NPRG075

History and philosophy of programming

Tomáš Petříček, 204 (2nd floor)

petricek@d3s.mff.cuni.cz

https://tomasp.net | @tomaspetricek

Lectures: Tuesday 12:20, S6

https://d3s.mff.cuni.cz/teaching/nprg075

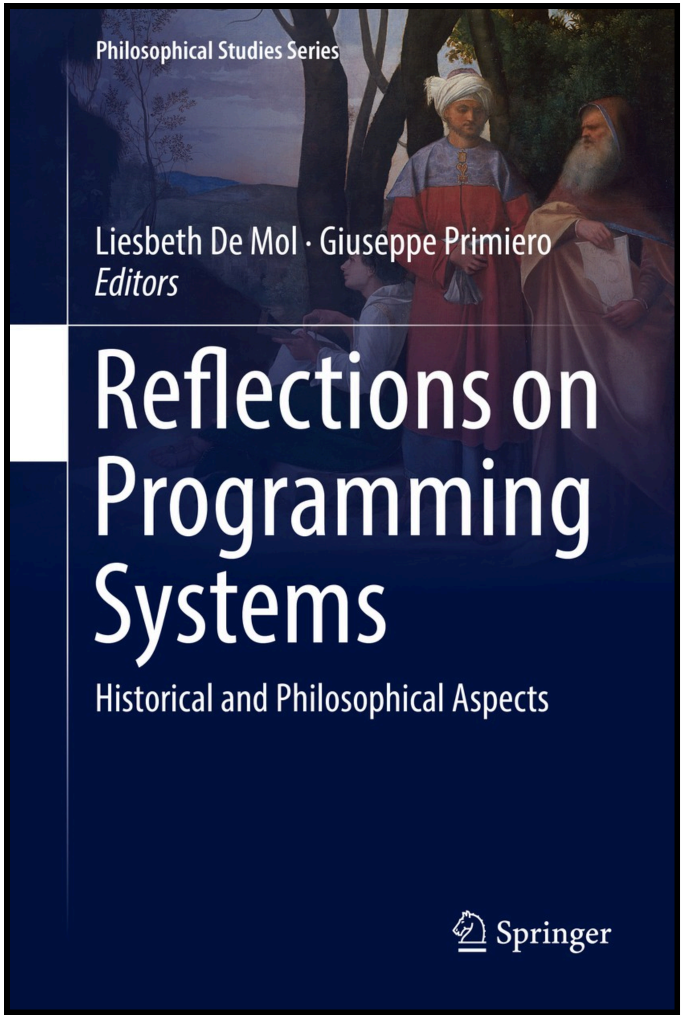
Philosophy of science

Why does it matter?

Philosophy of science

What can we learn about programming?

- What designers assume and never question
- How to understand odd designs of the past
- What is the nature of programming concepts
- What social forces shape programming



What do philosophers do?

Origins, languages, systems, correctness

How could it have gone differently?

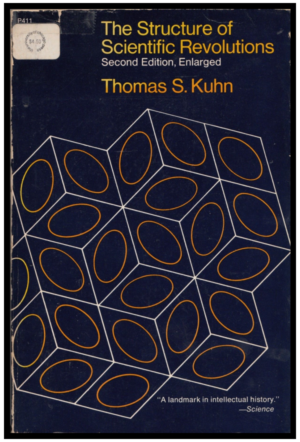
Reflections on ethics, politics, development

What if we took one aspect as primary?

Doing philosophy of programming

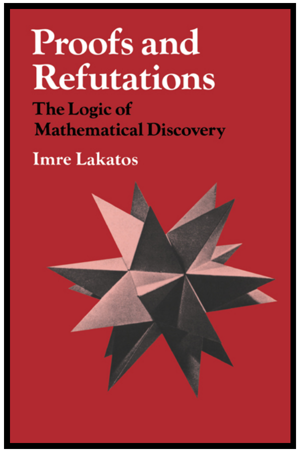
Methods

Try to explain how scientists think and work



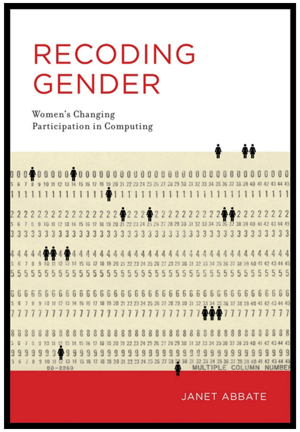
Entities

How concepts evolve & what are they?



Social forces

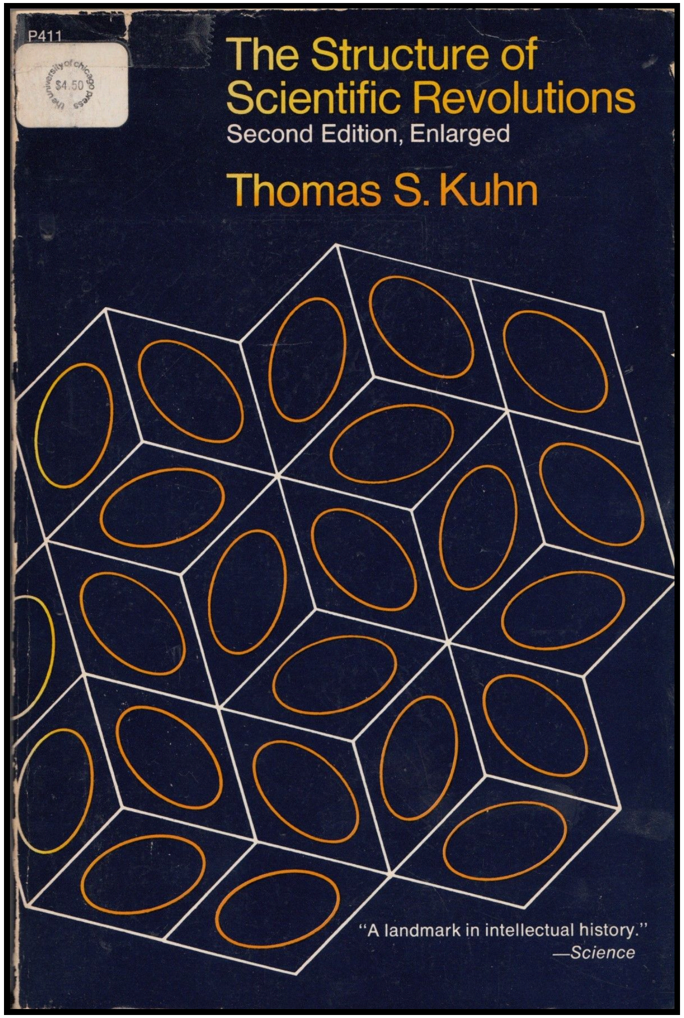
How social aspects shape technology



Paradigm shifts

Classic philosophy of science

Scientific revolutions



Periods of normal science disrupted by revolutions

New era with new assumptions when the old ways stop working

New incommensurable with the old thinking

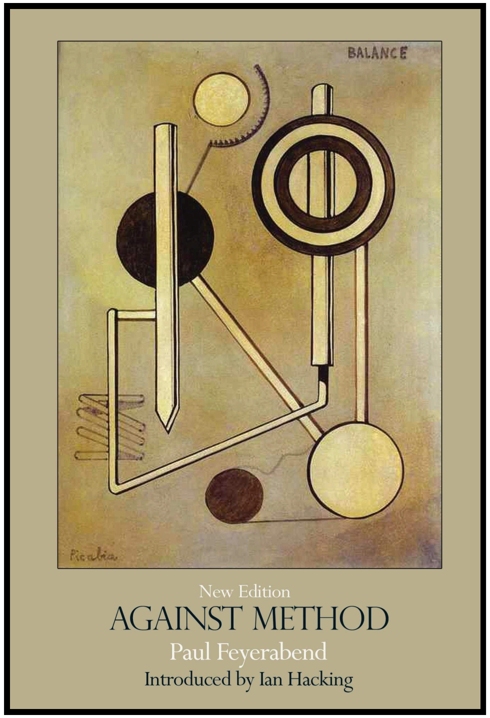
Philosophy of science

Research programmes (Lakatos)

- Groups of scientists share assumptions
- Explain failures by blaming secondary auxiliary assumptions

Against method (Feyerabend)

- No single rule explains science
- Hard to say what is reasonable!



Case study

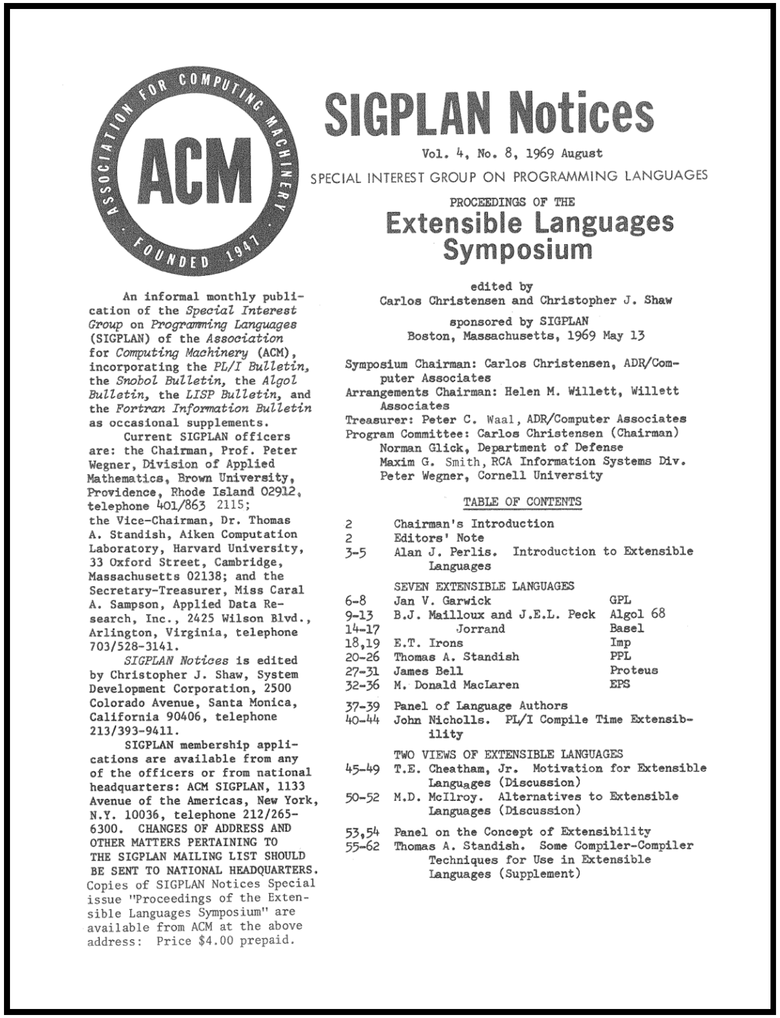
Extensible programming languages

- Extensibility in programming language design (Standish, 1975)
- tinyurl.com/nprg075-extensible (PDF)

What is the idea?

- Look at page 2 (left column)
- Can you make sense of the list?
- Are there extensible programming languages today?

Chairman's intro (1969)



ASSOCIATION FOR COMPUTING MACHINERY
ACM
FOUNDED 1947

SIGPLAN Notices

Vol. 4, No. 8, 1969 August

SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES

PROCEEDINGS OF THE
Extensible Languages Symposium

edited by
Carlos Christensen and Christopher J. Shaw

sponsored by SIGPLAN
Boston, Massachusetts, 1969 May 13

Symposium Chairman: Carlos Christensen, ADR/Computer Associates
Arrangements Chairman: Helen M. Willett, Willett Associates
Treasurer: Peter C. Waal, ADR/Computer Associates
Program Committee: Carlos Christensen (Chairman)
Norman Glick, Department of Defense
Maxim G. Smith, RCA Information Systems Div.
Peter Wegner, Cornell University

An informal monthly publication of the *Special Interest Group on Programming Languages* (SIGPLAN) of the *Association for Computing Machinery* (ACM), incorporating the *PL/I Bulletin*, the *Snobol Bulletin*, the *Algol Bulletin*, the *LISP Bulletin*, and the *Fortran Information Bulletin* as occasional supplements.

Current SIGPLAN officers are: the Chairman, Prof. Peter Wegner, Division of Applied Mathematics, Brown University, Providence, Rhode Island 02912, telephone 401/863 2115; the Vice-Chairman, Dr. Thomas A. Standish, Aiken Computation Laboratory, Harvard University, 33 Oxford Street, Cambridge, Massachusetts 02138; and the Secretary-Treasurer, Miss Caral A. Sampson, Applied Data Research, Inc., 2425 Wilson Blvd., Arlington, Virginia, telephone 703/528-3141.

SIGPLAN Notices is edited by Christopher J. Shaw, System Development Corporation, 2500 Colorado Avenue, Santa Monica, California 90406, telephone 213/393-9411.

SIGPLAN membership applications are available from any of the officers or from national headquarters: ACM SIGPLAN, 1133 Avenue of the Americas, New York, N.Y. 10036, telephone 212/265-6300. CHANGES OF ADDRESS AND OTHER MATTERS PERTAINING TO THE SIGPLAN MAILING LIST SHOULD BE SENT TO NATIONAL HEADQUARTERS.

Copies of SIGPLAN Notices Special issue "Proceedings of the Extensible Languages Symposium" are available from ACM at the above address: Price $4.00 prepaid.

TABLE OF CONTENTS

The ultimate [objective] is simple and attractive. A single universal programming system [that] includes a base language & a meta-language.

A program [consists of], statements in the meta-language which expand (...) the base language, [followed by a program in the derived language.]

A failed 1960s quest

Universal Language

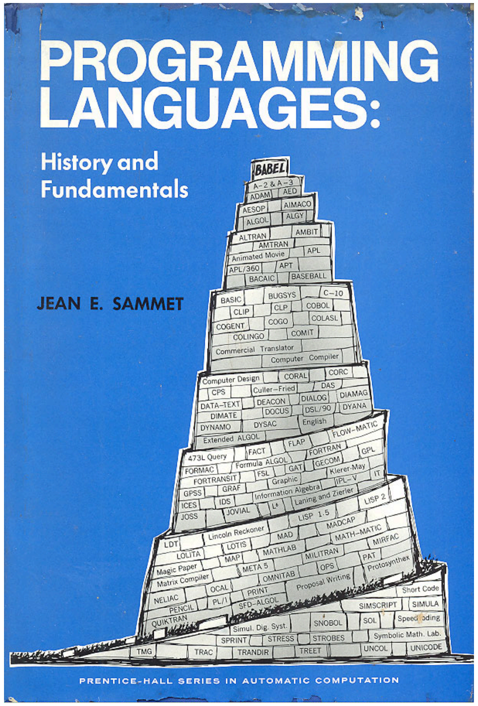
- Respectable disciplines (physics) have one (mathematics)

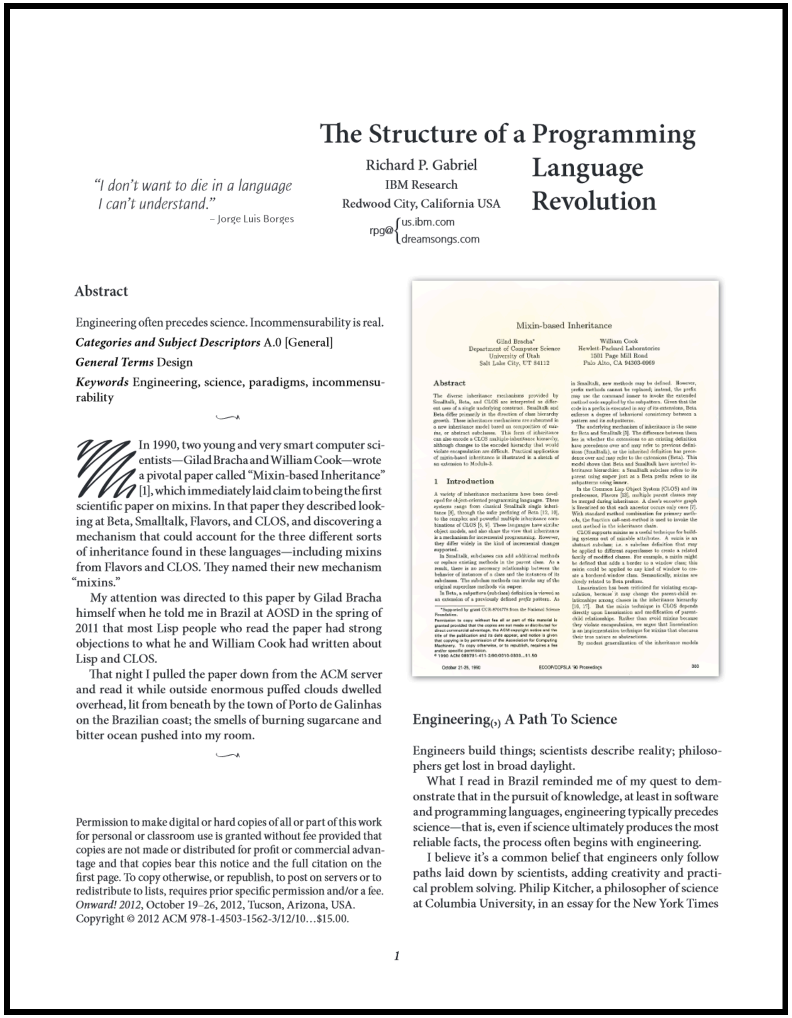
From Algol to PL/1

- Algol 60 only good for scientific use
- PL/1 aims to do all, but is too complex!

Extensible languages

- Last-ditch attempt for universality?





The Structure of a Programming Language Revolution

"I don't want to die in a language I can't understand."
– Jorge Luis Borges

Richard P. Gabriel
IBM Research
Redwood City, California USA
rpg@{us.ibm.com, dreamsongs.com}

Abstract

Engineering often precedes science. Incommensurability is real.

Categories and Subject Descriptors A.0 [General]

General Terms Design

Keywords Engineering, science, paradigms, incommensurability

In 1990, two young and very smart computer scientists—Gilad Bracha and William Cook—wrote a pivotal paper called "Mixin-based Inheritance" [1], which immediately laid claim to being the first scientific paper on mixins. In that paper they described looking at Beta, Smalltalk, Flavors, and CLOS, and discovering a mechanism that could account for the three different sorts of inheritance found in these languages—including mixins from Flavors and CLOS. They named their new mechanism "mixins."

My attention was directed to this paper by Gilad Bracha himself when he told me in Brazil at AOSD in the spring of 2011 that most Lisp people who read the paper had strong objections to what he and William Cook had written about Lisp and CLOS.

That night I pulled the paper down from the ACM server and read it while outside enormous puffed clouds dwelled overhead, lit from beneath by the town of Porto de Galinhas on the Brazilian coast; the smells of burning sugarcane and bitter ocean pushed into my room.

Mixin-based Inheritance

Gilad Bracha
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

William Cook
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303-0969

Engineering$_{(s)}$ A Path To Science

Engineers build things; scientists describe reality; philosophers get lost in broad daylight.

What I read in Brazil reminded me of my quest to demonstrate that in the pursuit of knowledge, at least in software and programming languages, engineering typically precedes science—that is, even if science ultimately produces the most reliable facts, the process often begins with engineering.

I believe it's a common belief that engineers only follow paths laid down by scientists, adding creativity and practical problem solving. Philip Kitcher, a philosopher of science at Columbia University, in an essay for the New York Times

Programming language revolution

(Gabriel, 2012)

From thinking about programming systems

Running, with evolving state, modified interactively

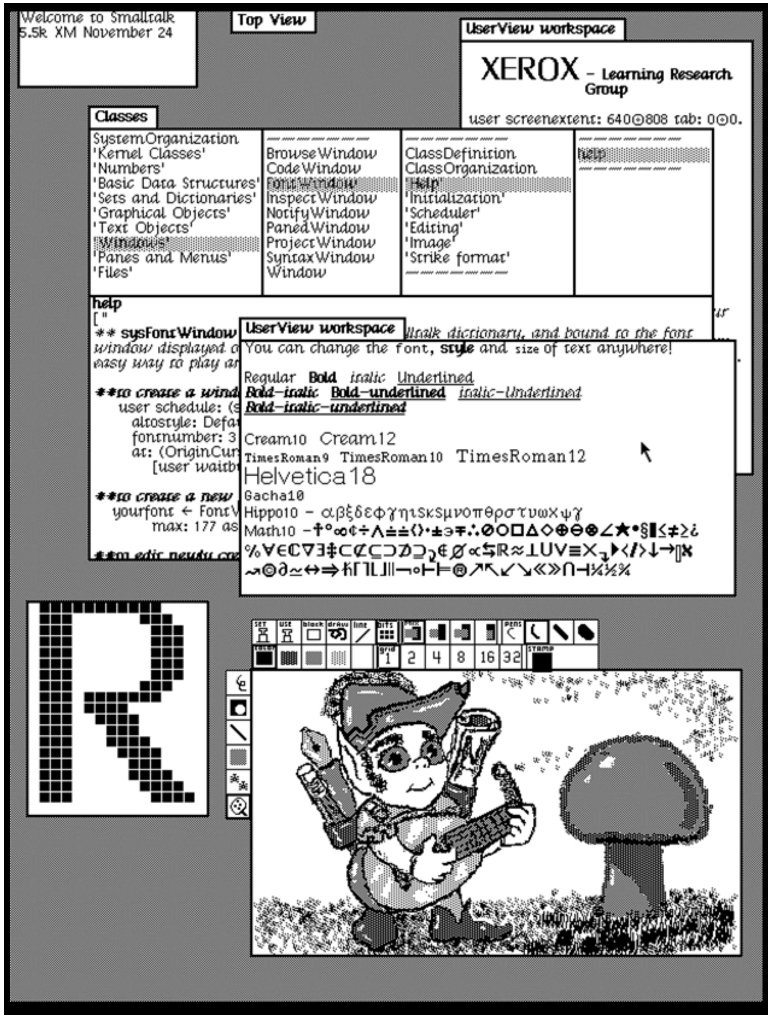
To thinking about programming languages

Relationships in static code

Smalltalk language

"Smalltalk is an object-oriented, dynamically typed reflective programming language"

What makes it interesting?

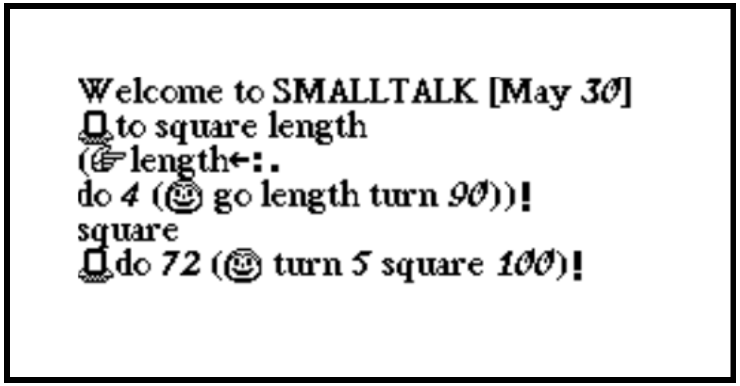


Smalltalk as a programming system

Think not about source code, but about evolving system state!

Demo

Smalltalk 72 and 78



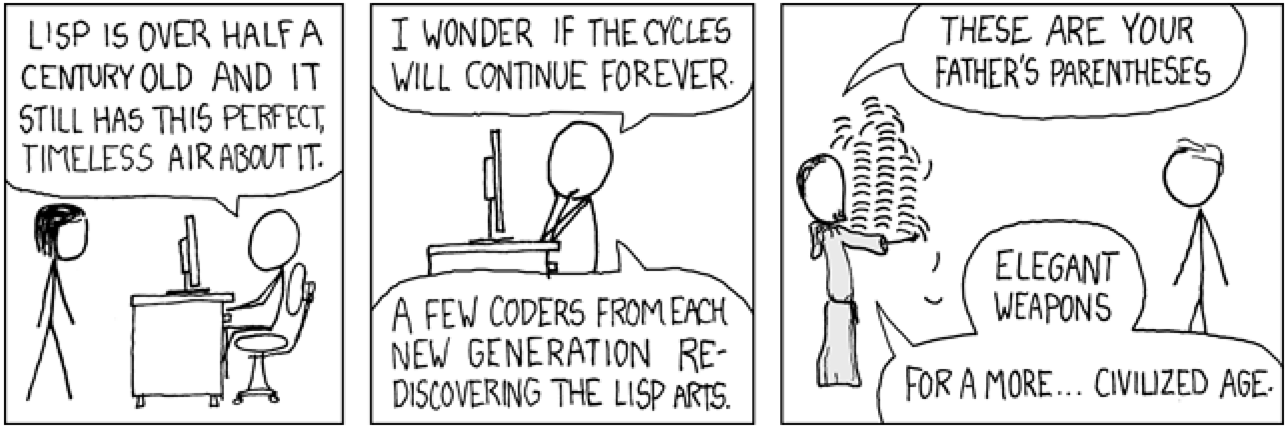
Smalltalk

Programming system view

- Image-based persistence rather than source
- Application ships with developer tools
- Class browser allows inspecting & editing
- Reflection lets the system change itself

LISP language

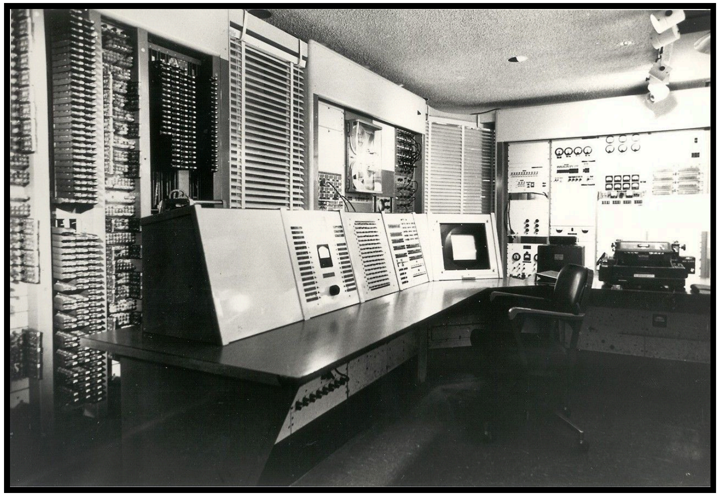
Functional programming language derived from the lambda calculus?



LISP environment

Time-sharing

- Batch processing in the 1950s
- TX-0 ('58) allowed interactive use
- Multi-user machines via teletype



AI research requirements

- Programming with symbolic data
- Interactive experimentation
- Programs that improve themselves

```
*EDITF(APPEND)
 EDIT
*(P ∅ 1∅∅)
 (LAMBDA (X) Y (COND ((NUL X) Z) (T (CONS (CAR) (APPEND (CDR X Y))))))
*(3)
*(2 (X Y))
*P
 (LAMBDA (X Y) (COND & &))
```

LISP editor

(Deutsch, 1967)

Interactive program editing on the terminal

Teletype, not a screen!

Print using: `P`
Delete child: `(3)`
Replace child: `(2 ..)`

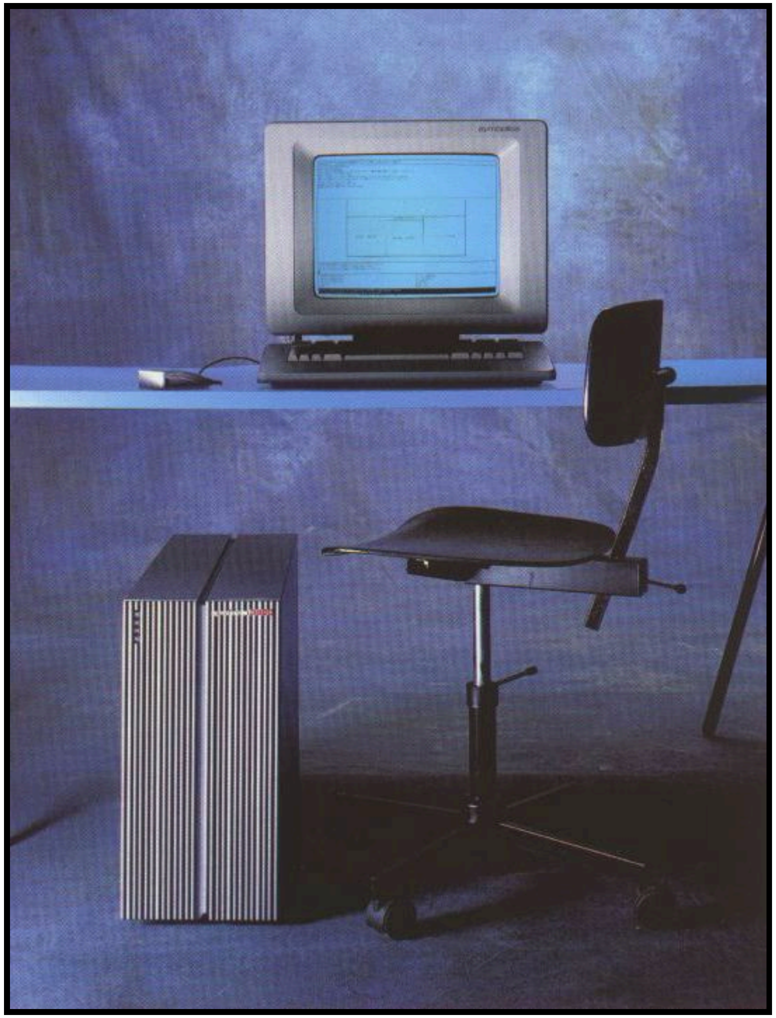
Interlisp: Interactive Lisp

PILOT (1966)

- Edit code via list transformations
- Advising to enhance procedures
- Modifying state of a running system

DWIM (1974)

- Interactive program correction
- Suggests automatic fixes when error occurs
- Do What I Mean / Damn Warren's Infernal Machine



Symbolics Lips Machines (1980s)

Machines optimized for LISP with LISP-based environment

Persistent memory with just cons-cells

Response to new hardware architecture

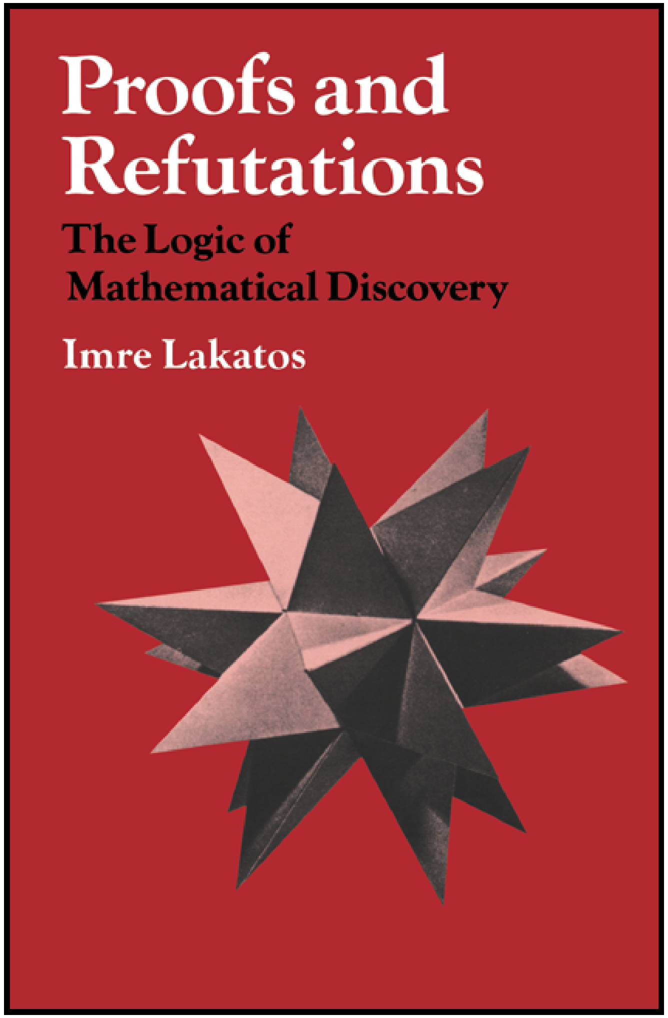
Scientific revolutions

Paradigm shifts in programming

- Understand what people really thought!
- The invention of a programming language
- The shift from systems to languages
- Functional programming "research programme"

Entities

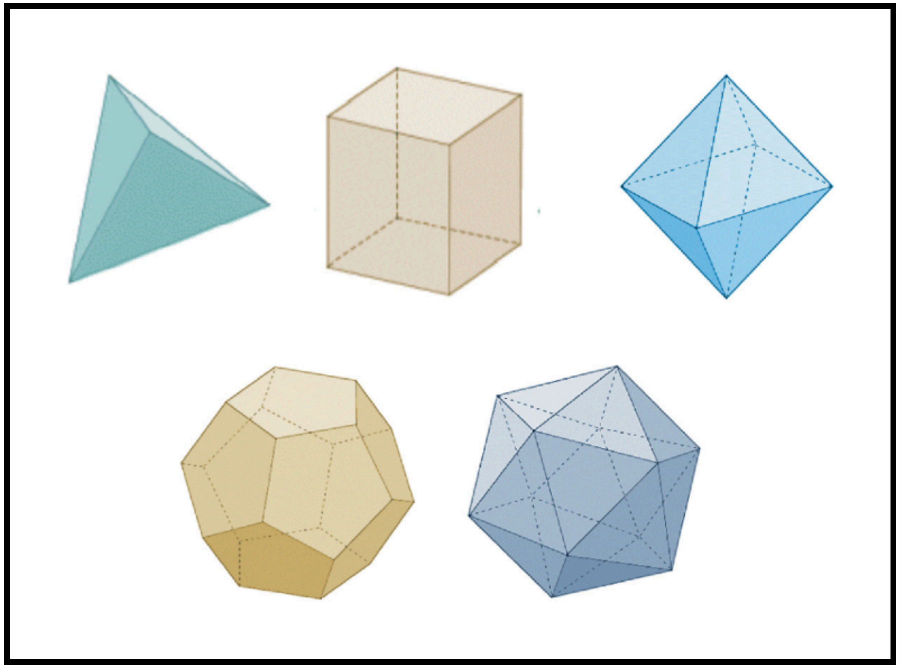
Evolution of programming concepts



How mathematical concepts evolve?

Polyhedra, space, graph, function, convergence, measurable set

How does the definition change and why?



Polyhedra

Euler's formula

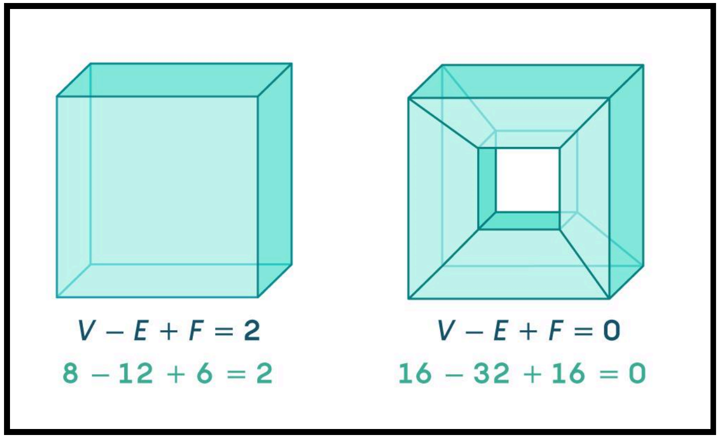
$$V - E + F = 2$$

A polyhedron is a solid whose surface consists of polygonal faces?

Counter example?

Convex polygons!

Through any point in space there will be at least one plane whose cross-section with the polyhedron will consist of one single polygon.



Monster-barring

I turn aside with a shudder of horror from this lamentable plague of functions which have no derivatives.
(Charles Hermite, 1893)

Concepts

Proofs and refutations

- Concept definitions are not constant but change
- Arising from proofs, counter-examples, lemmas
- Monster-barring and exception-barring
- Concept stretching when understanding evolves

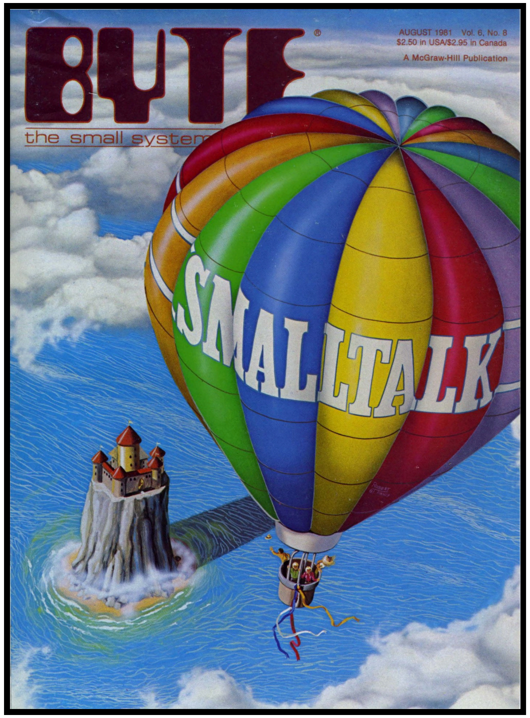
Concepts in programming

Change over time!

- Data types, logical types
- Monads and "railway" metaphor
- Processes become abstract

Multiple forces for change

- New implementation of the concept
- Different metaphor for thinking
- New formalization in a proof



Evolution of types

Implementation & formal modality

Data types like records, modelled as sets

Implementation modality evolves

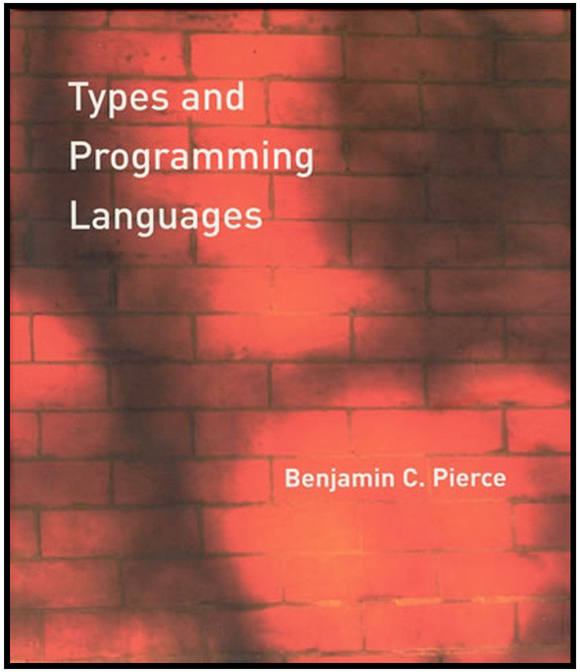
Abstract data types for modularity
Type checking ala lambda calculus

Intuitive modality evolves

Well-typed programs do not go wrong
New type systems based on this

Implementation modality evolves

Types for documentation and editor tooling



Understanding Monads

What are monads

- Origins in category theory
- Abstraction in functional programming
- Used for stateful computations

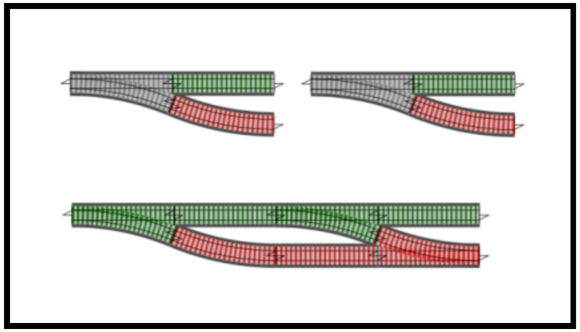
Writing about monads

- Compare how mathematicians and programmers talk about monads!
- tinyurl.com/nprg075-mcat
- tinyurl.com/nprg075-mprog

Evolution of monads

Formal and intuitive modality

Standard construction in algebraic topology
Monad as a "box" intuition



Implementation modality appears

Used for sequencing effectful computations
Definition in terms of *bind* and *return*

Implementation & intuition evolves

Monads in Haskell and the **do** notation
Monad as a "sequencing" intuition

Concepts

Programming language design

- There is more to concepts than just a name
- Ideas come from logic, linguistics, biology!
- Beware of concept stretching as with types?
- Capture a new intuition in the design?

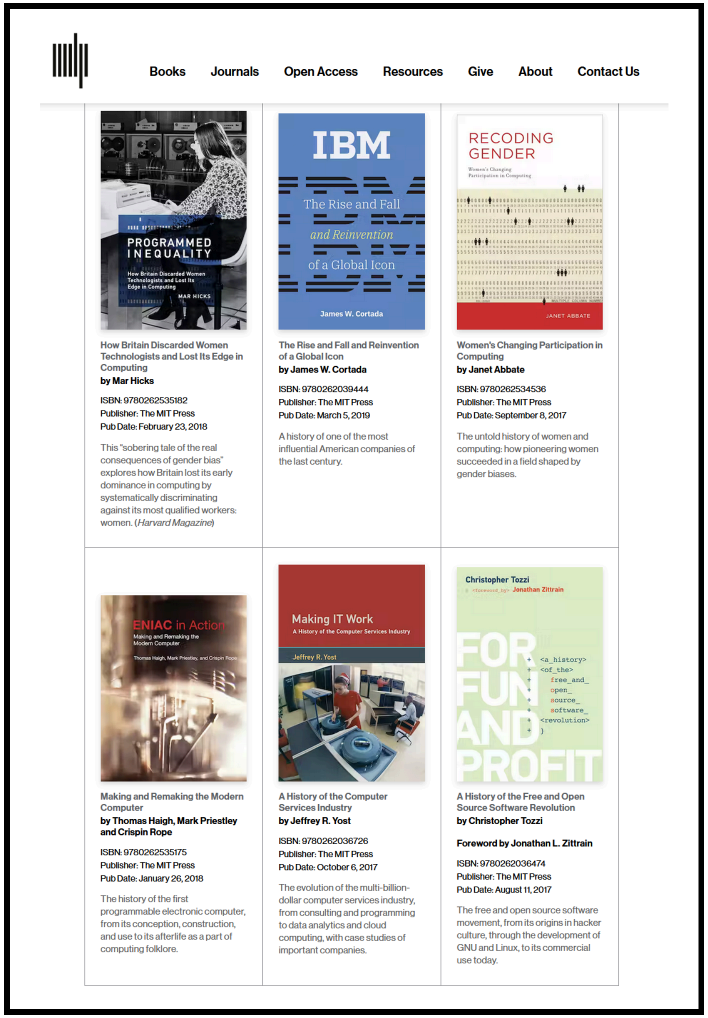
Social forces

What shapes programming?

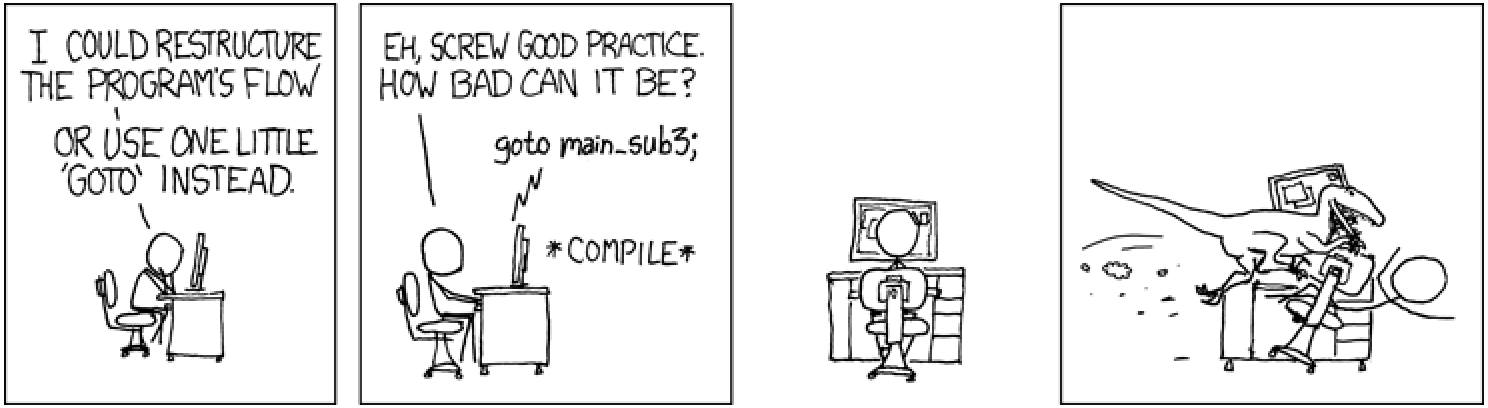
Social history of computing

How commercial interests or gender bias shape computing

Redefinition of programming as more masculine software engineering in the 1960s



Structured programming



Goto considered harmful (1968)

The quality of programmers is a decreasing function of the density of go to statements in the programs they produce.

Problems with goto

- Hard to reason about informally
- Hard to reason about formally
- Code structure does not match runtime behaviour

Edgar Dijkstra: Go To Statement Considered Harmful

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** $A1$ **else** $A2$), 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

Structured programming

Not obvious at the time!

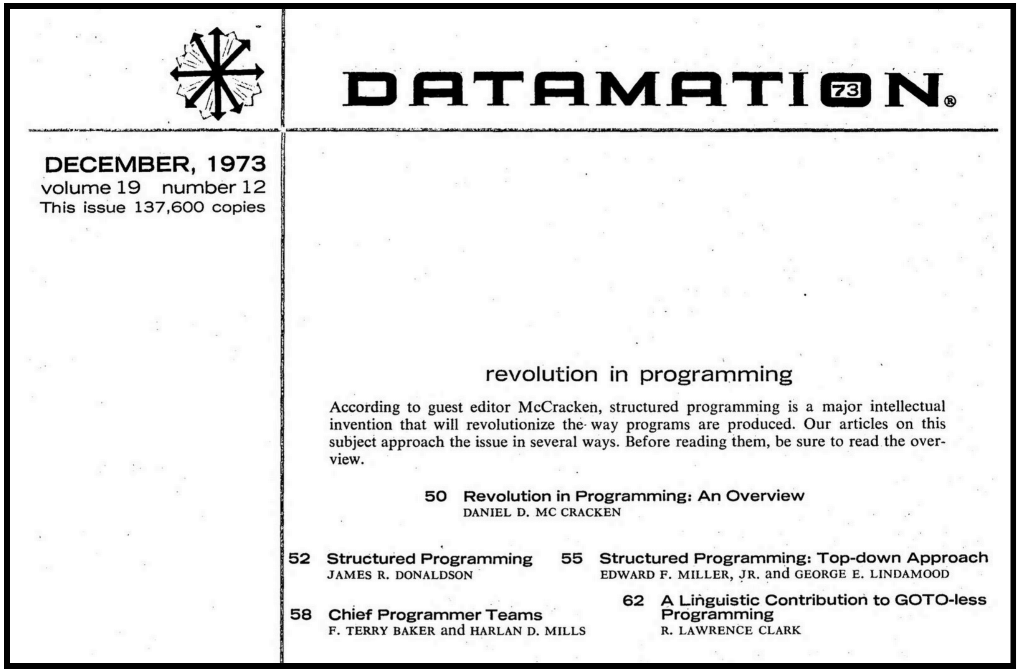
- Everyone used to assembly!
- Can the compiler optimize code?
- Is it possible to avoid gotos?

$s = 1; i = 1;$		$s = 1; i = 1;$
while $i < n$ do	$L1$	if $i = n$ then goto $L2$
$\quad i = i + 1;$		$i = i + 1;$
$\quad s = s \times i;$		$s = s \times i;$
end		goto $L1$;
print(s);	$L2$:	print(s);

Structured Programming Theorem (1966)

Us converts waved this interesting bit of news under the noses of the unreconstructed assembly-language programmers who kept trotting forth twisty bits of logic and saying, 'I betcha can't structure this.'

Datamation (1973)



DATAMATION

DECEMBER, 1973
volume 19 number 12
This issue 137,600 copies

revolution in programming

According to guest editor McCracken, structured programming is a major intellectual invention that will revolutionize the way programs are produced. Our articles on this subject approach the issue in several ways. Before reading them, be sure to read the overview.

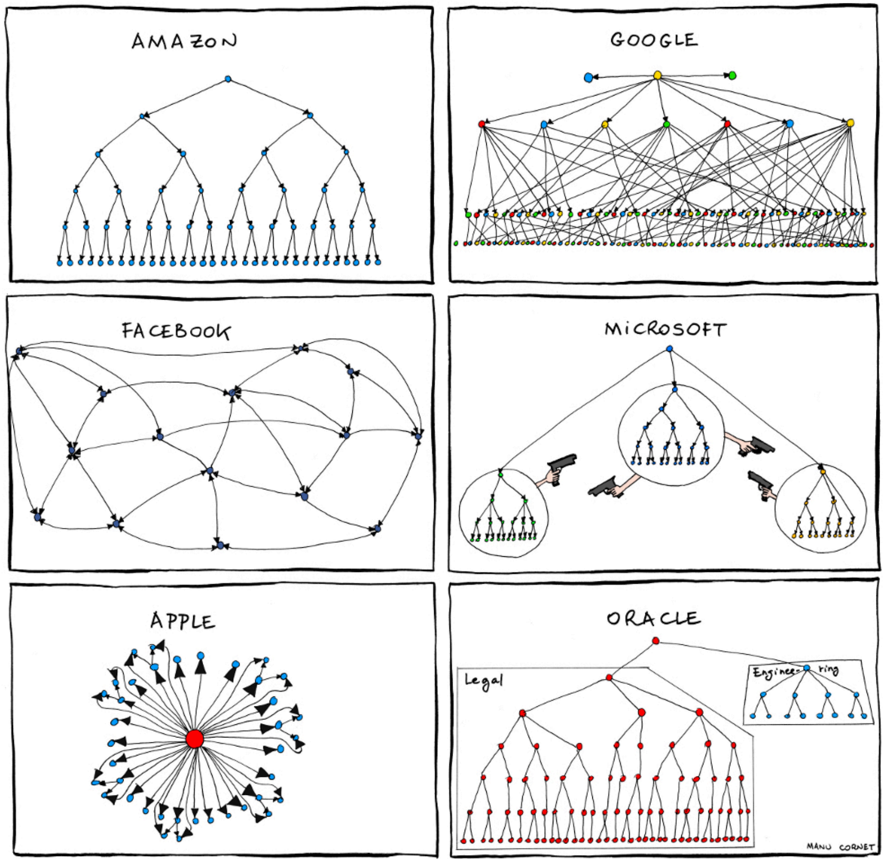
What is structured programming and how to do it in practice

From engineering concept to managerial concept

Chief programmer teams

Top-down management technique

- Structured programming for organizing people
- Chief-programmer leading & dividing code
- Supported by programmers, secretary, backup
- Hostile exchanges between Dijkstra and Mills



Conway's law

Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.

Social forces

Programming language design

- Language features linked to social structures
- Organizational structure and escape hatches
- Structured, microservices, information hiding
- Origins of languages - COBOL, Fortran, Algol

Conclusions

History and philosophy

Figure 1. A typical design process

History and philosophy

Learning from the past

Complex reasons why & how programming ideas work and do not work

Reading

```
10 PRINT CHR$(205.5+RND(1));
20 GOTO 10
```

- 15: REM Variations in Basic
- https://10print.org (look for the PDF)

Why should you read this?

- Fun look at an unexpected bit of programming history
- What can we learn from the past?

Conclusions

History and philosophy of programming

- Scientific paradigms and paradigm shifts
- The history of programming concepts
- How social forces shape programming

Tomáš Petříček, 204 (2nd floor)
petricek@d3s.mff.cuni.cz
https://tomasp.net | @tomaspetricek
https://d3s.mff.cuni.cz/teaching/nprg075

References (1/2)

Philosophy of science

- Kuhn, T. S., (2012). The Structure of Scientific Revolutions. Chicago
- Feyerabend, P. (1975). Against Method. Verso
- Lakatos, I. (1976). Proofs and Refutations. Cambridge

History & reflections

- De Mol, L., Primiero, G. eds. (2018). Reflections on Programming Systems: Historical and Philosophical Aspects. Springer
- Gabriel, R. (2012). The Structure of a Programming Language Revolution. Onward!
- Petricek, T. (2022). Cultures of Programming. Draft
- Petricek, T. (2018). What we talk about when we talk about monads

References (2/2)

Historical materials

- Teitelman, W. (1966). PILOT: A Step Toward Man-Computer Symbiosis. MIT
- Teitelman, W. (1974). Interlisp Reference Manual. Xerox PARC
- Deutsch, P. (1967). Preliminary Guide to the LISP Editor. Berkeley
- Dijkstra, E. (1968). Go To Statement Considered Harmful. ACM
- McCracken et al. (1973). Revolution in Programming. Datamation 12