A functional programming system

Carlos Kavka

Head of Research and Development

Agenda

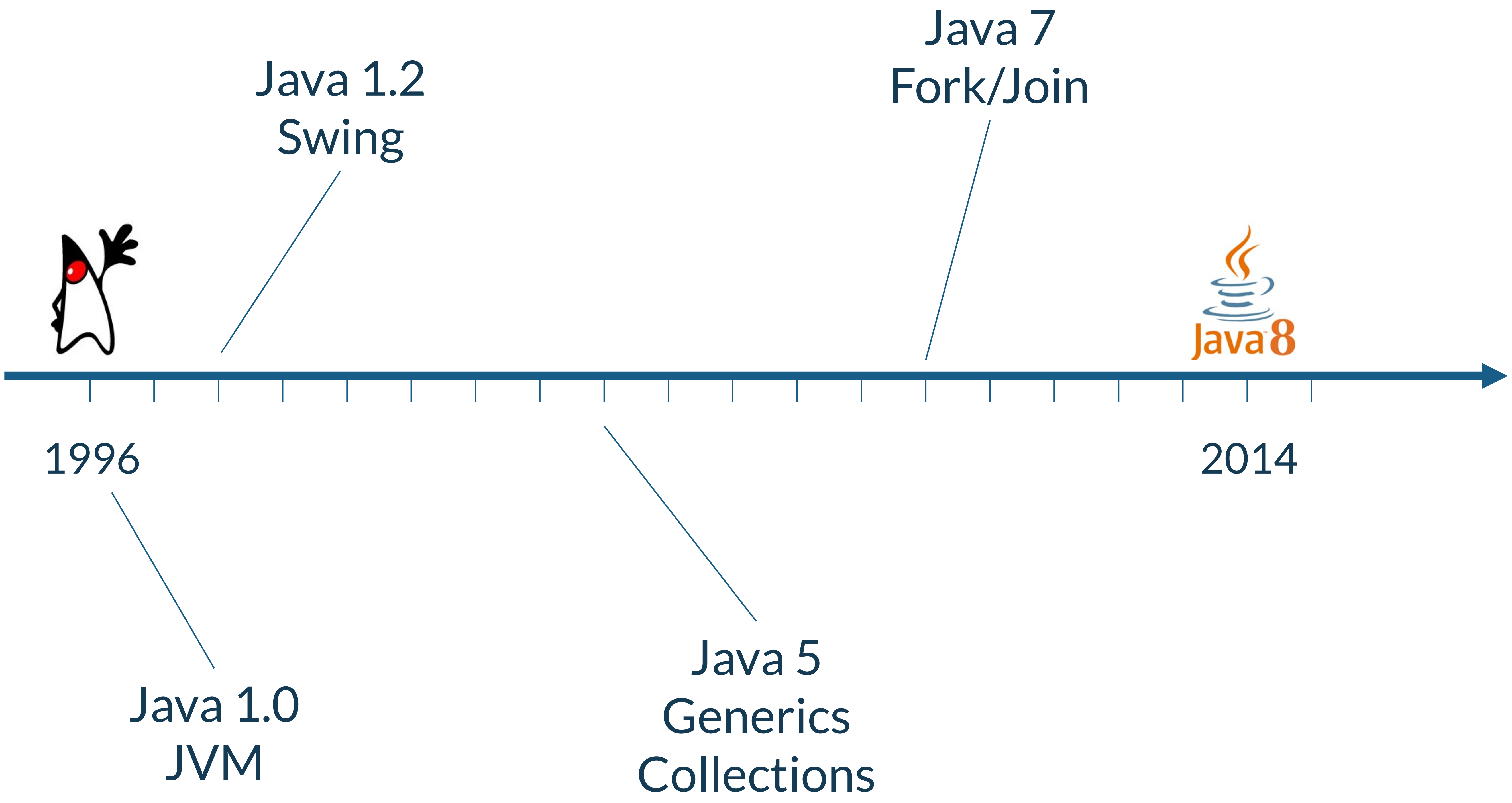
Why functional programming in Java?

The functional programming paradigm

A functional programming system

Analysis

Java evolution till functional programming



1996

Java 1.0
JVM

Java 1.2
Swing

Java 5
Generics
Collections

Java 7
Fork/Join

2014

An alignment with language trends was required!

C#

C++

ERLANG

JS JavaScript

Haskell

Ruby

Scala

Microsoft® Visual F#®

JRuby

Groovy

Java ecosystem

Clojure

Improvements introduced in Java 8

Functional style of programming

Collection enhancements

Stream processing

Optional values

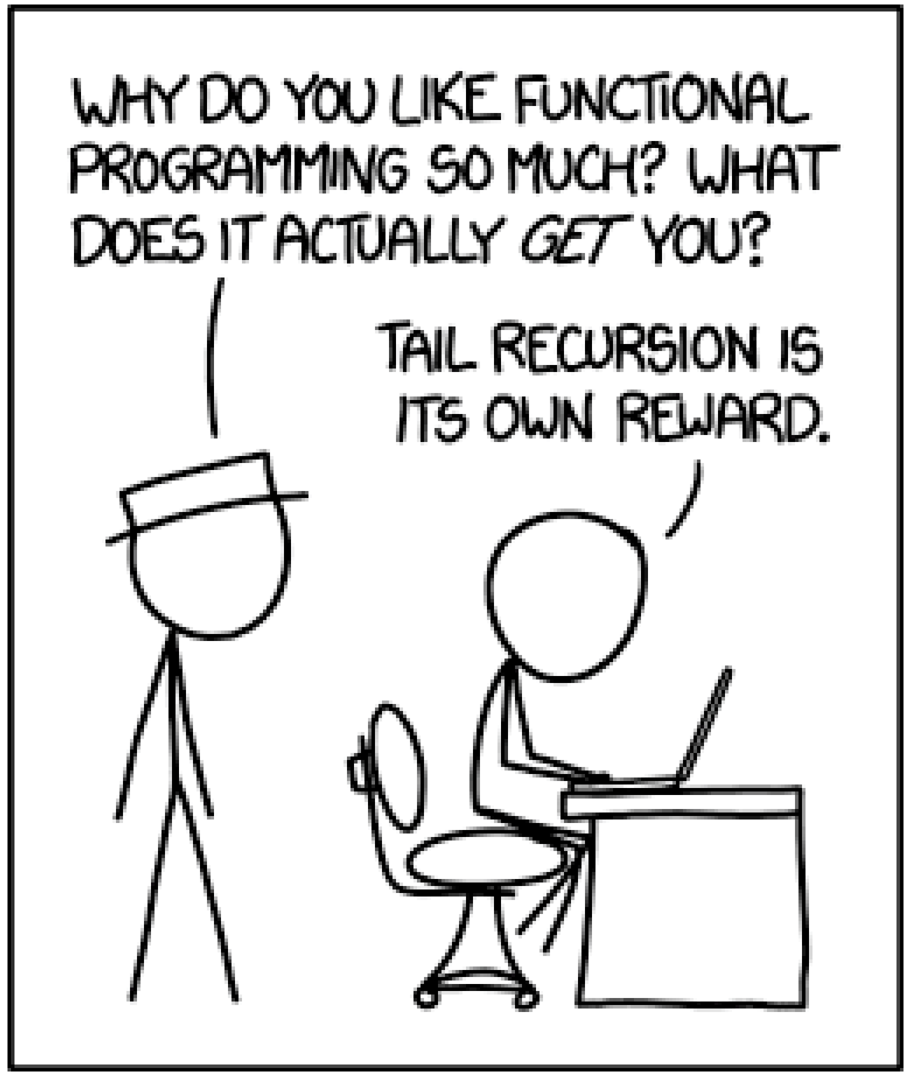
Lambda expressions

Method references

Default methods

Important point!

Using new Java elements is not enough!


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

(*)

A change in the way of thinking is required!

Many "expert" Java programmers use functional features in a really improper way!

(*) reprinted from https://www.explainxkcd.com/wiki/index.php/1270:_Functional

A change in the way of thinking is required!

Imperative programming

x++

style of programming modeled as a sequence of commands that modify state

Functional programming

f(g(x))

programs are expressions and transformations, modeling mathematical formulas

A change in the way of thinking is required!

```
count = 0;
for(i = 0; i < n; i++)
   if (a[i] > 0)
      count++;
```

$$/+ \circ \alpha(> \circ [id, \bar{0}] \rightarrow \bar{1}\ ; \bar{0})$$

programming means tell —declaratively—*what* we want rather than *how* to do it.

Imperative approach

```
count = 0;
for(i = 0; i < n; i++)
   if (a[i] > 0)
      count++;
```

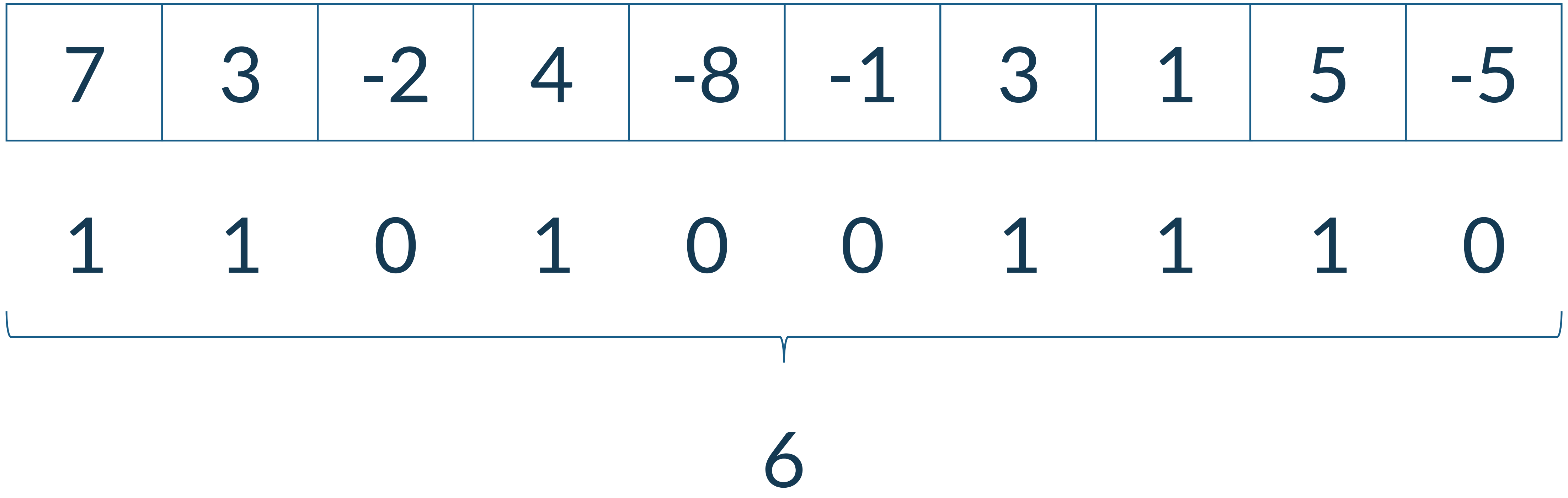
i	0
count	0

a

7	3	-2	4	-8	-1	3	1	5	-5

Functional approach

$$/+ \circ \alpha(> \circ [id, \bar{0}] \rightarrow \bar{1}\,; \bar{0})$$



7	3	-2	4	-8	-1	3	1	5	-5
1	1	0	1	0	0	1	1	1	0

$$\underbrace{1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0}_{6}$$

Comparison

```
count = 0;
for(i = 0; i < n; i++)
   if (a[i] > 0)
      count++;
```

$/+ \circ \alpha(> \circ [id,\bar{0}] \longrightarrow \bar{1} ; \bar{0})$

i	11
count	6

a

7	3	-2	4	-8	-1	3	1	5	-5

7	3	-2	4	-8	-1	3	1	5	-5

$$\underbrace{1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0}_{6}$$

parallelism

mutable objects

different approach:
what vs. how

what happen if we call
twice a function?

What about Oriented Programming?

```
class A {
  int x;
  int getX();
  void setX(int x);
}
```

abstracting over
data

f(g(x))

abstracting over
behavior

Functional programming

Is it new? No.

1930 - Lambda Calculus (A. Church)
1958 - Lisp (J. McCarthy)
...
1977 - FP (J. Backus)
...

What about Java implementation?

- no monads
- reduced lazy evaluation
- little support for immutability

...

however, it is better than nothing! 🙂

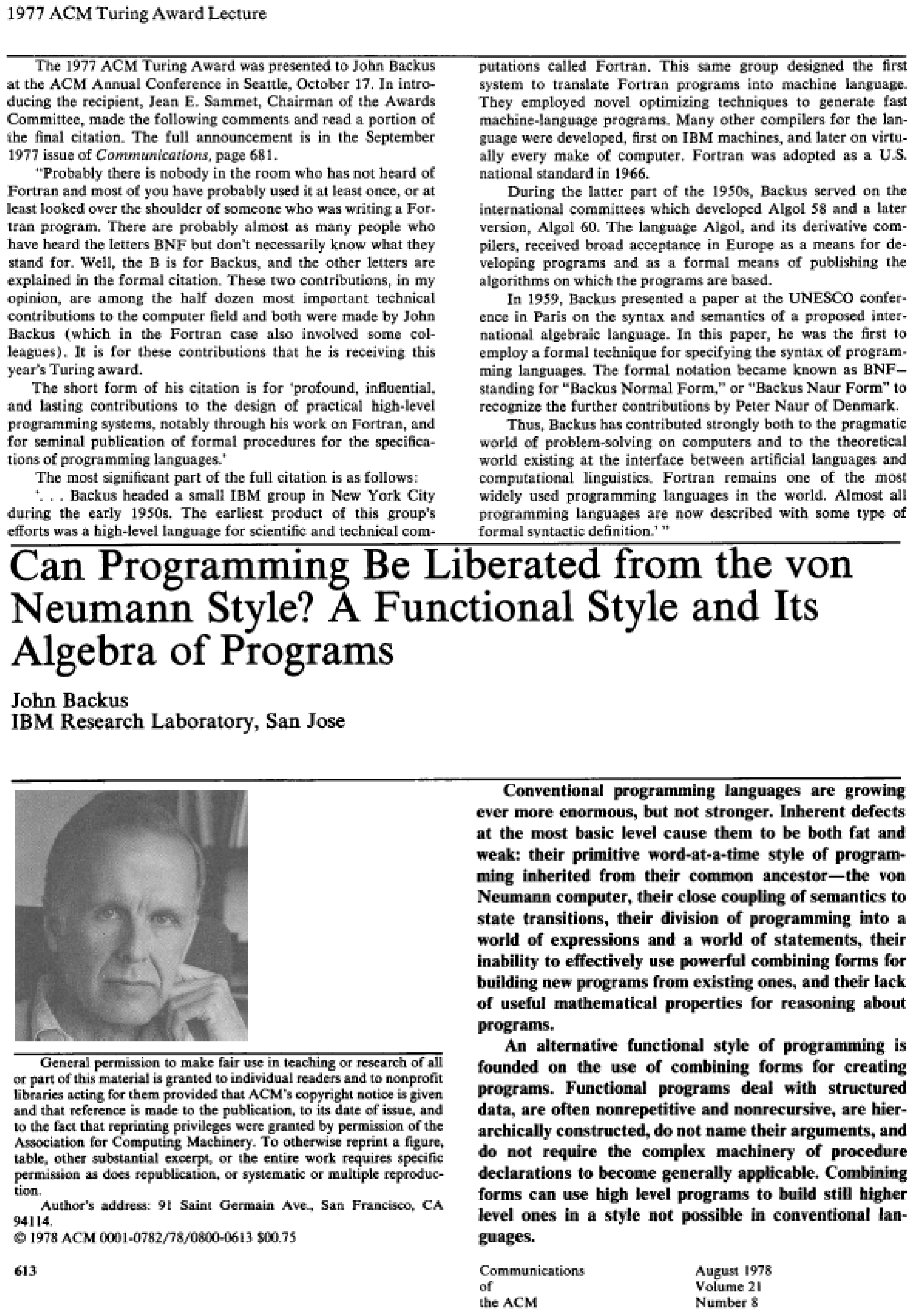
Benefits

- Simpler, cleaner, and easier-to-read code
- Simpler maintenance
- Great for collections!
- Enhanced parallelism/concurrency for multi-core CPUs

ACM Turing Award Lecture by John Backus

a functional programming system

its associated algebra of programs



1977 ACM Turing Award Lecture

The 1977 ACM Turing Award was presented to John Backus at the ACM Annual Conference in Seattle, October 17. In introducing the recipient, Jean E. Sammet, Chairman of the Awards Committee, made the following comments and read a portion of the final citation. The full announcement is in the September 1977 issue of *Communications*, page 681.

"Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B is for Backus, and the other letters are explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for 'profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specifications of programming languages.'

The most significant part of the full citation is as follows:

'. . . Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical computations called Fortran. This same group designed the first system to translate Fortran programs into machine language. They employed novel optimizing techniques to generate fast machine-language programs. Many other compilers for the language were developed, first on IBM machines, and later on virtually every make of computer. Fortran was adopted as a U.S. national standard in 1966.

During the latter part of the 1950s, Backus served on the international committees which developed Algol 58 and a later version, Algol 60. The language Algol, and its derivative compilers, received broad acceptance in Europe as a means for developing programs and as a formal means of publishing the algorithms on which the programs are based.

In 1959, Backus presented a paper at the UNESCO conference in Paris on the syntax and semantics of a proposed international algebraic language. In this paper, he was the first to employ a formal technique for specifying the syntax of programming languages. The formal notation became known as BNF—standing for "Backus Normal Form," or "Backus Naur Form" to recognize the further contributions by Peter Naur of Denmark.

Thus, Backus has contributed strongly both to the pragmatic world of problem-solving on computers and to the theoretical world existing at the interface between artificial languages and computational linguistics. Fortran remains one of the most widely used programming languages in the world. Almost all programming languages are now described with some type of formal syntactic definition.' "

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 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.

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

Definition

An FP system comprises the following:

1. a set O of objects
2. a set F of functions that map objects into objects
3. an operation: application
4. a set of functional forms; used to combine existing functions or objects, to form new functions in F
5. a set of definitions that define some functions in F and assign a name to each

Objects

An object x is either:

- an atom
- a sequence $<x_1, ..., x_n>$, whose elements x_i are objects
- ⊥ (undefined)

The sequence constructor is ⊥-preserving:
if x is a sequence with ⊥ as an element, then x = ⊥

Objects - examples

Numeric atoms

1
34

Lists

Ø
<1,2,Ø,3>

Boolean atoms

T
F

Lists

<1, 2, 3>
<1, <2,3,4>>
<<1,2>,<3,4,5>>

Undefined preservation

<1,⊥> = ⊥

An operation: the application

If f is a function and x is an object, then f:x is an application and denote the result of the application of f to x

+:<1,2> = 3

tl:<5,3,8> = <3,8>

1:<5,3,8> = 5
2:<5,3,8> = 3

Functions

All functions map objects into objects and are undefined-preserving.

Every functions is primitive, defined or a functional form

Functions

Identity

$\mathrm{id}{:}x \equiv x$

Atom

$\mathrm{atom}{:}x \equiv x \textit{ is an atom} \longrightarrow T\,;\, x \neq \bot \longrightarrow F\,;\, \bot$

Selector

$1{:}x \equiv x = \langle x_1, \ldots, x_n \rangle \longrightarrow x_1\,;\, \bot$

and for any positive integer s

$s{:}x \equiv x = \langle x_1, \ldots, x_n \rangle \;\&\; n \geq s \longrightarrow x_s\,;\, \bot$

Tail

$\mathrm{tl}{:}x \equiv x = \langle x_1 \rangle \longrightarrow \emptyset\,;$

$x = \langle x_1, \ldots, x_n \rangle \;\&\; n \geq 2 \longrightarrow \langle x_2, \ldots, x_n \rangle\,;\, \bot$

Null

$\mathrm{null}{:}x \equiv x = \emptyset \longrightarrow T\,;\, x \neq \bot \longrightarrow F\,;\, \bot$

Functions

Equality

$$\begin{aligned} eq{:}x \equiv\ & x = \langle y,z\rangle \;\&\; y = z \rightarrow T\,; \\ & x = \langle y,z\rangle \;\&\; y \neq z \rightarrow F\,;\,\perp \end{aligned}$$

Reverse

$$\begin{aligned} reverse{:}x \equiv\ & x = \varnothing \rightarrow \varnothing\,; \\ & x = \langle x_1, \ldots, x_n\rangle \rightarrow \langle x_n, \ldots, x_1\rangle\,;\,\perp \end{aligned}$$

Length

$$\begin{aligned} length{:}x \equiv\ & x = \langle x_1, \ldots, x_n\rangle \rightarrow n\,; \\ & x = \varnothing \rightarrow 0\,;\,\perp \end{aligned}$$

Arithmetic

$+{:}x = \langle y,z\rangle \;\&\; y,z$ *are numbers* $\rightarrow y+z\,;\,\perp$

$-{:}x\ = \langle y,z\rangle \;\&\; y,z$ *are numbers* $\rightarrow y-z\ ;\,\perp$

$\times{:}x = \langle y,z\rangle \;\&\; y,z$ *are numbers* $\rightarrow y\times z\,;\,\perp$

$\div{:}x = \langle y,z\rangle \;\&\; y,z$ *are numbers* $\rightarrow y\div z\,;\,\perp$

Functions

Append

$\mathrm{apndl}:x \equiv x = \langle y, \varnothing \rangle \rightarrow \langle y \rangle ;$

$x = \langle y, \langle z_1, \ldots, z_n \rangle \rangle \rightarrow \langle y, z_1, \ldots, z_n \rangle ; \perp$

$\mathrm{apndr}:x \equiv x = \langle \varnothing, y \rangle \rightarrow y ;$

$x = \langle \langle z_1, \ldots, z_n \rangle, y \rangle \rightarrow \langle z_1, \ldots, z_n, y \rangle ; \perp$

Transpose

$\mathrm{trans}:x \equiv x = \langle \varnothing, \ldots, \varnothing \rangle \rightarrow \langle \varnothing, \ldots, \varnothing \rangle ;$

$x = \langle x_1, \ldots, x_n \rangle \rightarrow \langle y_1, \ldots, y_m \rangle ; \perp$

where

$x_i = \langle x_{i1}, \ldots, x_{im} \rangle$ *and* $y_j = \langle x_{1j}, \ldots, x_{nj} \rangle , 1 \leq i \leq n, 1 \leq j \leq m$

Selector right

$1r:x \equiv x = \langle x_1, \ldots, x_n \rangle \rightarrow x_n ; \perp$

$2r:x \equiv x = \langle x_1, \ldots, x_n \rangle \; n \geq 2 \rightarrow x_{n-1} ; \perp$

etc

Functions

Distribute

$$\mathrm{distl}{:}x \equiv x = <y, \emptyset > \longrightarrow \emptyset \; ;$$

$$x = <y,<z_1, \ldots, z_n>> \longrightarrow <<y, z_1 >, \ldots, <y, z_n >> \; ; \perp$$

$$\mathrm{distr}{:}x \equiv x = <\emptyset,y> \longrightarrow \emptyset \; ;$$

$$x = <<z_1, \ldots, z_n>, y> \longrightarrow <<z_1,y>, \ldots, <z_n ,y>> \; ; \perp$$

Tail right

$$\mathrm{tlr}{:}x \equiv x = <x_1> \longrightarrow \emptyset \; ;$$

$$x = <x_1, \ldots, x_n> \;\&\; n \geq 2 \longrightarrow <x_1, \ldots, x_{n-1}> \; ; \perp$$

Rotate

$$\mathrm{rotl}{:}x \equiv x = \emptyset \longrightarrow \emptyset \; ; x = <x_1> \longrightarrow <x_1> \; ;$$

$$x = <x_1, \ldots, x_n> \;\&\; n \geq 2 \longrightarrow <x_2 , \ldots, x_n, x_1 > \; ; \perp$$

Functional forms

A functional form is an expression denoting a function

Composition

$$(f \circ g):x \equiv f:(g:x)$$

Construction

$$[f_1, \dots, f_n] :x \equiv <f_1:x, \dots, f_n:x>$$

Constant

$$\bar{x} : y \equiv y = \bot \rightarrow \bot; x$$

Condition

$$\begin{aligned}(p \rightarrow f;g): x \equiv {} & (p:x) = T \rightarrow f:x \,; \\ & (p:x) = F \rightarrow g:x \,; \bot\end{aligned}$$

Functional forms

Apply to all

$$\alpha f{:}x \equiv x = \emptyset \rightarrow \emptyset;$$
$$x = \langle x_1, \ldots, x_n \rangle \rightarrow \langle f{:}x_1, \ldots, f{:}x_n \rangle ; \bot$$

Insert

$$/f{:}x \equiv x = \langle x_1 \rangle \rightarrow x_1;$$
$$x = \langle x_1, \ldots, x_n \rangle \;\&\; n \geq 2 \rightarrow f{:}\langle x_1, /f{:}\langle x_2, \ldots, x_n \rangle\rangle ; \bot$$

If f has a unique right unit $u_f \neq \bot$, where $f{:}\langle x,u_f \rangle \in \{x, \bot\}$ for all objects x, then the above definition is extended:

$/f{:}\ \emptyset = u_f$

Definitions

A set of definitions that define some functions in F and assign a name to each

$$\boxed{\text{Def } f \equiv r}$$

Programming examples

Factorial

$$\text{Def } ! \equiv \text{eq}_0 \rightarrow \bar{1};\ \times\circ[\text{id},\ !\circ \text{sub}_1]$$

where

$$\text{Def eq}_0 \equiv \text{eq} \circ[\text{id}, \bar{0}]$$
$$\text{Def sub}_1 \equiv - \circ[\text{id}, \bar{1}]$$

Programming examples

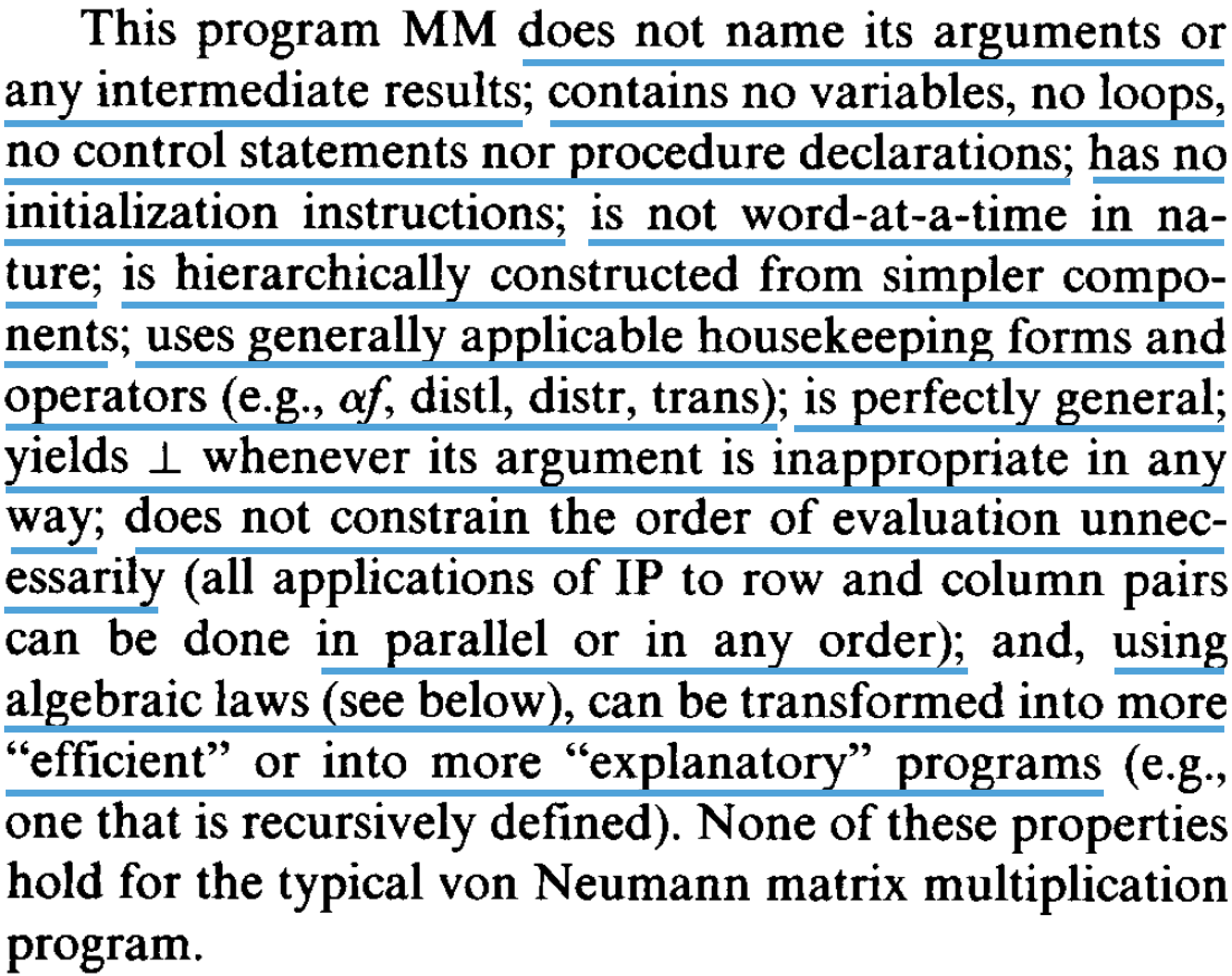
Inner product

$$\text{Def IP} \equiv (/+) \circ (\alpha \times) \circ \text{trans}$$

Matrix multiply

$$\text{Def MM} \equiv (\alpha\ \alpha\ \text{IP}) \circ (\alpha\ \text{distl}) \circ \text{distr} \circ [1, \text{trans} \circ 2]$$

Comparison



This program MM does not name its arguments or any intermediate results; contains no variables, no loops, no control statements nor procedure declarations; has no initialization instructions; is not word-at-a-time in nature; is hierarchically constructed from simpler components; uses generally applicable housekeeping forms and operators (e.g., αf, distl, distr, trans); is perfectly general; yields ⊥ whenever its argument is inappropriate in any way; does not constrain the order of evaluation unnecessarily (all applications of IP to row and column pairs can be done in parallel or in any order); and, using algebraic laws (see below), can be transformed into more "efficient" or into more "explanatory" programs (e.g., one that is recursively defined). None of these properties hold for the typical von Neumann matrix multiplication program.

Conclusions

different approach for problem solving: **what** and not **how**

parallelism opportunities

$$(/+) \circ (\alpha \times) \circ \mathrm{trans}$$

new important properties

what happen if we call twice a function?

what about mutable state?

Thank you!

esteco.com