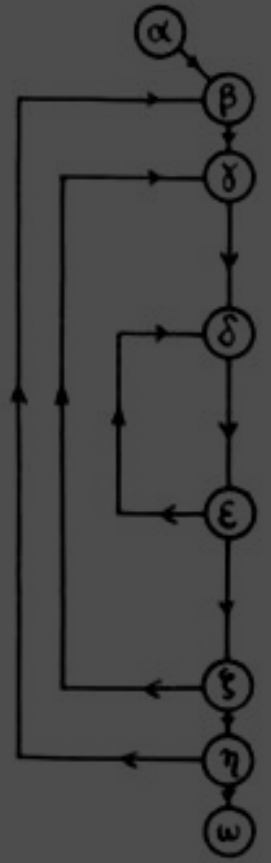
1. Wirth, Niklaus, and Hoare, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. Böhm, Corrado, and Jacopini, Guiseppe. Flow diagrams, Turing machines and languages with [illegible] formation [illegible]


STRUCTURED
PROGRAMMING
O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE


Ord 1(V) ⇒R
0 0
m × σ m × σ
Ordnung: order, i.e., sort
Block, i.e., {…}
Wiederholung: repetition, i.e., while
V ⇒Z
V 0 0
S m × σ m × σ
W1(m − 1)
Z⇒Z i ⇒ε
0 1
σ σ 1.n 1.n
W
ε ≥ 0→ Z < Z→ Z ⇒Z ε − 1 ⇒ ε
1 0 0 0
ε ε ε + 1
σ σ σ σ
Z < Z→ Z ⇒Z Fin3
1 0 0 0
ε ε + 1
σ σ σ σ
ε =−1→Z⇒Z
1 0
0
1.n 1.n σ σ
Conditional, i.e., if (…) …
Else, i.e., if (!…) …
Fin: end, i.e., break
Z ⇒R
0 0
m × σ m × σ
Plankalkül



craftofcoding.wordpress.com/2021/03/05/superplan-groovy



für

Für l(1)n 14 :

Für k=1(1)n :

$0 \geqslant h_o$

Für j=1(1)n :

$h_{j-1} + (a_{ij} \times b_{jk}) \geqslant h_j$

Ende Index j

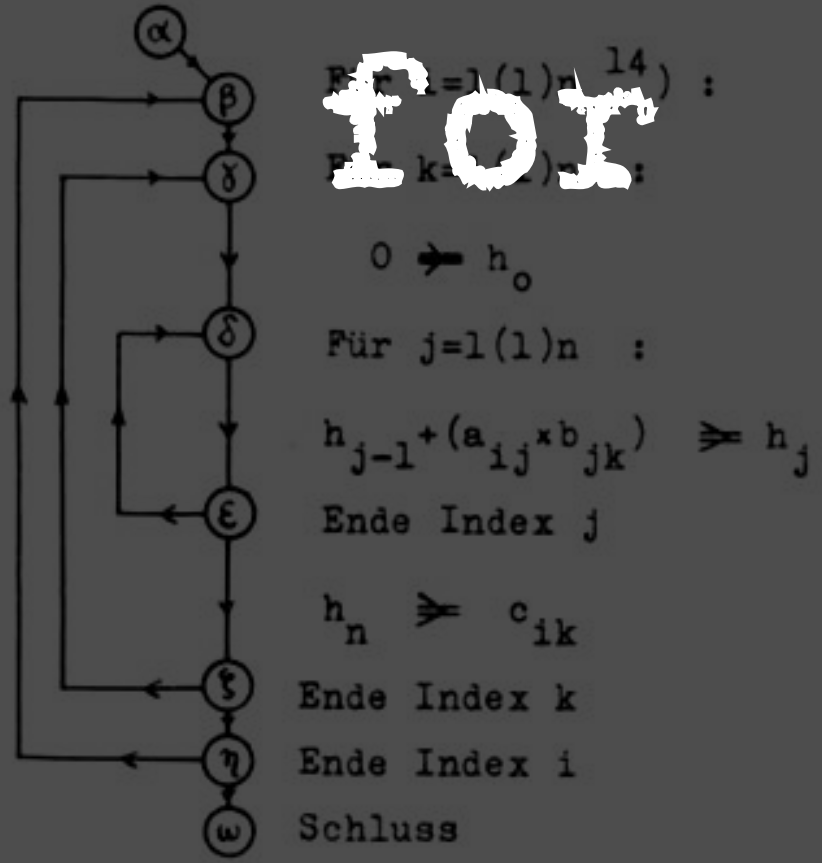
$h_n \geqslant c_{ik}$

Ende Index k

Ende Index i

Schluss

craftofcoding.wordpress.com/2021/03/05/superplan-groovy



craftofcoding.wordpress.com/2021/03/05/superplan-groovy

I'm here to talk about what seems like an absurd idea: that if-then-else had to be invented.

github.com/e-n-f/if-then-else/blob/master/if-then-else.md

PROPOSAL FOR A UNIVERSAL LANGUAGE FOR THE DESCRIPTION OF COMPUTING PROCESSES

by

F. L. Bauer
H. Bottenbruch
H. Rutishauser
K. Samelson

INTRODUCTION

In the following, the first stage of an algorithmic language representing the basis of the formula translation project Zurich-Munchen-Mainz-Darmstadt is developed. The language is intended to permit complete description of any computing process in a compact and easily legible and controllable form, with the important restriction that for the time being only calculations with real numbers are permitted. Hypercomplex quantities of all kinds (such as complex and double precision numbers, vectors, matrices, etc.) are not included at the moment; calculations with such quantities must be written down componentwise.

The symbols necessary for representation of the elements of the language are chosen as far as possible in conformity with usages of printing. Furthermore the language has been adapted as far as possible

a) The Definition of a case has the general form:
Designation of the case (letter λ, followed by one or more figures and closing bracket)
and a criterion (same as in IF-formulae, see Section 12)
terminated by semicolon.

example:
a1) $x < 0$
a2) $x = 0$ (20)
a3) ELSE

Here three cases are defined: case a1) corresponds to $x < 0$, case a2) to $x = 0$ and case a3) to the complement of the sum of the two others, i.e., $x > 0$.

What is done with case a3) is called a remainder - case definition and is preferably used to avoid that in complicated situations for some values of the variables no case at all is defined.

PRELIMINARY REPORT—INTERNATIONAL ALGEBRAIC LANGUAGE

Note:

In the interest of immediate circulation of the results of the ACM-GAMM committee work on an algebraic programming language, this preliminary report is presented. The language described naturally enough represents a compromise, but one based more upon differences of taste than on content or fundamental ideas. Even so, it provides a natural and simple medium for the expression of a large class of algorithms. This report has not been thoroughly examined for errors and inconsistencies. It is anticipated that the committee will prepare a more complete description of the language for publication.

For all scientific purposes, reproduction of this report is explicitly permitted without any charge.

Acknowledgements:

The members of the conference wish to express their appreciation to the Association for Computing Machinery, the "Deutsche Forschungsgemeinshaft" and the Swiss Federal Institute of Technology for substantial help in making this conference and resultant report possible.

A. J. PERLIS
K. SAMELSON
for the ACM-GAMM Committee

PART I—INTRODUCTION

In 1955, as a result of the Darmstadt meeting on electronic computers, the GAMM (association for applied mathematics and mechanics), Germany, set up a committee on programming (Programmierungsausschus). Later a subcommittee began to work on formula translation and on the construction of a translator, and a considerable amount of work was done in this direction.

B. *BASIC SYMBOLS*

The reference language is built up from the basic symbols listed in Part IIIa. These are:

1. *Letters* λ (the standard alphabet of small and capital letters)
2. *Figures* ζ (arabic numerals 0, ... 9)
3. *Delimiters* δ consisting of
   a. operators ω:

arithmetic	$+ \quad - \quad \times \quad /$
relational	$< \quad \leq \quad = \quad \geq \quad > \quad \neq$
logical	$\neg \quad \vee \quad \wedge \quad \equiv$
sequential	*go to* *do* *return* *stop* *for* *if* *or* *if either* *or if*

   b. separators σ: . , : ; := =: $\rightarrow$ $_{10}$ *begin* *end*
   c. brackets β: ( ) [ ] $\uparrow$ $\downarrow$
   d. declarators ϕ: *procedure* *array* *switch* *type* *comment*

B. *BASIC SYMBOLS*

The reference language is built up from the basic symbols listed in Part IIIa. These are:

1. *Letters* λ (the standard alphabet of small and capital letters)
2. *Figures* ζ (arabic numerals 0, ... 9)
3. *Delimiters* δ consisting of
   a. operators ω:

arithmetic	$+ \quad - \quad \times \quad /$
relational	$< \quad \leq \quad = \quad \geq \quad > \quad \neq$
logical	$\neg \quad \vee \quad \wedge \quad \equiv$
sequential	*go to* *do* *return* *stop* *for* *if* *or* *if either* *or if*

   b. separators σ: . , : ; := =: $\rightarrow$ $_{10}$ *begin* *end*

   c. brackets β: ( ) [ ] $\uparrow$ $\downarrow$

   d. declarators ϕ: *procedure* *array* *switch* *type* *comment*

Reprinted from the COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY
Vol. 3, No. 5, May 1960
Made in U.S.A.
With typographical corrections as of June 28, 1960

Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS, F. L. BAUER, J. GREEN, C. KATZ, J. MCCARTHY, A. J. PERLIS, H. RUTISHAUSER, K. SAMELSON, B. VAUQUOIS, J. H. WEGSTEIN, A. VAN WIJNGAARDEN, M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language ALGOL,[1, 2] as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL *Bulletin*, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the ACM *Communications* where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM *Communications*. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

Reprinted from the COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY
Vol. 3, No. 5, May 1960
Made in U.S.A.
With typographical corrections as of June 28, 1960

Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS, F. L. BAUER, J. GREEN, C. KATZ, J. MCCARTHY, A. J. PERLIS, H. RUTISHAUSER, K. SAMELSON, B. VAUQUOIS, J. H. WEGSTEIN, A. VAN WIJNGAARDEN, M. WOODGER

Dedicated to the memory of WILLIAM TURANSKI

else

ROD

Background

After the publication of a preliminary report on the algorithmic language ALGOL,[1, 2] as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL *Bulletin*, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal

M while, ed States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the ACM *Communications* where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM *Communications*. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

Functions

Procedures

Subroutines

Subroutines are special cases of more general program components, called *coroutine*s. In contrast to the unsymmetric relationship between a main routine and a subroutine, there is complete symmetry between coroutines, which *call on each other*.

Donald Knuth

The Art of Computer Programming, Volume 1

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to

Coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.

The coroutine notion can greatly simplify the conception of a program when its modules do not communicate with each other synchronously.

stract specifications.

owledgments. We owe a great deal to our col-
t Carnegie-Mellon University and the Univer-
outhern California Information Sciences Insti-
ecially Mario Barbacci, Neil Goldman, Donald
ohn Guttag, Paul Hilfinger, David Jefferson,
nes, David Lamb, David Musser, Karla Per-
mesh Ramakrishna, and David Wile. We
so like to thank James Horning and Barbara
nd their groups at the University of Toronto
T., respectively, for their critical reviews of
We also appreciate very much the perceptive
s that a number of our colleagues have made
lier draft of this paper. Finally, we are grateful
ond Bates, David Lamb, Brian Reid, and
onke for their expert assistance with the docu-
matting programs.

O.-J., and Hoare, C.A.R. Hierarchical program struc-
uctured Programming, O.-J. Dahl, E.W. Dijkstra, and
are, Academic Press, New York, 1972, pp. 175–220.
C.A.R. A note on the for statement. *BIT 12* (1972),

C.A.R. Proof of correctness of data representations.
natica 1, 4 (1972), 271–281.

Language Design for Reliable Software

S.L. Graham Editor

Abstraction Mechanisms in CLU

Barbara Liskov, Alan Snyder,
Russell Atkinson, and Craig Schaffert
Massachusetts Institute of Technology

CLU is a new programming language designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions – procedural, control, and especially data abstractions – are useful in the programming process. Of these, only the procedural abstraction is supported well by

Iterators are a form of coroutine; however, their use is sufficiently constrained that they are implemented using just the program stack.

Language Design for Reliable Software
S.L. Graham Editor

Abstraction Mechanisms in CLU

Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert
Massachusetts Institute of Technology

CLU is a new programming language designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions — procedural, control, and especially data abstractions — are useful in the programming process. Of these, only the procedural abstraction is supported well by

stract specifications.

owledgments. We owe a great deal to our col-
t Carnegie-Mellon University and the Univer-
outhern California Information Sciences Insti-
ecially Mario Barbacci, Neil Goldman, Donald
ohn Gu
nes, David Lamb, David Musser, Karla Per-
mesh Ramakrishna, and David Wile. We
so like
nd their groups at the University of Toronto
T., respectively, for their critical reviews of
We als
s that a number of our colleagues have made
lier dra
ond
onke for their expert assistance with the docu-
matting

D.-J., and Hoare, C.A.R. Hierarchical program struc-
uctured Programming, O.-J. Dahl, E.W. Dijkstra, and
are, Academic Press, New York, 1972, pp. 175–220.
C.A.R. A note on the for statement. *BIT 12* (1972),

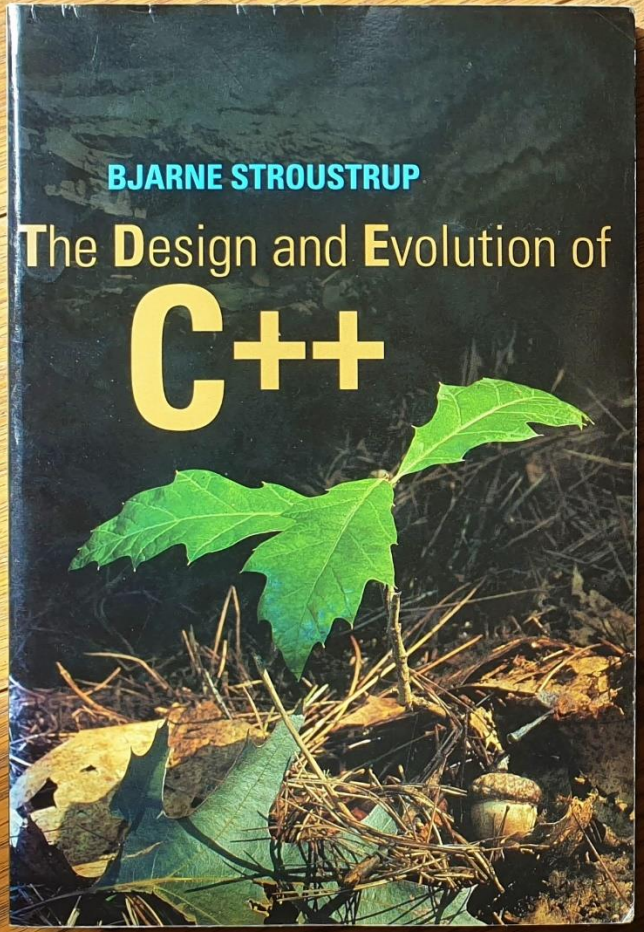
C.A.R. Proof of correctness of data representations.
natica 1, 4 (1972), 271–281.

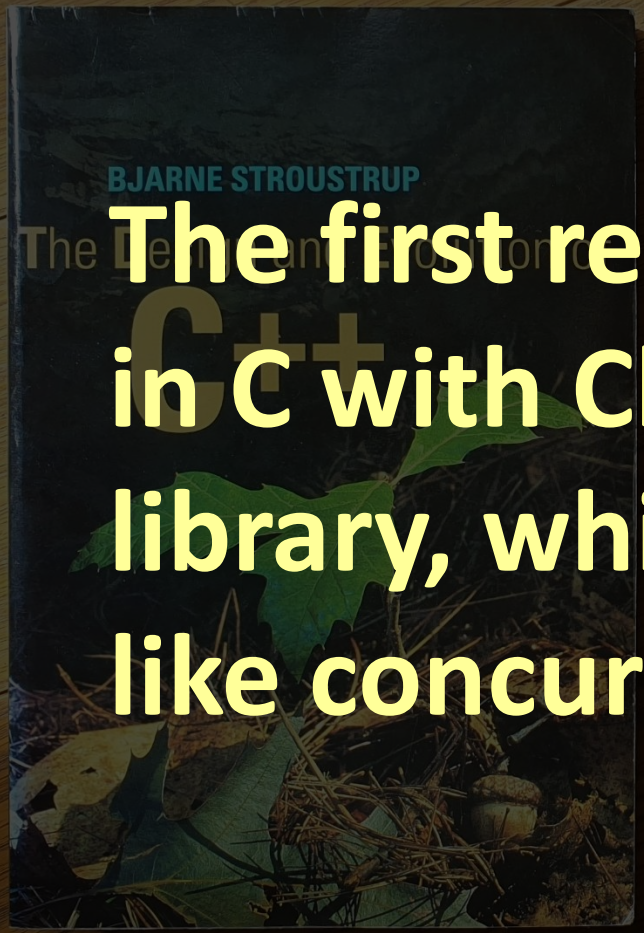
```
for n: int in fibonacci() do
    print(n)
end
```

```
fibonacci = iter () yields (int)
    Fn_2: int := 0
    yield (Fn_2)
    Fn_1: int := 1
    yield (Fn_1)
    while true do
        Fn: int := Fn_1 + Fn_2
        yield (Fn)
        Fn_2, Fn_1 := Fn_1, Fn
    end
end fibonacci
```

```
def fibonacci():
    Fn_2 = 0
    yield Fn_2
    Fn_1 = 1
    yield Fn_1
    while True:
        Fn = Fn_1 + Fn_2
        yield Fn
        Fn_2, Fn_1 = Fn_1, Fn
```

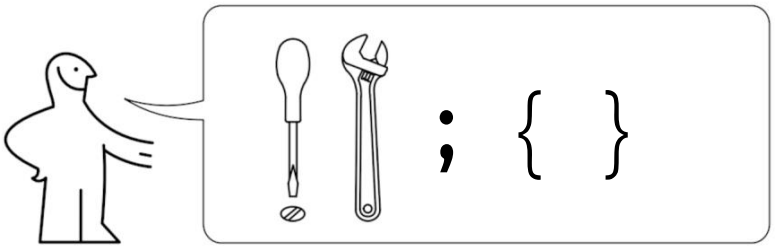
```
IEnumerable<int> fibonacci() {
    int Fn_2 = 0;
    yield return Fn_2;
    int Fn_1 = 1;
    yield return Fn_1;
    while (true) {
        int Fn = Fn_1 + Fn_2;
        yield return Fn;
        (Fn_2, Fn_1) = (Fn_1, Fn);
    }
}
```

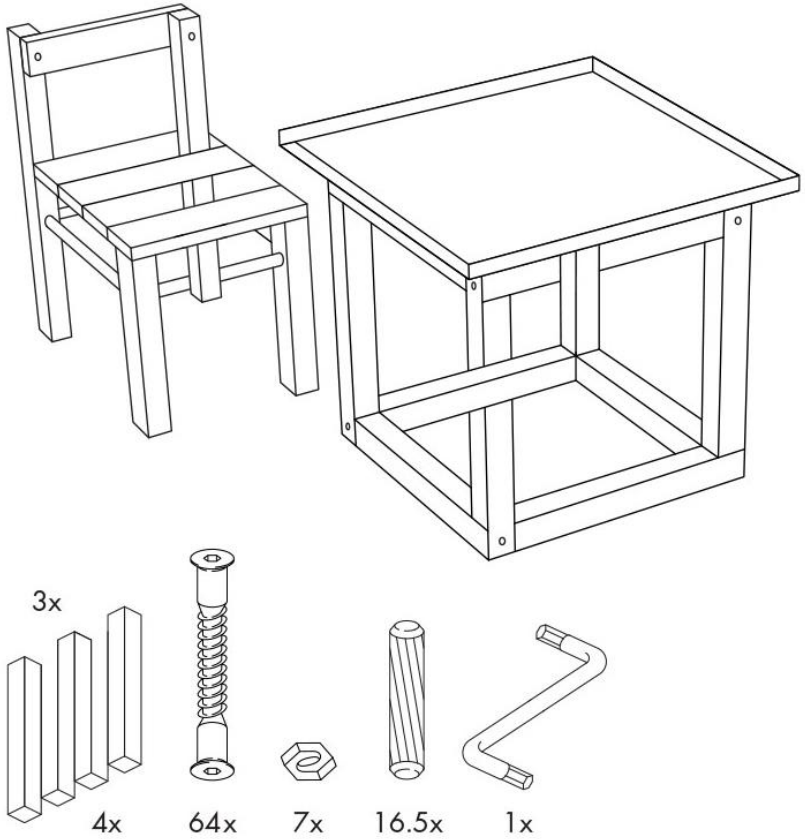

BJARNE STROUSTRUP
The Design and Evolution of
C++



The first real code to be written in C with Classes was the task library, which provides Simula-like concurrency for simulation.

CÖRUTIN



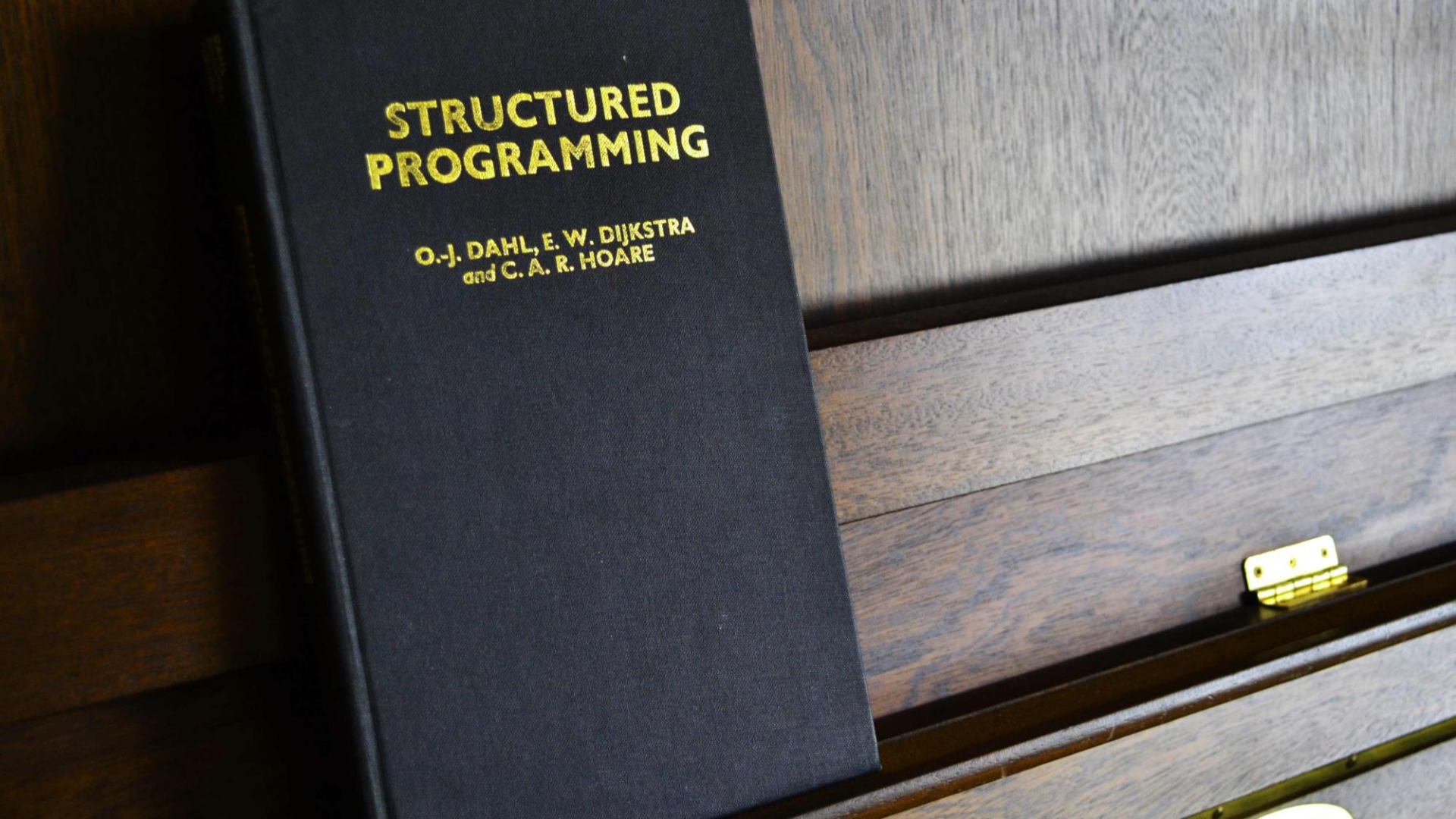


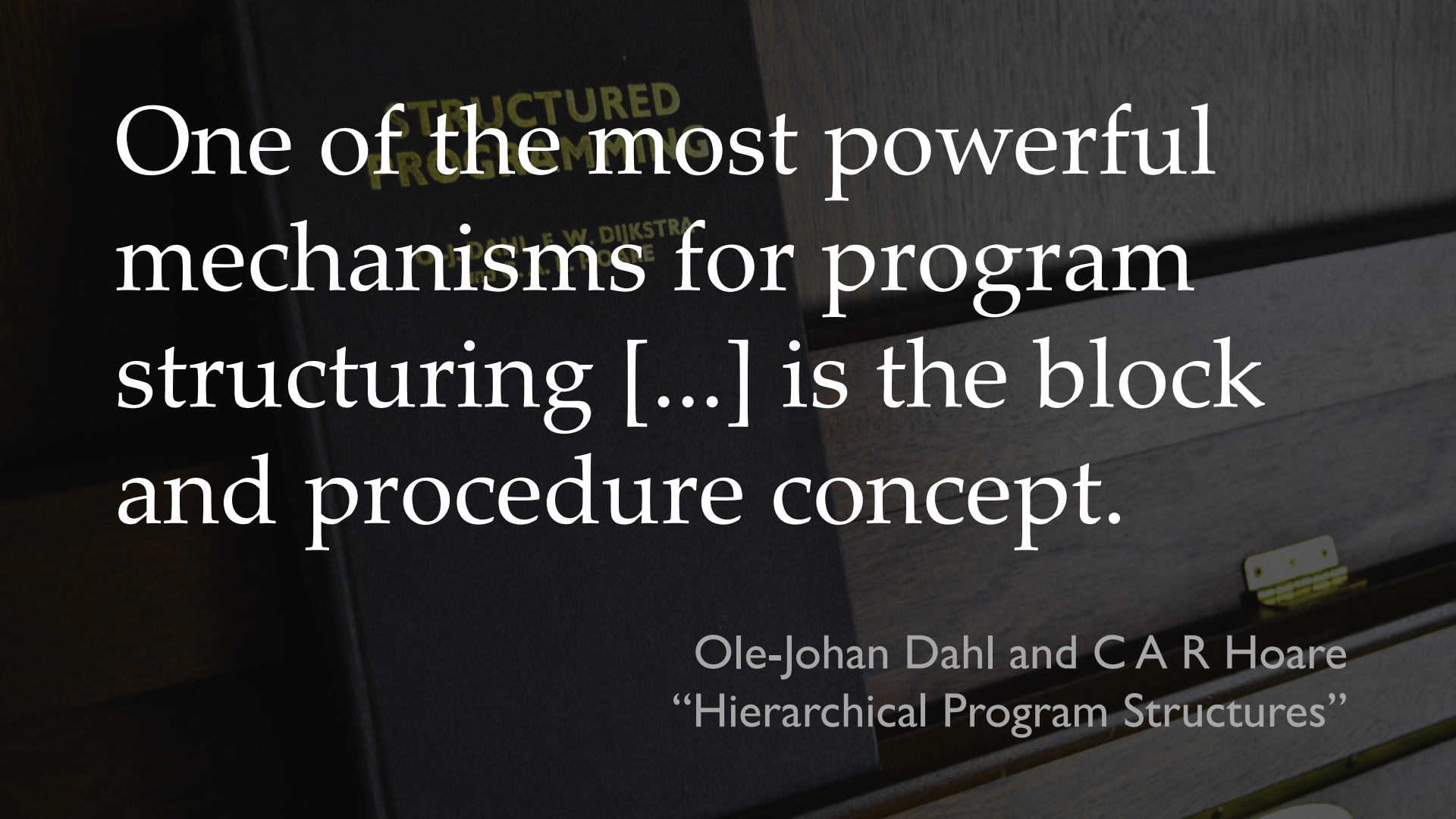
In Simula I [...] a process was a bundle of data attributes associated with a single action known as the process' *operating rule*.

Each process' operating rule could be suspended or resumed at any time—the operating rules were a kind of coroutine.

OOP Before OOP with Simula

twobithistory.org/2019/01/31/simula.html

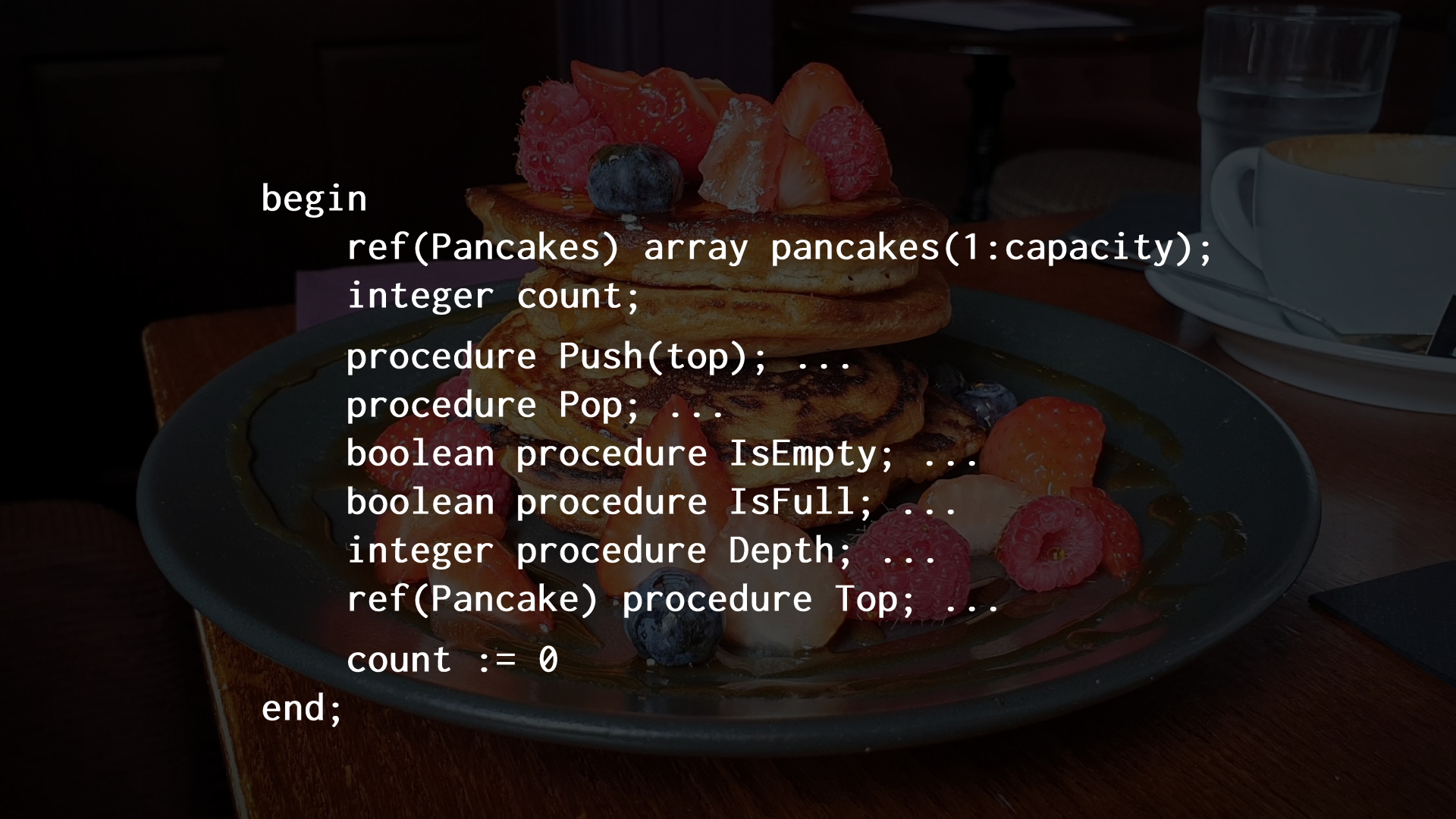

STRUCTURED
PROGRAMMING
O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE



One of the most powerful mechanisms for program structuring [...] is the block and procedure concept.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"





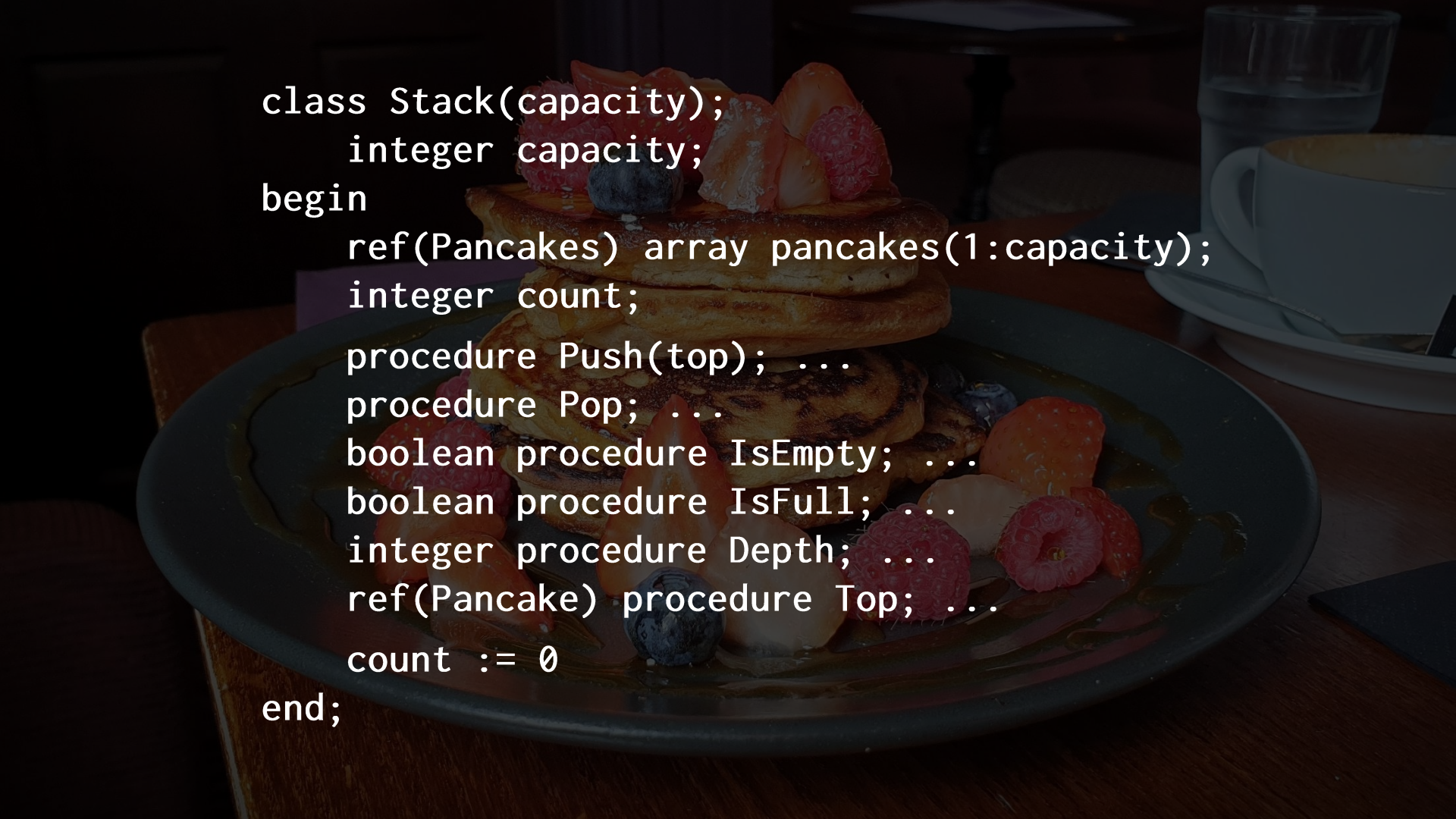
```
begin
    ref(Pancakes) array pancakes(1:capacity);
    integer count;

    procedure Push(top); ...
    procedure Pop; ...
    boolean procedure IsEmpty; ...
    boolean procedure IsFull; ...
    integer procedure Depth; ...
    ref(Pancake) procedure Top; ...

    count := 0
end;
```

A procedure which is capable of giving rise to block instances which survive its call will be known as a class; and the instances will be known as objects of that class.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"



```
class Stack(capacity);
    integer capacity;
begin
    ref(Pancakes) array pancakes(1:capacity);
    integer count;

    procedure Push(top); ...
    procedure Pop; ...
    boolean procedure IsEmpty; ...
    boolean procedure IsFull; ...
    integer procedure Depth; ...
    ref(Pancake) procedure Top; ...

    count := 0
end;
```

PROGRAMMING WITH ABSTRACT DATA TYPES

Barbara Liskov
Massachusetts Institute of Technology
Project MAC
Cambridge, Massachusetts

Stephen Zilles
Cambridge Systems Group
IBM Systems Development Division
Cambridge, Massachusetts

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

PROGRAMMING WITH ABSTRACT DATA TYPES

Barbara Liskov
Massachusetts Institute of Technology
[illegible]

Stephen Zilles
[illegible]
Cambridge, Massachusetts

An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects.

Abstract

The motivation behind the work in very high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

PROGRAMMING WITH ABSTRACT DATA TYPES

Barbara Liskov
Massachusetts Institute of Technology
Project MAC
Cambridge, Massachusetts

Stephen Zilles
IBM Systems Development Division
Cambridge, Massachusetts

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program is more reliable as a consequence. Thus it is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

A programmer [...] is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation.

could easily have been abstract types instead. In that case, the built-in predicate alphanumeric, for example, would have been expressed as char$alphanumeric. Only the syntax would change; the meaning and use of the types would be the same in either case.

To sum up, Polish_gen makes use of five data abstractions, infile, outfile, grammar, token and stack, plus one functional abstraction, scan. The power of the data abstractions is illustrated by the types infile and outfile, which are used to shield Polish_gen from any physical facts concerning its input and output, respectively. Polish_gen does not know what input and output devices are being used, when the I/O actually takes place, nor does it know how characters are represented on the devices. What it does know is just enough for its needs: For parameter output it knows how to add a string of characters (outfile$out_str) and how to signify that the output is complete (outfile$close). For parameter input, it knows how to obtain the next character (infile$get), how to look at the next character without removing it from input (infile$peek), and how to recognize the end of input (infile$eof). (Note that for scan to operate correctly, infile must provide a non-blank, non-alphanumeric character on any call on infile$get or infile$peek after the end of file has been reached.) In every case its knowledge consists of the names of the operations which provide these services.

Defining Abstract Data Types

In this section, we describe the programming object -- the operation cluster -- whose translation provides an implementation of a type. The cluster

```
stack: cluster(element_type: type)
         is push, pop, top, erasetop, empty;

   rep(type_param: type) = (tp: integer;
                            e_type: type;
                            stk: array[1..]
                                   of type_param;

   create

      s: rep(element_type);

      s.tp := 0;
      s.e_type := element_type;
      return s;
      end

   push: operation(s: rep, v: s.e_type);

      s.tp := s.tp+1;
      s.stk[s.tp] := v;
      return;
      end

   pop: operation(s: rep) returns s.e_type;

      if s.tp = 0 then error;
      s.tp := s.tp-1;
      return s.stk[s.tp+1];
      end

   top: operation(s: rep) returns s.e_type;

      if s.tp = 0 then error;
      return s.stk[s.tp];
      end
```

```
stack: cluster(element_type: type) is push, pop, top, ...;
rep(type_param: type) = (tp: integer; e_type: type; stk: array[1..] of type_param);
    create
        s: rep(element_type);
        s.tp := 0;
        s.e_type := element_type;
        return s;
        end
    push: operation(s: rep, v: s.e_type);
        s.tp := tp + 1;
        s.stk[s.tp] := v;
        return;
        end
    pop: operation(s: rep) returns s.e_type;
        if s.tp = 0 then error;
        s.tp := s.tp - 1;
        return s.stk[s.tp + 1];
        end
    top: operation(s: rep) returns s.e_type;
        if s.tp = 0 then error;
        return s.stk[s.tp];
        end
    ...
end stack
```

```
stack: cluster(element_type: type)
    is push, pop, top, erasetop, empty;
    e_type: type;
    stk: array[1..]
        of type_param;

create

    s: rep(element_type);

    s.tp := 0;
    s.e_type := element_type;
    return s;
    end

push: operation(s: rep, v: s.e_type);

    s.tp := s.tp+1;
    s.stk[s.tp] := v;
    return;
    end

pop: operation(s: rep) returns s.e_type;

    if s.tp = 0 then error;
    s.tp := s.tp-1;
    return s.stk[s.tp+1];
    end

top: operation(s: rep) returns s.e_type;

    if s.tp = 0 then error;
    return s.stk[s.tp];
    end
```

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks

A data structure, its internal linkings, accessing procedures and modifying procedures are part of a single module.

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism [illegible] system [illegible] time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into [illegible] are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption

Introduction

A [illegible] of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, p. 10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays

CR Categories: 4.20, 4.22, 4.32

grams, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself (as in the multiple function units of the CDC 6600) or by the software (as in an I/O control package, or a multiprogrammed operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in Algol 68 [18], PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs and it may lead to expense (e.g. crossbar switches) and

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

processes

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays

CR Categories: 4.20, 4.22, 4.32

grams, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, [illegible] attempt has been made to disguise this fact from the programmer by [illegible] its use in the multiple function units of the CDC 6600 or the software (as in OS/360 control package) of a multiprogramming operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in Algol 68 [18], PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs and it may lead to expense (e.g. crossbar switches) and

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen[illegible]niversity
Belfast, Nor[illegible]rn Ireland

channels

This paper [illegible]gests[illegible]t input[illegible]put a[illegible]asic [illegible]itives of p[illegible]ramm[illegible]and[illegible]t pa[illegible]el co[illegible]mmu[illegible]ing[illegible]is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

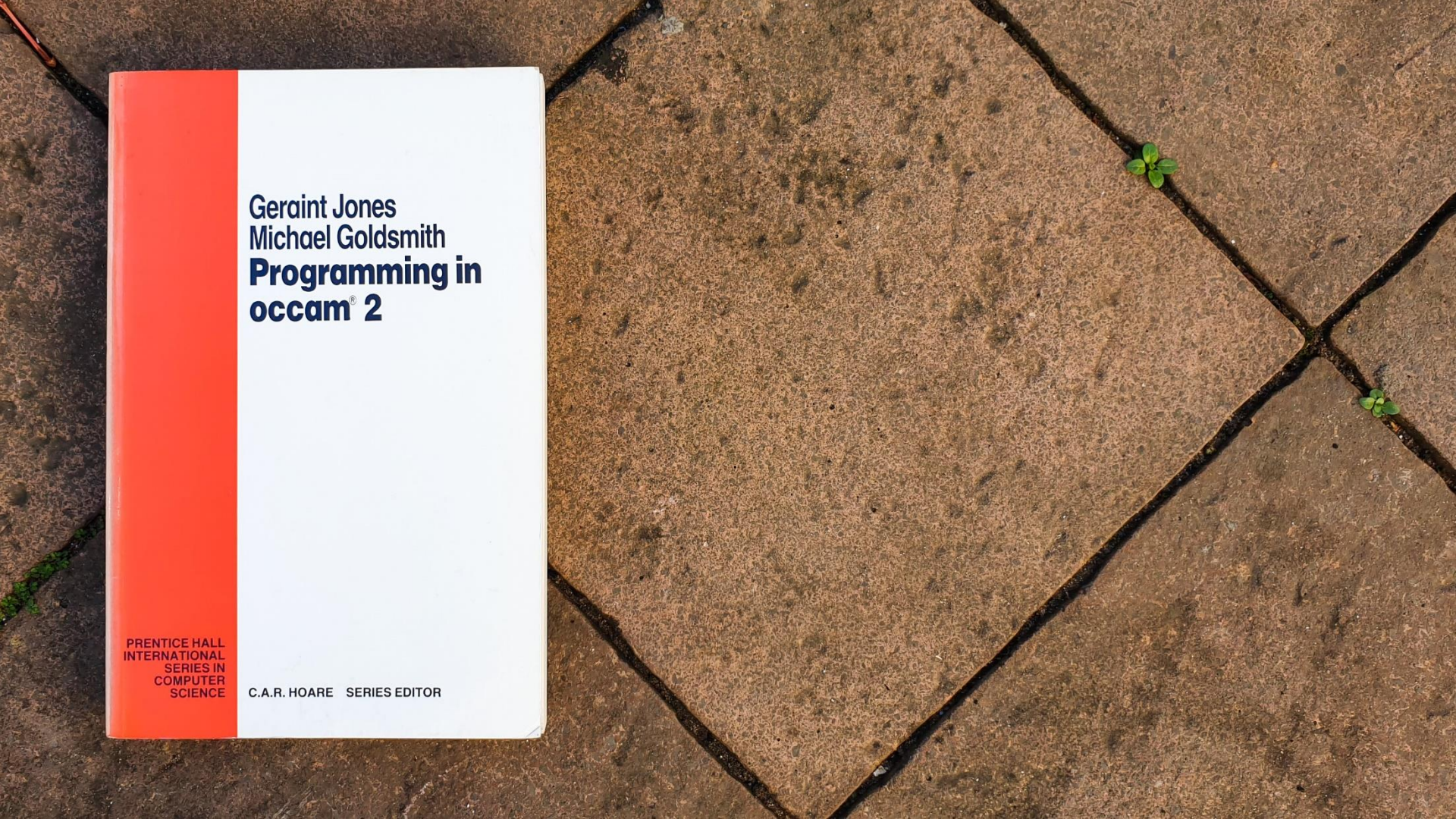
Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays

CR Categories: 4.20, 4.22, 4.32

grams, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digit[illegible] computer has been designed primarily for determin[illegible] execution of a single sequential program. Where the[illegible]sire for greater sp[illegible]s[illegible] the int[illegible]on of [illegible]rallelis[illegible] atten[illegible]has[illegible]ma[illegible]to d[illegible]ise[illegible]fac[illegible]om the prog[illegible]mer,[illegible]er b[illegible]itself[illegible]in th[illegible] funct[illegible]units[illegible]the[illegible]C 6600) or b[illegible]he so[illegible]tware[illegible] in a[illegible]O co[illegible]pac[illegible]mul[illegible]gra[illegible]ned[illegible] erating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in Algol 68 [18], PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs and it may lead to expense (e.g. crossbar switches) and


Geraint Jones
Michael Goldsmith
Programming in occam® 2
PRENTICE HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE
C.A.R. HOARE SERIES EDITOR

```
PROC fibonacci(CHAN OF INT out)
  INT Fn_2, Fn_1 :
  SEQ
    Fn_2 := 0
    out ! Fn_2
    Fn_1 := 1
    out ! Fn_1
    WHILE TRUE
      INT Fn :
      SEQ
        Fn := Fn_1 + Fn_2
        out ! Fn;
        Fn_2, Fn_1 := Fn_1, Fn
```

```
func Fibonacci(out chan<- int) {
    Fn_2 := 0
    out <- Fn_2
    Fn_1 := 1
    out <- Fn_1
    for {
        Fn := Fn_1 + Fn_2
        out <- Fn
        Fn_2, Fn_1 = Fn_1, Fn
    }
}
```

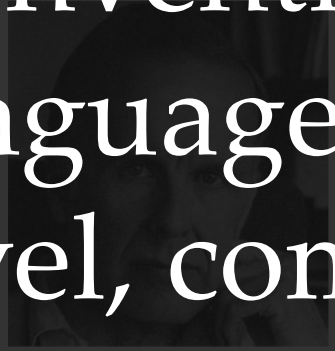
We still need a revolution in software.

Sophie Wilson

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

Conventional programming languages are basically high level, complex versions of the von Neumann computer.



General permission to make [illegible] use in teaching or research of all part[illegible] granted to individuals and non-profit libraries [illegible] provided that [illegible] copyright notice is given and that reference is made to the publication, its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

© 1978 ACM 0001-0782/78/0800-0613 $00.75

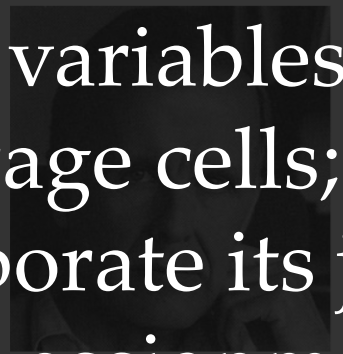
[illegible] ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming [illegible] their common bottleneck [illegible] their [illegible] copying of [illegible] state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for [illegible] programs from [illegible] and their lack of useful [illegible] programs. [illegible] programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs [illegible] are [illegible] archically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic.



or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

© 1978 ACM 0001-0782/78/0800-0613 $00.75

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects ... weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von ... of semantics to ... programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for ... of useful mathematical properties for reasoning about programs.

... programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed ... their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Bash
C
C#
C++
COBOL
Dart
Delphi
Fortran
Go
Java
JavaScript
Kotlin
Lua
MATLAB
Mathematica
Objective-C
PHP
PowerShell
Python
R
Ruby
Rust
SAS
SQL
Scala
Scratch
Swift
TypeScript
Visual Basic

Bash
C
C#
C++
COBOL
Dart
Delphi
Fortran
Go
Java
JavaScript
Kotlin
Lua
MATLAB
Mathematica
Objective-C
PHP
PowerShell
Python
R
Ruby
Rust
SAS
SQL
Scala
Scratch
Swift
TypeScript
Visual Basic

Bash
C
C#
C++
COBOL
Dart
Delphi
Fortran
Go
Java
JavaScript
Kotlin
Lua
MATLAB
Mathematica
Objective-C
PHP
PowerShell
Python
R
Ruby
Rust
SAS
SQL
Scala
Scratch
Swift
TypeScript
Visual Basic

Bash
C
C#
C++
COBOL
Dart
Delphi
Fortran
Go
Java
JavaScript
Kotlin
Lua
MATLAB
Mathematica
Objective-C
PHP
PowerShell
Python
R
Ruby
Rust
SAS
SQL
Scala
Scratch
Swift
TypeScript
Visual Basic

```
SELECT number
FROM numbers
WHERE number % 2 = 0
```

```
[number
 for number in numbers
 if number % 2 == 0]
```

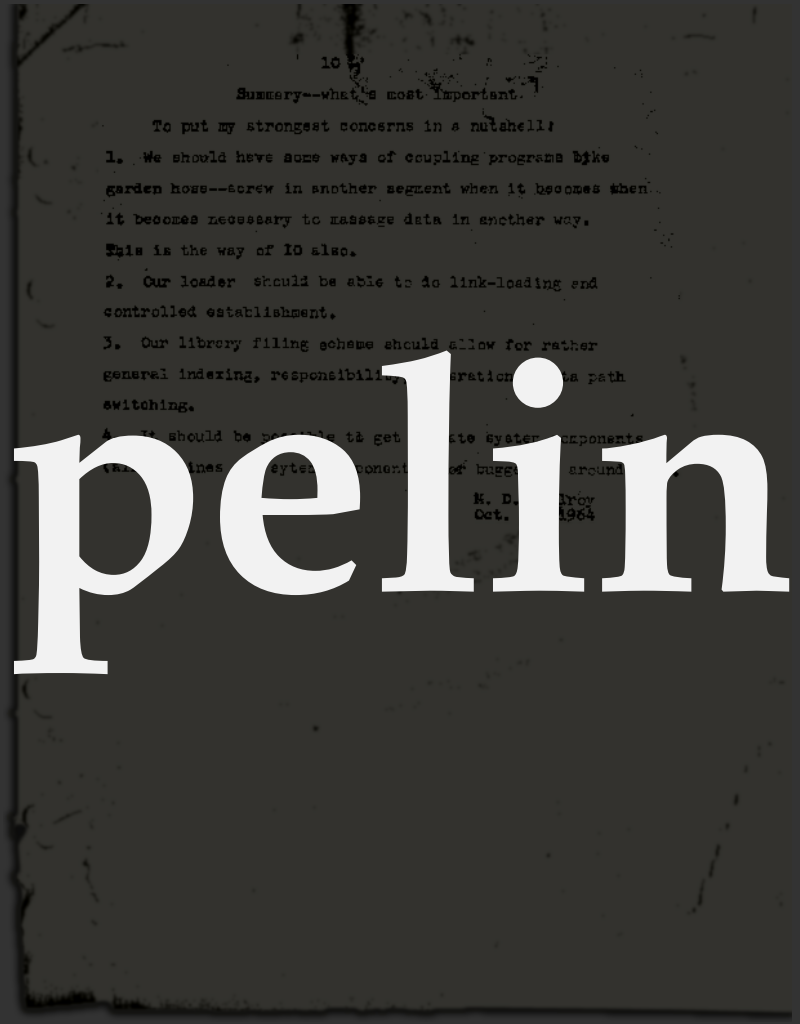
```
[number|
 number <- numbers
 number `mod` 2 == 0]
```

```
{number|
 number ∈ numbers,
 number mod 2 = 0}
```

```
from number in numbers
where number % 2 == 0
select number
```

```
numbers
.Where(number => number % 2 == 0)
```

```
numbers.stream()
.filter(number -> number % 2 == 0)
```


10

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, [illegible] path switching.

[illegible] should be possible to get [illegible] system components [illegible] bugs around.

M. D. [illegible]roy
Oct. 1964

pipelines

Summary--what's most important.

To put my strongest concerns in a nutshell:

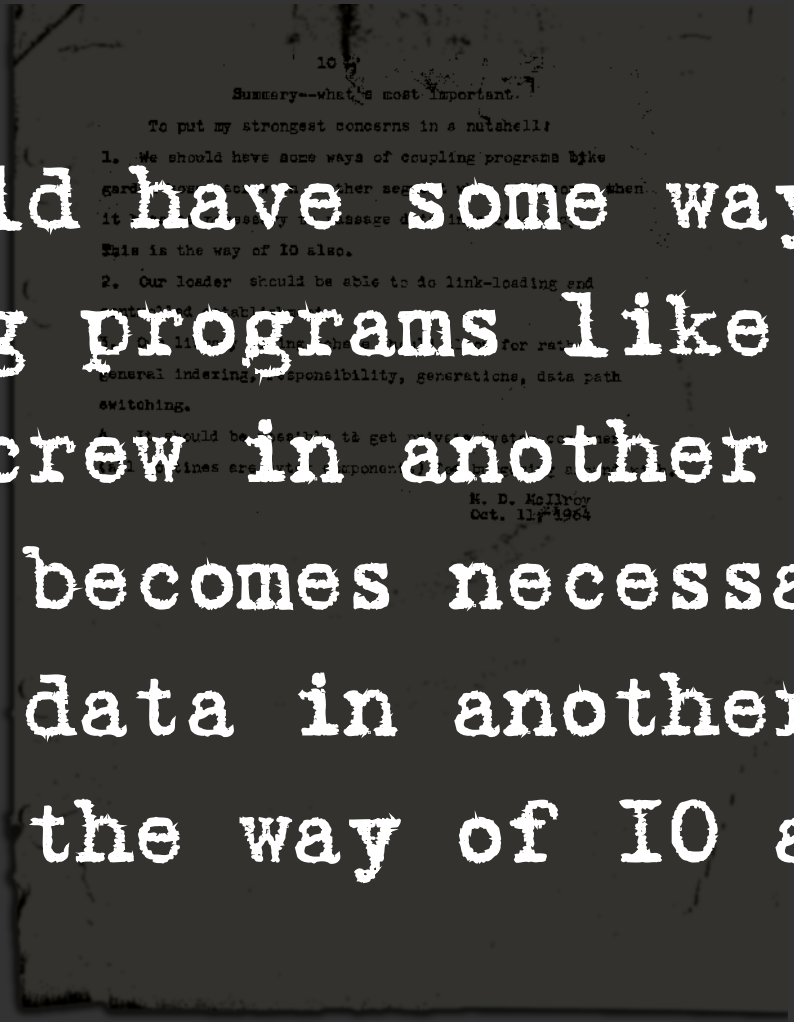
1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are system components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964


10

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like

This is the way of IO also.

2. Our loader should be able to do link-loading and

for ra

general indexing, responsibility, generations, data path switching.

M. D. McIlroy
Oct. 11, 1964

We should have some ways of coupling programs like garden hose--screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also.

Excel is the world's most popular functional language

Simon Peyton-Jones

Don't put lambdas in there — it's not going to help

Kevlin Henney

March 2020

In December 2020, we announced LAMBDA, which allows users to define new functions written in Excel's own formula language.

microsoft.com/en-us/research/blog/lambda-the-ultimatae-excel-worksheet-function

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

---

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.[1]

By Alonzo Church.[2]

1. **Introduction.** In this paper we present a set of postulates for the foundation of formal logic, in which we avoid use of the free, or real, variable, and in which we introduce a certain restriction on the law of excluded middle as a means of avoiding the paradoxes connected with the mathematics of the transfinite.

Our reason for avoiding use of the free variable is that we require that every combination of symbols belonging to our system, if it represents a proposition at all, shall represent a particular proposition, unambiguously, and without the addition of verbal explanations. That the use of the free variable involves violation of this requirement, we believe is readily seen. For example, the identity

On Understanding Data Abstraction, Revisited

William R. Cook

University of Texas at Austin

wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming, which continues to this day. Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form.

On Understanding Data Abstraction, Revisited

William R. Cook
University of Texas at Austin
wcook@cs.utexas.edu

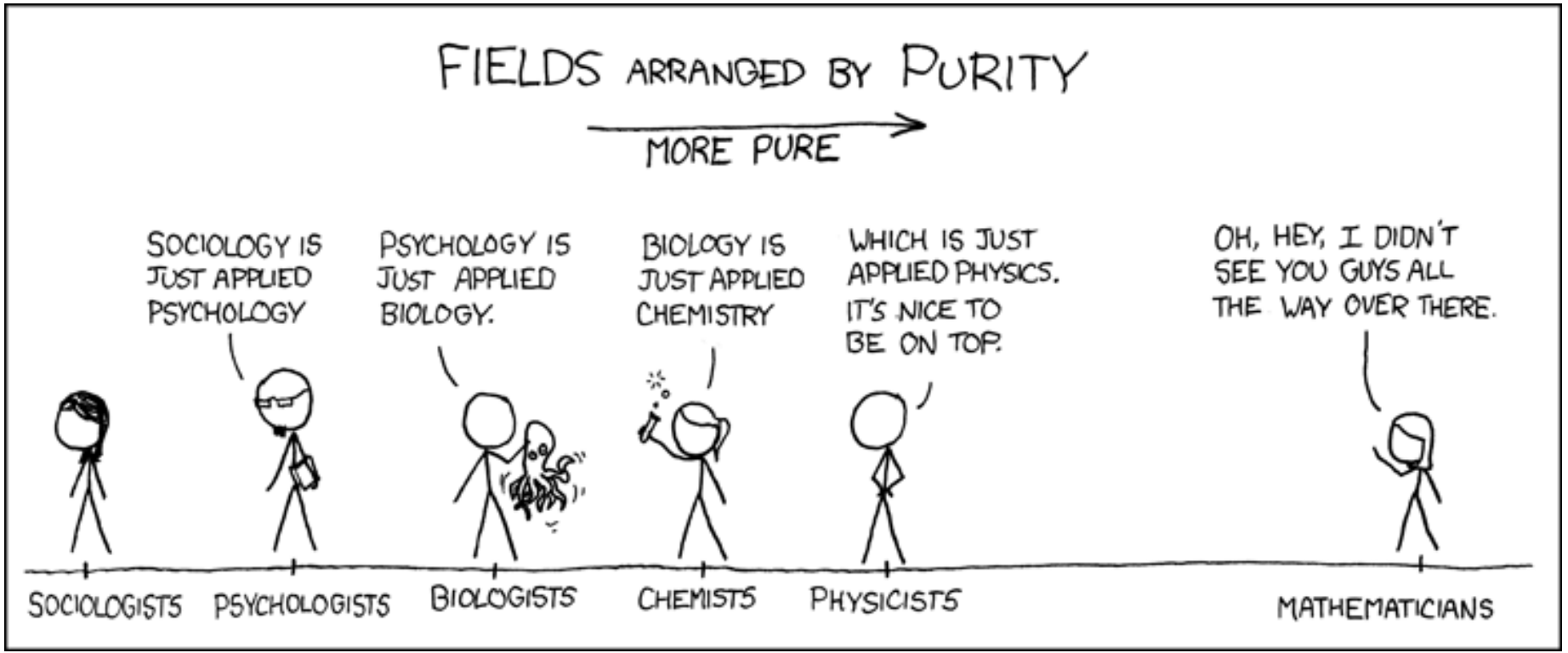
λ-calculus was the first object-oriented language.

Abstract In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper entitled "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming. Cardelli and Wegner's paper has been cited over 1,000 times and, 25 years later, there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

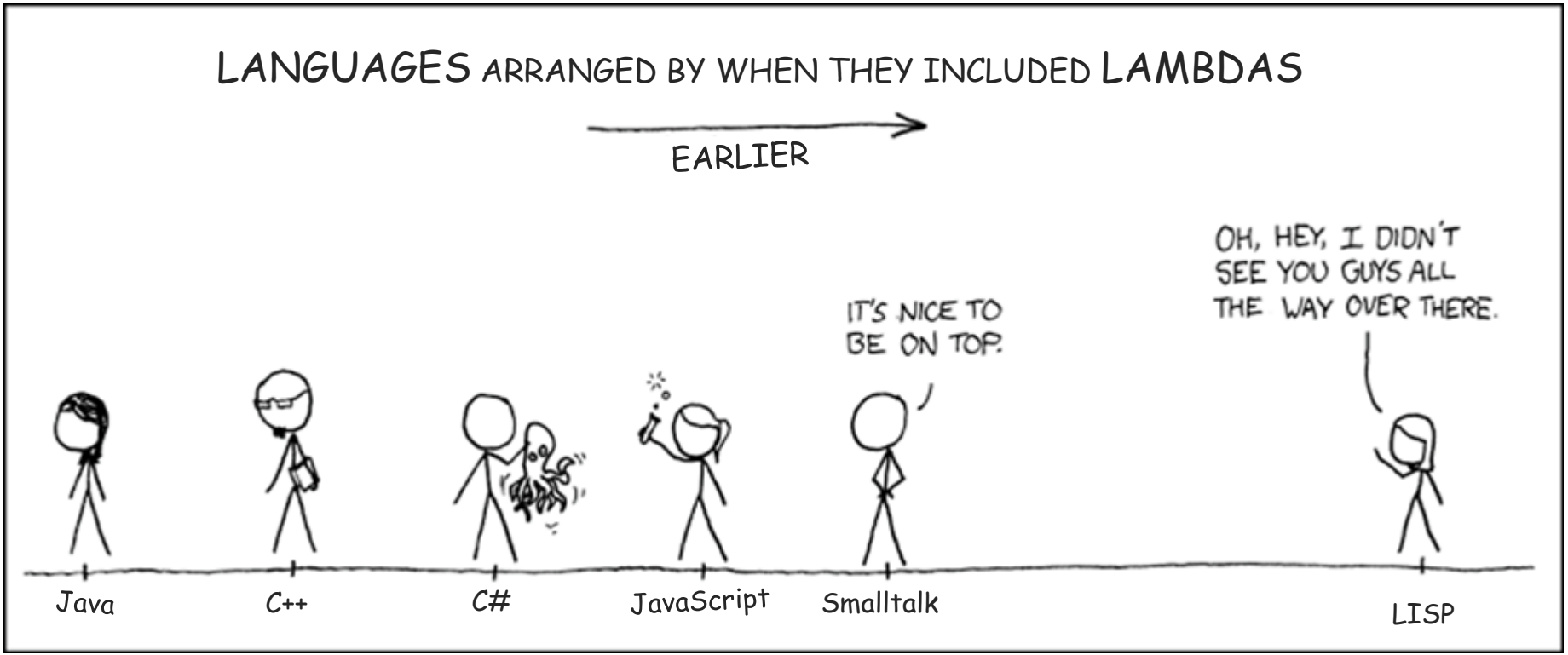
Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language

...what's the point of asking this question? Everyone knows the answer. I think I do. Sure, there are a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form.


FIELDS ARRANGED BY PURITY
MORE PURE
SOCIOLOGY IS JUST APPLIED PSYCHOLOGY
PSYCHOLOGY IS JUST APPLIED BIOLOGY.
BIOLOGY IS JUST APPLIED CHEMISTRY
WHICH IS JUST APPLIED PHYSICS. IT'S NICE TO BE ON TOP.
OH, HEY, I DIDN'T SEE YOU GUYS ALL THE WAY OVER THERE.
SOCIOLOGISTS
PSYCHOLOGISTS
BIOLOGISTS
CHEMISTS
PHYSICISTS
MATHEMATICIANS

xkcd.com/435



with apologies to xkcd.com/435

LISP 1.5 Programmer's Manual

The Computation Center
and Research Laboratory of Electronics

Massachusetts Institute of Technology

```
(defun fibonacci (n)
    (cond ((< n 2) n)
          (t (+ (fibonacci (- n 1))
                (fibonacci (- n 2))))))
```

```
(defun fibonacci (n)
    (if (< n 2)
        n
        (+ (fibonacci (- n 1))
           (fibonacci (- n 2)))))
```

Machine learning

Machine learning algorithm walks into a bar.

Bartender asks, “What’ll you have?”

Algorithm says, “What’s everyone else having?”

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Permission to make digital or hard copies of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reproduce, store, transmit, or use substantial excerpts of this work requires specific permission as do all publications or systematic or multiple distribution.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

© 1978 ACM 0001-0782/78/0800-0613 $00.75

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more.

Each successive version incorporates, with a little cleaning up, all the features of its peers plus a few more.

Java, as a language,
is interesting.

Daniel Terhorst-North

No, it's not.

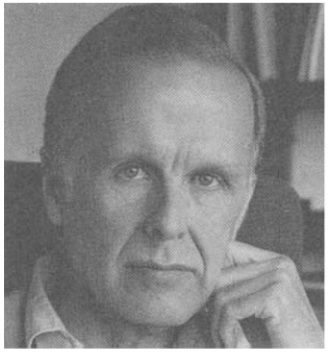
Kevlin Henney

That's the point.

Martin Fowler

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



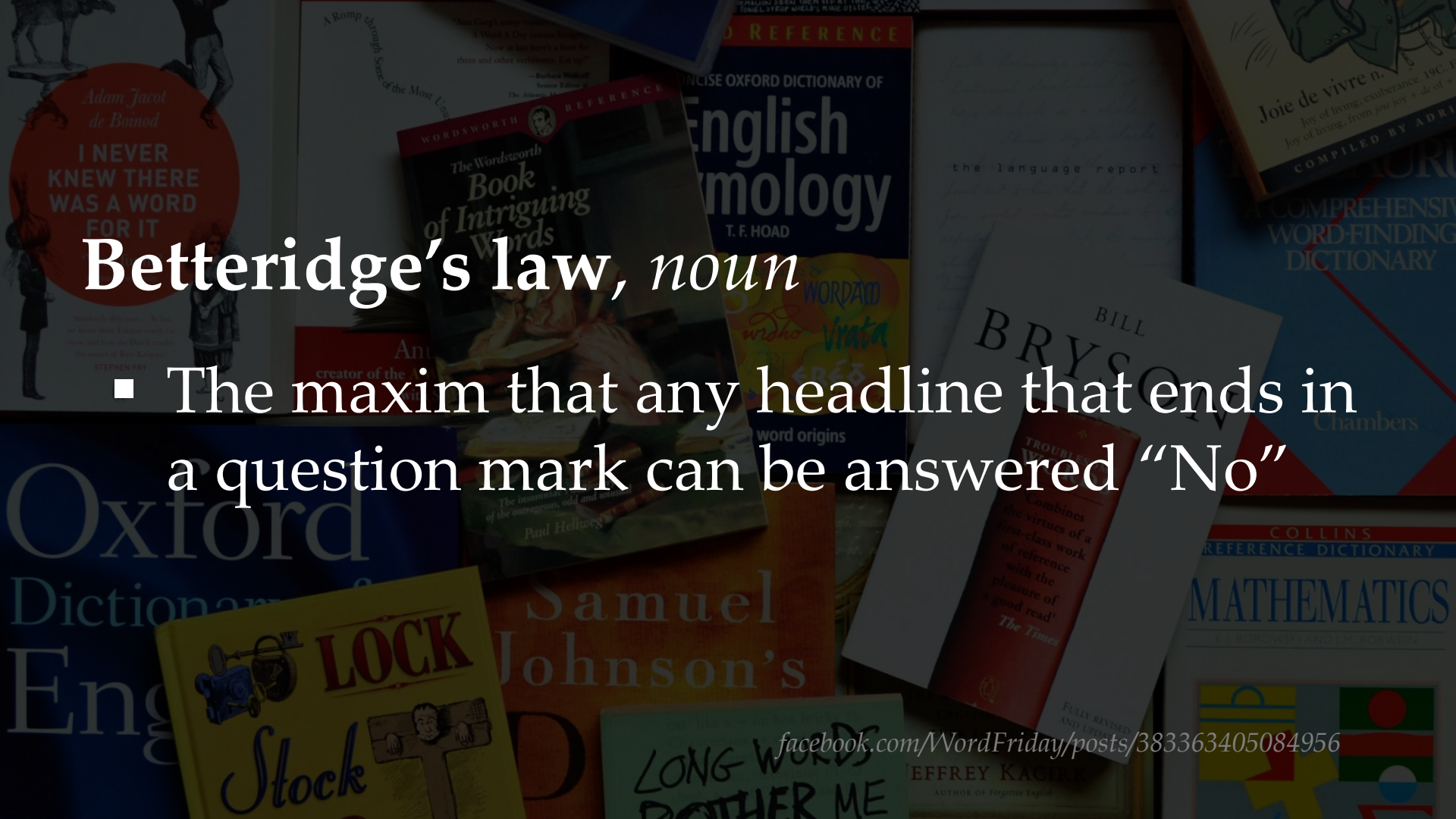
General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

© 1978 ACM 0001-0782/78/0800-0613 $00.75

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

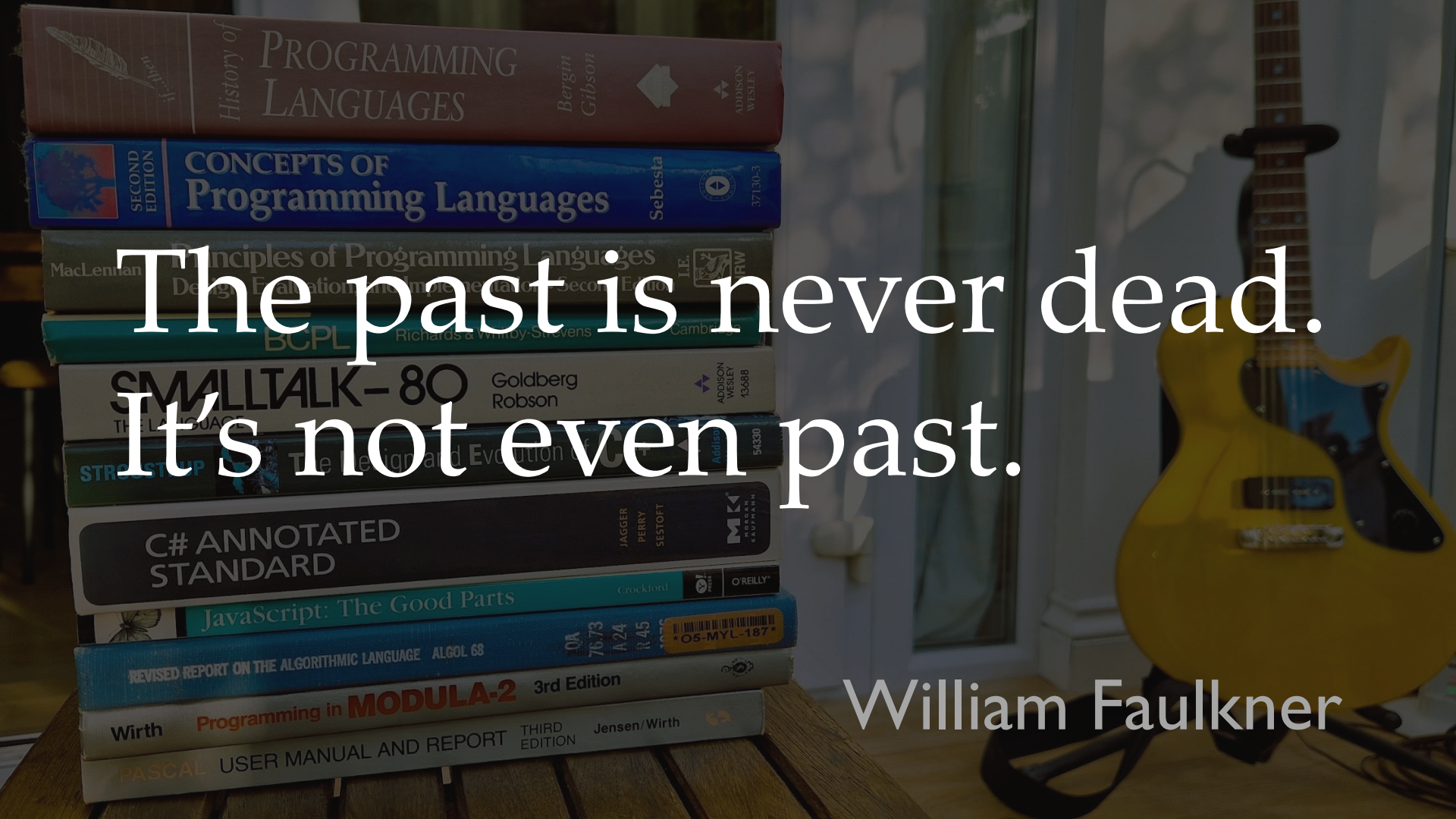
An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.



Betteridge's law, *noun*

- The maxim that any headline that ends in a question mark can be answered "No"

facebook.com/WordFriday/posts/383363405084956


The past is never dead.
It's not even past.
William Faulkner