Association Internationale
pour les Technologies Objets

# The AITO Test of Time Award 2023

is awarded to

## Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, Andrew P. Black

for their work

**Traits: Composable Units of Behaviour**,
ECOOP 2003

Seattle, United States, July 2023

For the Nomination Committee:

Tijs van der Storm
Chairman

For AITO:

Eric Jul
President

# Why Programming Languages Matter: an Improvisation in six languages

Andrew P. Black

Portland State University
Portland, Oregon

Portland State
UNIVERSITY

# Why Programming Languages Matter: an Improvisation in ~~six~~ seven languages

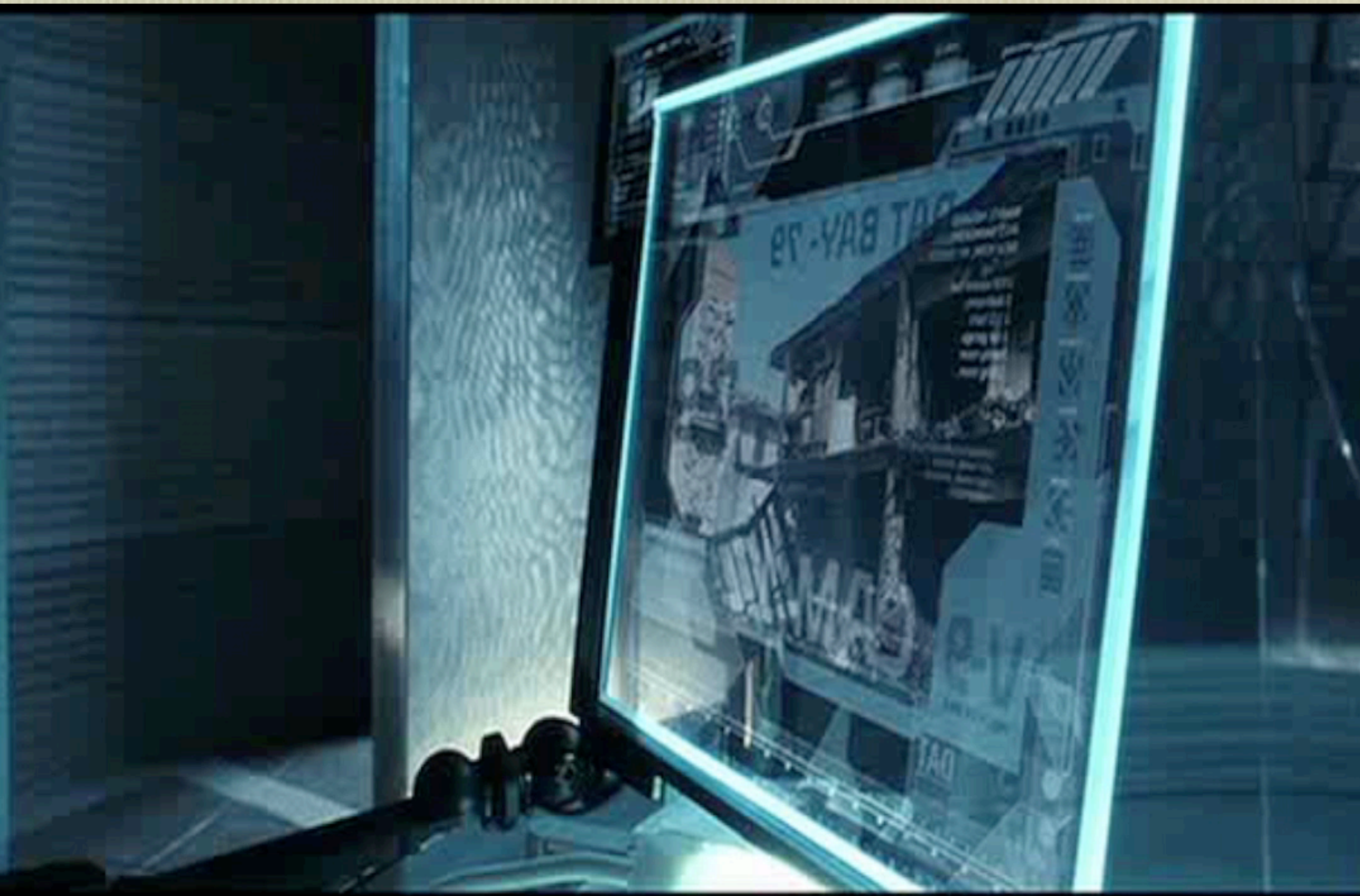
Andrew P. Black

Portland State University
Portland, Oregon

Portland State
UNIVERSITY

# Program Design is Hard

I want to make it easier






Clip from "Minority Report"

# Programming *Language* Design is *Meta*-Hard

Portland State
UNIVERSITY

# Why So?

- A programming language is not just — or even primarily — a means for programmers to communicate with *computers*
- It is also a means for programmers to communicate with *programmers* — including themselves
- It is a *social*, as well as a *technical*, enabler
  - language adoption is slow, like any social change

# Why So?

- A programming language is not just — or even primarily — a means for programmers to communicate with *computers*
- It is also a means for programmers to communicate with *programmers* — including themselves
- It is a *social*, as well as a *technical*, enabler
  - language adoption is slow, like any social change
  - but enjoys the "100th monkey" effect

# Seven Languages

| Language | Years | Place | Customer |
|---|---|---|---|
| **Algol H** | 1977 | UEA | VWRS |
| **3R** | 1977–80 | Oxford | B. Shearing |
| **EPL** | 1982 | UW | Eden Programmers |
| **Emerald** | 1983–6 | UW | Ourselves |
| **Traits** | 2001– | U Bern | Smalltalk Programmers |
| **Fortress** | 2008 | Sun Labs | Engineers |
| **Grace** | 2010– | Cyberspace | Novices |

# 1977: Algol H

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,
C. H. Lindsey, L. G. L. T. Meertens and
R. G. Fisker

Springer-Verlag
Berlin·Heidelberg·New York



APIC Studies in Data Processing No. 8

# STRUCTURED PROGRAMMING

O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare

Academic Press
London New York San Francisco
A Subsidiary of Harcourt Brace Jovanovich, Publishers

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,
C. H. Lindsey, L. G. L. T. Meertens and
R. G. Fisker

Springer-Verlag
Berlin·Heidelberg·New York



APIC Studies in Data Processing No. 8

## II. Notes on Data Structuring *

C. A. R. Hoare

### 1. Introduction

In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on these similarities, and to ignore for the time being the differences. As soon as we have discovered which similarities are relevant to the prediction and control of future events, we will tend to regard the similarities as fundamental and the differences as trivial. We may then be said to have developed an abstract concept to cover the set of objects or situations in question. At this stage, we will usually introduce a word or picture to symbolise the abstract concept; and any particular spoken or written occurrence of the word or picture may be used to *represent* a particular or general instance of the corresponding situation.

The primary use for representations is to convey information about important aspects of the real world to others, and to record this information in written form, partly as an aid to memory and partly to pass it on to future generations. However, in primitive societies the representations were sometimes believed to be useful in their own right, because it was supposed that manipulation of representations might in itself cause corresponding changes in the real world; and thus we hear of such practices as sticking pins into wax models of enemies in order to cause pain to the corresponding part of the real person. This type of activity is characteristic of magic and witchcraft. The modern scientist on the other hand, believes that the manipulation of representations could be used to predict events and the results of changes in the real world, although not to cause them. For example, by manipulation of symbolic representations of certain functions and equations,

*This monograph is based on a series of lectures delivered at a Nato Summer School, Marktoberdorf, 1970.

- Algol 68: good

- Algol 68: good
- Hoare's Structured Data: + good

- Algol 68: good
- Hoare's Structured Data: + good
- Algol 68 + Hoare's Structured Data:

- Algol 68: good
- Hoare's Structured Data: + good
- Algol 68 + Hoare's Structured Data: + + good

- Algol 68: good
- Hoare's Structured Data: + good
- Algol 68 + Hoare's Structured Data:

- Algol 68: good
- Hoare's Structured Data: + good
- Algol 68 + Hoare's Structured Data:

> "a closing of the gap between the data structures of the program and the real-world objects they represent."

A. P. Black and V. Rayward-Smith. Proposals for Algol H — a superlanguage of Algol 68. Algol Bulletin, 42:36–49, May 1978.

# Lessons:

- Consolidation is harder than innovation
  - Mostly, Hoare's data and Algol 68 meshed well
    - Both inspired by Algol 60
  - The exception: tagged and untagged unions
- If you have a destination in mind, be careful from where you start

# Recommended Reading

- C. H. Lindsey. A history of Algol 68. In *History of Programming Languages—II*, pages 27–96. Association for Computing Machinery, New York, NY, USA, 1996.

“2.3.4.1 ***Parameter Passing***

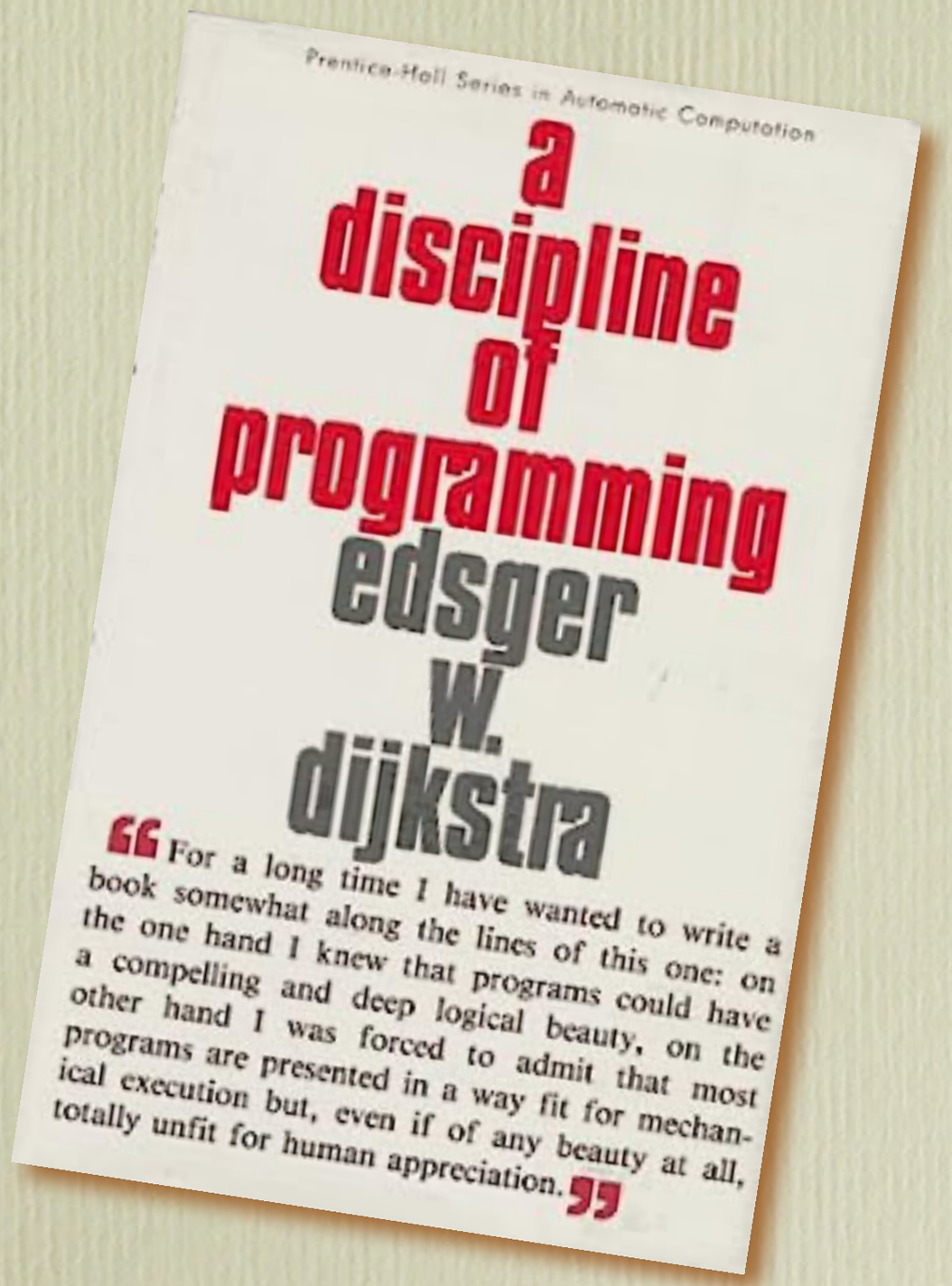
It is said that an Irishman, when asked how to get to some remote place, answered that if you really wanted to get to that place, then you shouldn’t start from here. In trying to find an acceptable parameter-passing mechanism, WG 2.1 started from ALGOL 60 …”

# 1978–80: 3R

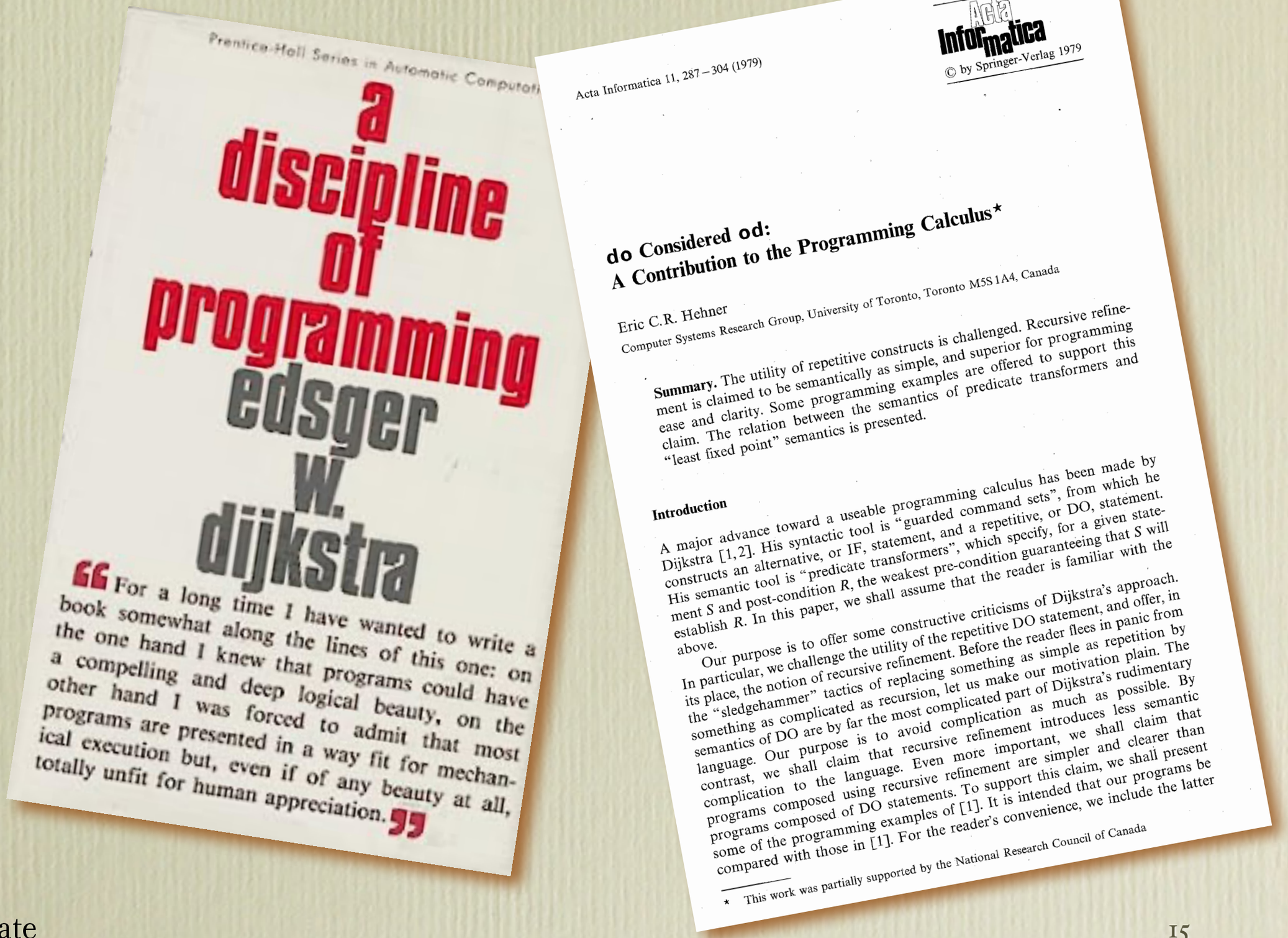
Portland State
UNIVERSITY

- "Reading, 'riteing, and 'rithmetic"
- Programming language designed for *readability*
  - Names made up of multiple words
- Flat (no nesting): *Blocks* and *Blocklets*
  - Blocks (procedures) can have (multiple) arguments, *e.g.*, delete [i]th line of page[p]
  - Blocklets have no arguments
- No loops !
  - named code fragments

# Influences


Prentice-Hall Series in Automatic Computation
a discipline of programming
edsger w. dijkstra
"For a long time I have wanted to write a book somewhat along the lines of this one: on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation."

# Influences


Prentice-Hall Series in Automatic Computati
a discipline of programming
edsger w. dijkstra
“For a long time I have wanted to write a book somewhat along the lines of this one: on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation.”
Acta Informatica 11, 287–304 (1979)
Acta Informatica
© by Springer-Verlag 1979
do Considered od:
A Contribution to the Programming Calculus*
Eric C.R. Hehner
Computer Systems Research Group, University of Toronto, Toronto M5S 1A4, Canada
Summary. The utility of repetitive constructs is challenged. Recursive refinement is claimed to be semantically as simple, and superior for programming ease and clarity. Some programming examples are offered to support this claim. The relation between the semantics of predicate transformers and “least fixed point” semantics is presented.
Introduction
A major advance toward a useable programming calculus has been made by Dijkstra [1,2]. His syntactic tool is “guarded command sets”, from which he constructs an alternative, or IF, statement, and a repetitive, or DO, statement. His semantic tool is “predicate transformers”, which specify, for a given statement S and post-condition R, the weakest pre-condition guaranteeing that S will establish R. In this paper, we shall assume that the reader is familiar with the above.
Our purpose is to offer some constructive criticisms of Dijkstra’s approach. In particular, we challenge the utility of the repetitive DO statement, and offer, in its place, the notion of recursive refinement. Before the reader flees in panic from the “sledgehammer” tactics of replacing something as simple as repetition by something as complicated as recursion, let us make our motivation plain. The semantics of DO are by far the most complicated part of Dijkstra’s rudimentary language. Our purpose is to avoid complication as much as possible. By contrast, we shall claim that recursive refinement introduces less semantic complication to the language. Even more important, we shall claim that programs composed using recursive refinement are simpler and clearer than programs composed of DO statements. To support this claim, we shall present some of the programming examples of [1]. It is intended that our programs be compared with those in [1]. For the reader’s convenience, we include the latter
* This work was partially supported by the National Research Council of Canada

# Influences

# Influences

- Brian Shearing
  - *knew* that he needed a language
  - contracted to produce a description of an algorithm that was both *readable* and executable

# Influences

- Brian Shearing
  - *knew* that he needed a language
  - contracted to produce a description of an algorithm that was both *readable* and executable
- Algol 60, Cobol?

# Influences

- Brian Shearing
  - *knew* that he needed a language
  - contracted to produce a description of an algorithm that was both *readable* and executable
- Algol 60, Cobol?
- Tony Hoare:
  - Simplify, simplify, simplify until it hurts. Then simplify some more.

# Recommended Reading

- E. Hehner. do considered od: A contribution to the programming calculus. Acta Informatica, 11(4):287– 304, 1979.
- Dijkstra's Language of Guarded Commands

**if**
[] $guard_1 \rightarrow stmt_1$
[] $guard_2 \rightarrow stmt_2$
**fi**

**do**
[] $guard_1 \rightarrow stmt_1$
[] $guard_2 \rightarrow stmt_2$
**od**

# Recommended Reading

- E. Hehner. do considered od: A contribution to the programming calculus. Acta Informatica, 11(4):287– 304, 1979.
- Dijkstra's Language of Guarded Commands

**if**
[] $guard_1 \rightarrow stmt_1$
[] $guard_2 \rightarrow stmt_2$
**fi**

**do**
[] $guard_1 \rightarrow stmt_1$
[] $guard_2 \rightarrow stmt_2$
**od**

Execute one of the stmts whose guard is true.
If there is none, **abort**

# Recommended Reading

- E. Hehner. do considered od: A contribution to the programming calculus. Acta Informatica, 11(4):287– 304, 1979.
- Dijkstra's Language of Guarded Commands

**if**
[] $guard_1 \rightarrow stmt_1$
[] $guard_2 \rightarrow stmt_2$
**fi**

**do**
[] $guard_1 \rightarrow stmt_1$
[] $guard_2 \rightarrow stmt_2$
**od**

Execute one of the stmts whose guard is true,
*and then execute the whole do..od again*
If there is none, **skip**

# Recommended Reading

- E. Hehner. do considered od: A contribution to the programming calculus. Acta Informatica, 11(4):287– 304, 1979.
- Dijkstra's Language of Guarded Commands

**if**
[] $guard_1 \rightarrow stmt_1$
[] $guard_2 \rightarrow stmt_2$
**fi**

**do**
[] $guard_1 \rightarrow stmt_1$
[] $guard_2 \rightarrow stmt_2$
**od**

- Program development by stepwise refinement
  - descriptive names are later elaborated into code

## 4.5. Scanning One Word

This block scans the current line and returns the next word or perhaps a null string if one is not found. A word is a letter followed by zero or more letters, digits, or underscore characters.

```
LET New Word := Get One Word BE
  USES Current Character
  RESULT New Word IS TEXT
  INVARIABLE Underscore Character IS '_'
  New Word := ''
  Remove Front Blanks
  IF (Current Character >= 'a' AND Current Character <= 'z') OR ...
    (Current Character >= 'A' AND Current Character <= 'Z')
    New Word := New Word + Current Character
    Get Next Character
    Add Characters Until Delimiter
  IF NOT (...
    (Current Character >= 'a' AND Current Character <= 'z') OR ...
    (Current Character >= 'A' AND Current Character <= 'Z'))
    PASS
OTHERWISE CHAOS

WHERE Add Characters Until Delimiter IS
  IF (Current Character >= 'a' AND Current Character <= 'z') OR ...
    (Current Character >= 'A' AND Current Character <= 'A') OR ...
    (Current Character >= '0' AND Current Character <= '9') OR ...
    (Current Character = Underscore Character)
    New Word := New Word + Current Character
    Get Next Character
    Add Characters Until Delimiter
  IF NOT ( ...
    (Current Character >= 'a' AND Current Character <= 'z') OR ...
```

```
USES Current Character
RESULT New Word IS TEXT
INVARIABLE Underscore Character IS '_'
New Word := ''
Remove Front Blanks
IF (Current Character >= 'a' AND Current Character <= 'z') OR ...
  (Current Character >= 'A' AND Current Character <= 'Z')
  New Word := New Word + Current Character
  Get Next Character
  Add Characters Until Delimiter
IF NOT (...
  (Current Character >= 'a' AND Current Character <= 'z') OR ...
  (Current Character >= 'A' AND Current Character <= 'Z'))
  PASS
OTHERWISE CHAOS

WHERE Add Characters Until Delimiter IS
  IF (Current Character >= 'a' AND Current Character <= 'z') OR ...
    (Current Character >= 'A' AND Current Character <= 'A') OR ...
    (Current Character >= '0' AND Current Character <= '9') OR ...
    (Current Character = Underscore Character)
    New Word := New Word + Current Character
    Get Next Character
    Add Characters Until Delimiter
  IF NOT ( ...
    (Current Character >= 'a' AND Current Character <= 'z') OR ...
    (Current Character >= 'A' AND Current Character <= 'Z') OR ...
    (Current Character >= '0' AND Current Character <= '9') OR ...
    (Current Character = Underscore Character))
    PASS
  OTHERWISE CHAOS
END OF BLOCK { new word := get one word }
```

```
USES Current Character
RESULT New Word IS TEXT
INVARIABLE Underscore Character IS '_'
New Word := ''
Remove Front Blanks
IF (Current Character >= 'a' AND Current Character <= 'z') OR ...
  (Current Character >= 'A' AND Current Character <= 'Z')
  New Word := New Word + Current Character
  Get Next Character
  Add Characters Until Delimiter
IF NOT (...
  (Current Character >= 'a' AND Current Character <= 'z') OR ...
  (Current Character >= 'A' AND Current Character <= 'Z'))
  PASS
OTHERWISE CHAOS

WHERE Add Characters Until Delimiter IS
  IF (Current Character >= 'a' AND Current Character <= 'z') OR ...
    (Current Character >= 'A' AND Current Character <= 'A') OR ...
    (Current Character >= '0' AND Current Character <= '9') OR ...
    (Current Character = Underscore Character)
    New Word := New Word + Current Character
    Get Next Character
    Add Characters Until Delimiter
  IF NOT (...
    (Current Character >= 'a' AND Current Character <= 'z') OR ...
    (Current Character >= 'A' AND Current Character <= 'Z') OR ...
    (Current Character >= '0' AND Current Character <= '9') OR ...
    (Current Character = Underscore Character))
    PASS
  OTHERWISE CHAOS
END OF BLOCK { new word := get one word }
```

# Language as a Simplifier

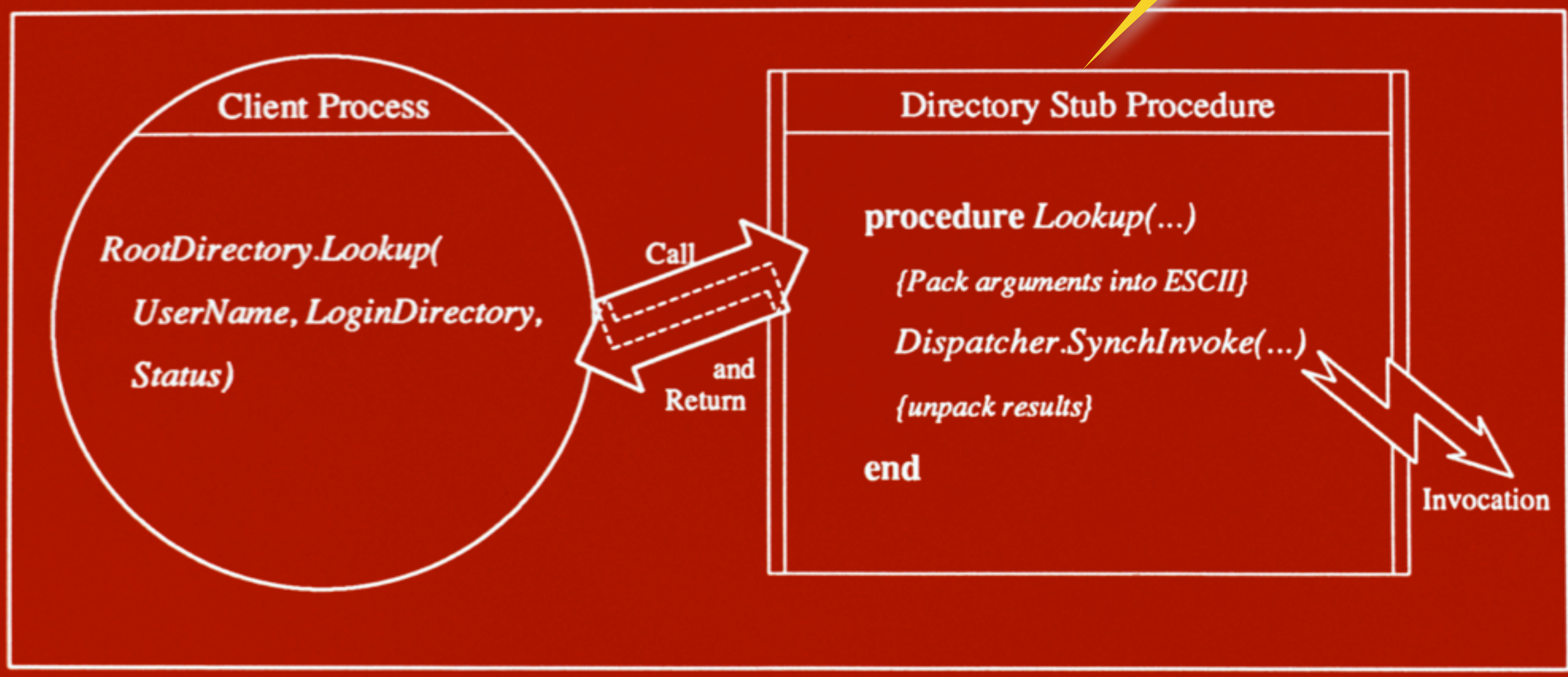
# 1982–1984: Eden Programming Language

Portland State
UNIVERSITY

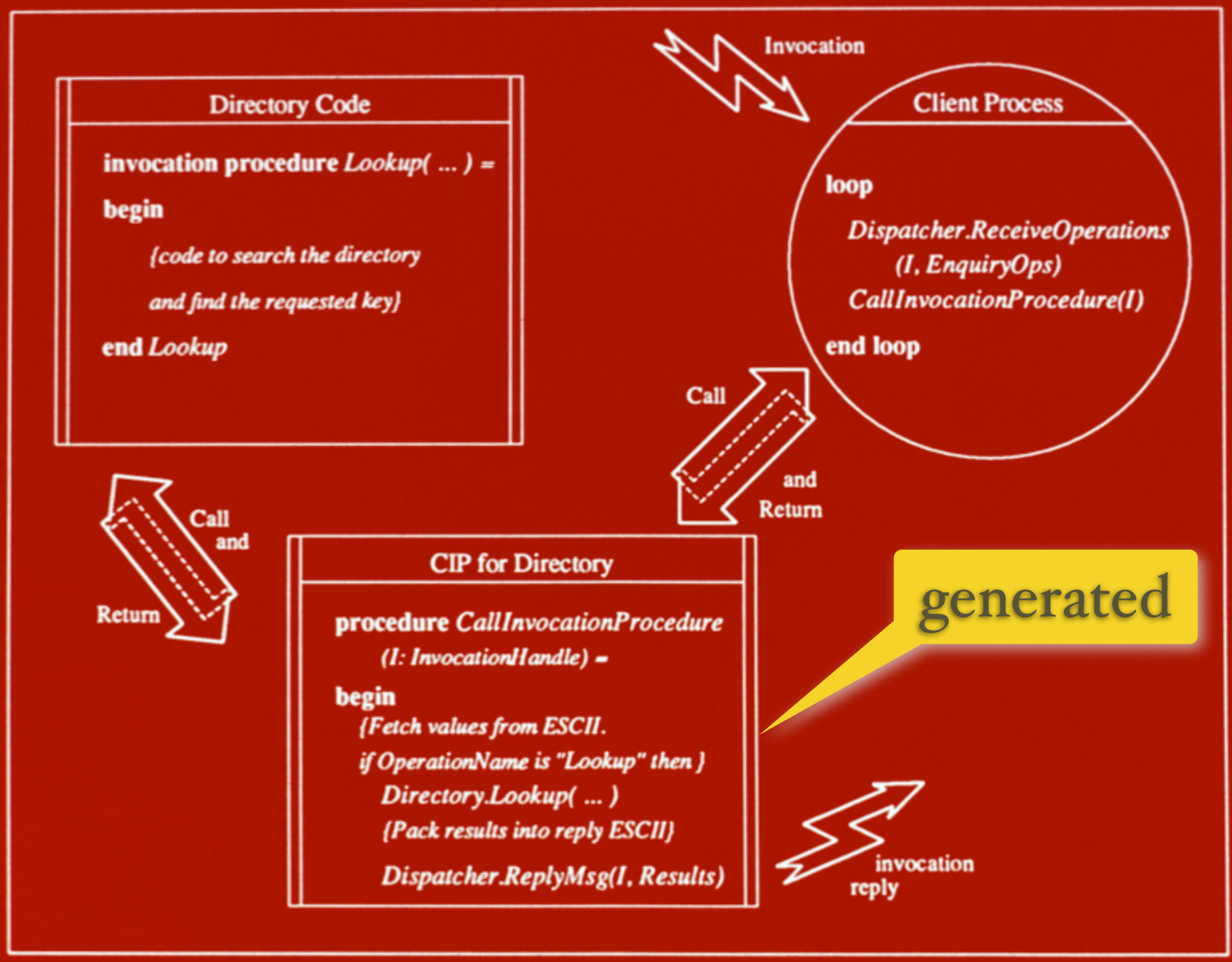
- Eden Project (1980–1984) — early attempt to build a “distributed, integrated” computing system.
- EPL implemented by translation into Concurrent Euclid (CE)
- EPL provided:
  - synchronous (local or remote) object invocation
  - concurrency inside Eden objects
  - capabilities to address objects
  - strings (because CE didn’t!)

# Sending an Invocation

generated



# Receiving an Invocation



# Reflections

- Eden saw itself as *distributed systems* research
  - no one on the project knew that they needed a programming language!
- In hindsight, EPL was essential:
  - it hid the messy, boring stuff (marshaling, dispatch), and
  - freed programers to focus on the interesting and hard stuff (algorithms, concurrency)

# 1983–87: Emerald

Portland State
UNIVERSITY

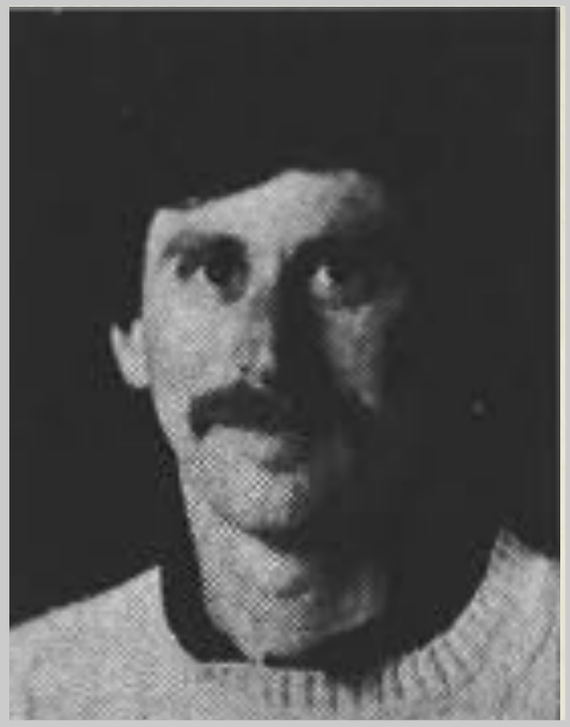
# The People

Andrew Black

Norm Hutchinson

Eric Jul

Henry (Hank) Levy



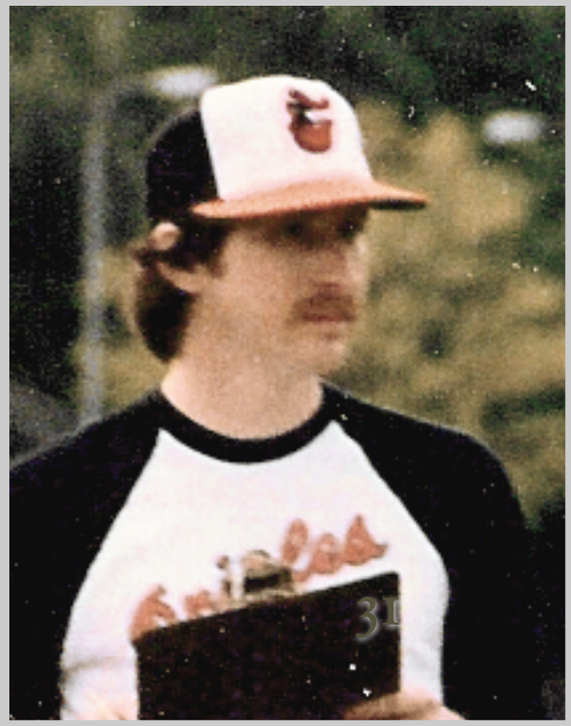
# The People

Andrew Black

Norm Hutchinson

Eric Jul

Henry (Hank) Levy



# The People

## Andrew Black

Exception Handling

## Norm Hutchinson

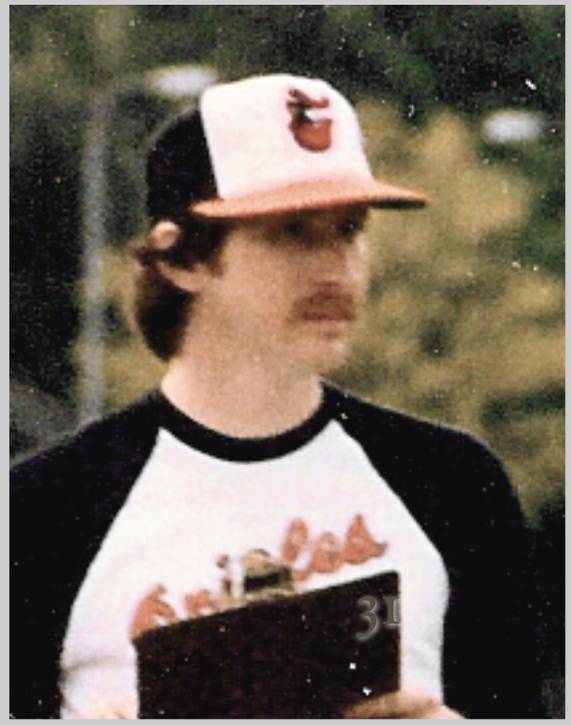
Simula 67

## Eric Jul

Simula 67, Concurrent Pascal

## Henry (Hank) Levy

Capability architectures, systems



# Emerald

- Addressed building a distributed system as a *language* problem
- Separated "semantics" from "locatics"
  - Local and remote objects had same semantics: "Location-independent invocation"
- Compiled code about as efficient as C in local case,
  - and 100 x faster than Eden in the remote case

# Emerald Language Features

- Innovations:
  - Object constructors
    - mutable & immutable objects
  - Failure handling
  - Parameterized types
- Conventional:
  - Objects had processes (as in Simula)
  - Hoare monitors for synchronization
- Simplifications:
  - No classes, no inheritance

# Reflections

- Emerald was about 20 years before its time
  - NSF called it “unimplementable”
  - Still generating dissertations in 2023

# Almost wasn't Published

#90 "Fine-Grained Mobility in the Emerald System"

Referee's Report

This is a straightforward implementation of a simple idea. It is hard to see what is unique about this operating system.

# Almost wasn't Published

#90 "Fine-Grained Mobility in the Emerald System"

Referee's Report

This is a straightforward implementation of a simple idea. It is hard to see what is unique about this operating system.

My most influential paper (over 1200 citations)

# Reflections

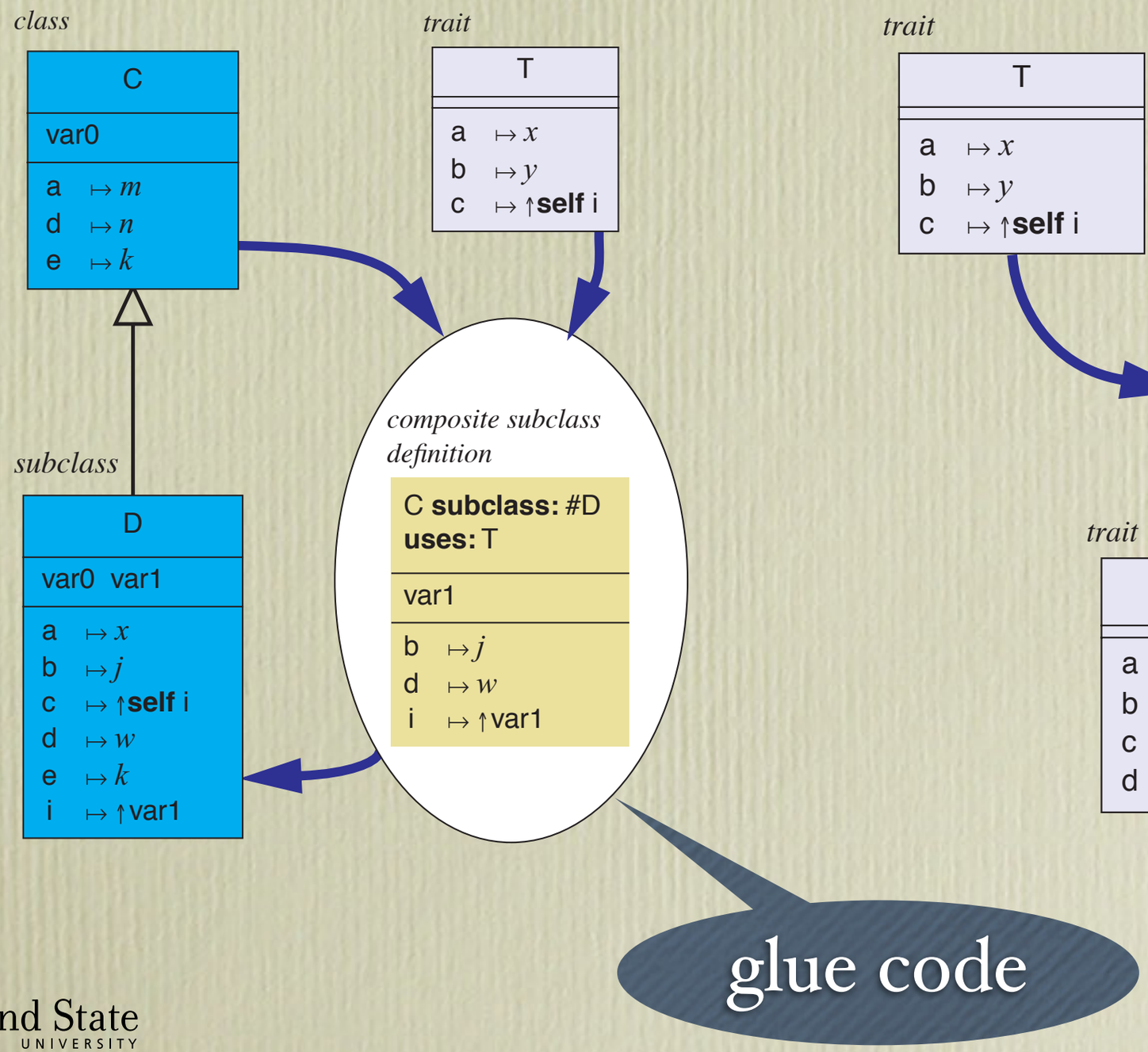
- Not widely used, but widely influential
  - ANSA DPL, OMG CORBA, INRIA's *Guide*, Birrell et al.'s Network Objects, the ANSI Smalltalk standard, Java RMI
- We were our own customers. *We* realized that we needed a language ...
  - Dramatic simplification of the programmer's world (compared to Eden)
  - Freed programmers to think about the hard problems: object location, and concurrency.

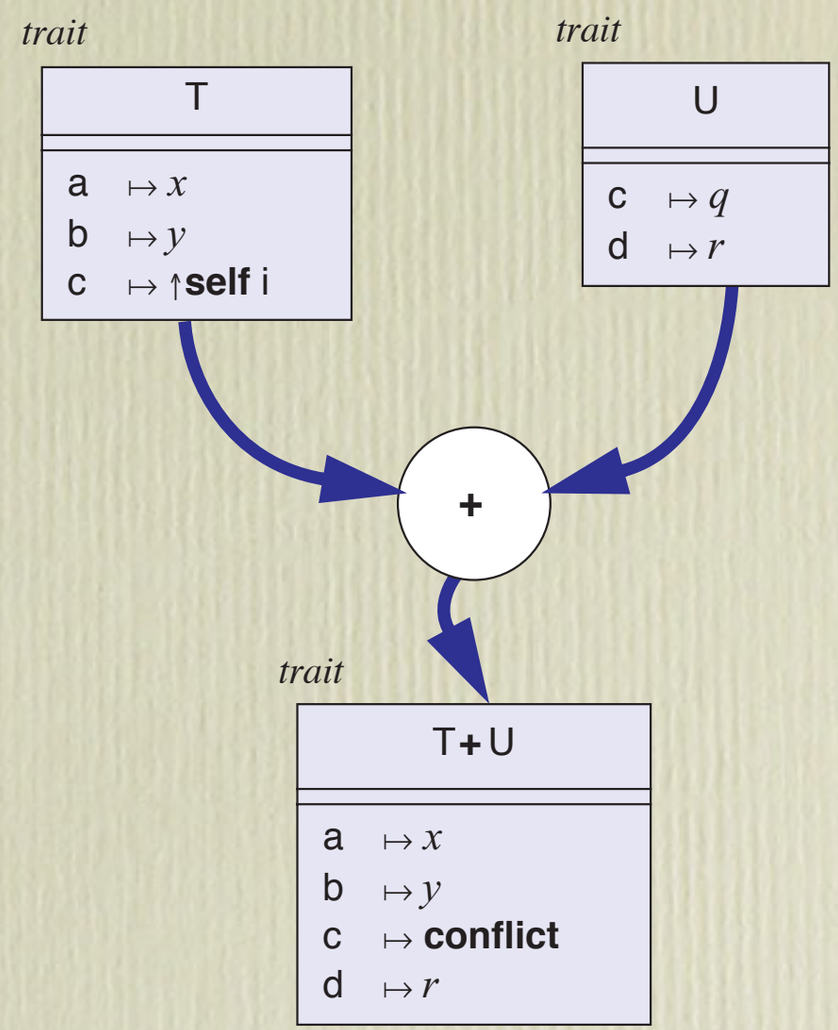
# 2001–present: Traits

- Traits: a language feature, not a language
- a *Trait* is a Smalltalk class without any instance variables
- Traits can be
  - combined with +,
  - modified with @ (alias) and – (exclusion)
  - *used* in other traits and classes.

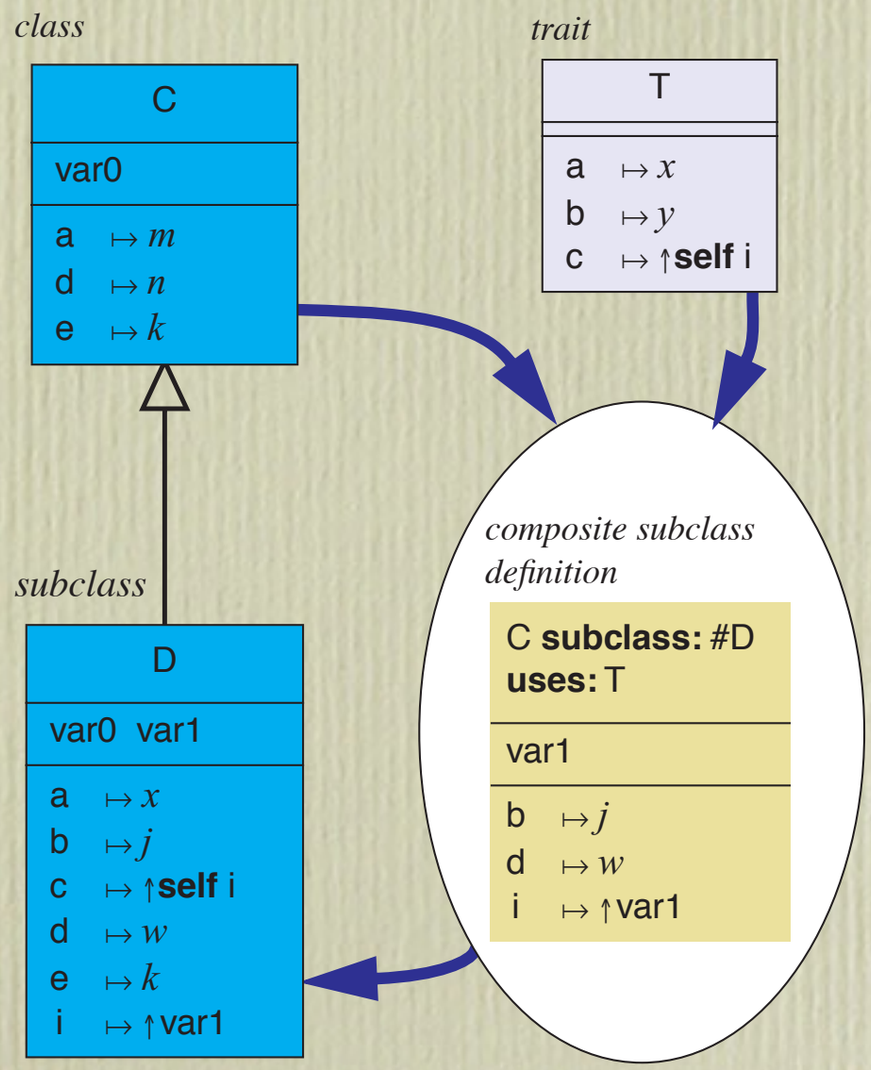
- Trait = set of methods, without instance vars

- *Sum*, *alias*, *exclude* and *uses* as combinators

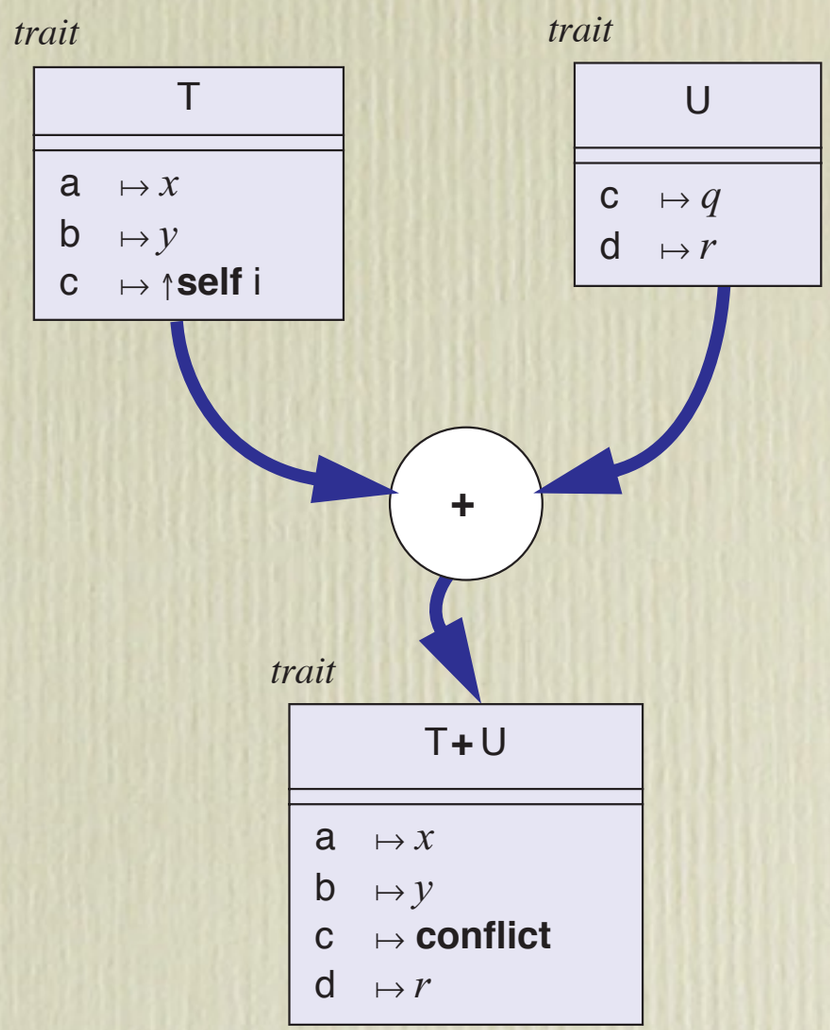


- Trait = set of methods, without instance vars




- *Sum*, *alias*, *exclude* and *uses* as combinators



# Influences

- Deep experience with Smalltalk
- The sad history of multiple inheritance

> "multiple inheritance is good, but there is no good way to do it"
>
> *Steve Cook* channeling *Alan Snyder*

- Nathanael Schärli, who cut the gordian knot
- A little lattice theory
- Excellent toolbuilding environment & skills

# Reflections

- *Smallest* contribution
- Largest impact?
  - Pearl 6, Java, Pharo, Visualworks, Fortress, Racket, Ruby, C#, Scala, Joose, PHP, ActionScript, ...
- We underestimated the importance of programming tools
  - many of the properties we claimed for traits depended also on tool support

# Simplicity?

# Simplicity?

- Did traits simplify the Smalltalk Language?

# Simplicity?

- Did traits simplify the Smalltalk Language?
  - No! They made it significantly more complicated!

# Simplicity?

- Did traits simplify the Smalltalk Language?
  - No!  They made it significantly more complicated!
  - But they removed complexity, redundancy, and misleading structure from the Smalltalk system

# Simplicity?

- Did traits simplify the Smalltalk Language?
  - No!  They made it significantly more complicated!
  - But they removed complexity, redundancy, and misleading structure from the Smalltalk system
- The overall programming system was simpler

# Simplicity?

- Did traits simplify the Smalltalk Language?
  - No!  They made it significantly more complicated!
  - But they removed complexity, redundancy, and misleading structure from the Smalltalk system
- The overall programming system was simpler
- Traits would be simpler without, *e.g.*, –

# Simplicity?

- Did traits simplify the Smalltalk Language?
  - No!  They made it significantly more complicated!
  - But they removed complexity, redundancy, and misleading structure from the Smalltalk system
- The overall programming system was simpler
- Traits would be simpler without, *e.g.*, –
  - but programming with traits would be harder

# Simplicity?

- Did traits simplify the Smalltalk Language?
  - No! They made it significantly more complicated!
  - But they removed complexity, redundancy, and misleading structure from the Smalltalk system
- The overall programming system was simpler
- Traits would be simpler without, *e.g.*, –
  - but programming with traits would be harder
  - A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection classes. In OOPSLA'03, pp 47–64

# Recommended Reading

- R. P. Gabriel. The structure of a programming language revolution. In Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2012, pages 195–214.

> “The real paradigm shift? Systems versus languages.”

# Programming Systems & Complexity

- Programming System:
  - Language + Libraries + Tools + project code
- A new feature adds complexity ...
  - which must be paid for by removing more complexity from the system as a whole
- “Feature Debt”

# 2008: Fortress

- Large language: aimed to displace Fortran
- Large team (by academic standards):
  - Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele Jr., plus visitors (me) and interns
- Support for mathematical notation
  - Parsing depends on type inference, is space-sensitive, and context dependent
  - Extensible: new syntax, with semantics defined in libraries

# Mathematical Notation

- Math notation is familiar, but *not* simple

# Mathematical Notation

- Math notation is familiar, but *not* simple
  - We spend 15 or more years in school learning it

$$3x \sin x \cos 2x \log \log x$$

## Juxtaposition

$3x \sin x \cos 2x \log \log x$

## Juxtaposition

$3x \sin x \cos 2x \log \log x$

## Juxtaposition

$3x \sin x \cos 2x \log \log x$

## Juxtaposition

$3x \sin x \cos 2x \log \log x$

$x$: Number

sin, cos, log: Number → Number

$3x \sin x \cos 2x \log \log x$

$x$: Number
sin, cos, log: Number → Number

juxtaposition-operator

$3x \sin x \cos 2x \log \log x$

FunctionApplication

$x$: Number

sin, cos, log: Number → Number

$$\{\, |x| \mid x \leftarrow mySet,\ 3|x \,\}$$

S. Ryu. Parsing Fortress syntax. In Proc. 7th Int. Conf. Principles and Practice of Programming in Java, PPPJ '09, pages 76–84, New York, NY, USA, 2009. Association for Computing Machinery.

$$\{\, |x| \mid x \leftarrow mySet,\ 3|x \,\}$$

enclosing operator

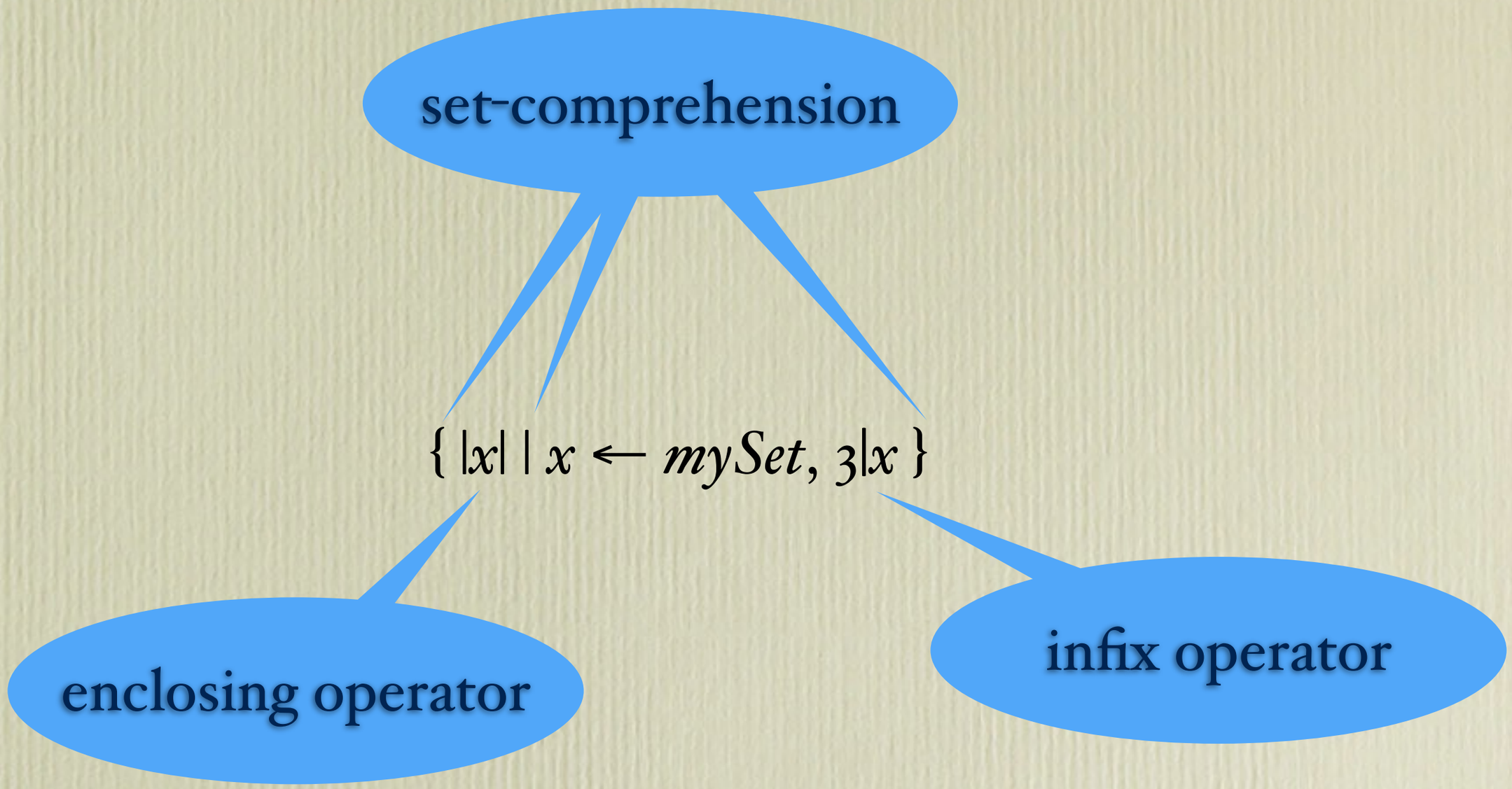
S. Ryu. Parsing Fortress syntax. In Proc. 7th Int. Conf. Principles and Practice of Programming in Java, PPPJ '09, pages 76–84, New York, NY, USA, 2009. Association for Computing Machinery.

$$\{\, |x| \mid x \leftarrow mySet,\ 3|x \,\}$$

enclosing operator

infix operator

S. Ryu. Parsing Fortress syntax. In Proc. 7th Int. Conf. Principles and Practice of Programming in Java, PPPJ '09, pages 76–84, New York, NY, USA, 2009. Association for Computing Machinery.




S. Ryu. Parsing Fortress syntax. In Proc. 7th Int. Conf. Principles and Practice of Programming in Java, PPPJ '09, pages 76–84, New York, NY, USA, 2009. Association for Computing Machinery.

# 2010–present: Grace

Portland State
UNIVERSITY

# Grace

- Simple O-O language for teaching
  - block-structured
  - dialects, realized as enclosing modules
  - optional, gradual types
  - indentation matters
- An effort at *consolidation*, not *innovation*
- Open-source implementation

# Reflections

- The *consumer* is a novice student
  - but the *customer* is an instructor in a introductory programming course
- ~~Surprisingly~~ challenging to please both
  - e.g., clean object model *or* existing practice?
- Design skills ⇄ implementation skills
  - The first language where I was the prime implementor

# Is Grace Simple?

- Simpler than Java, Python, C++, ...
  - But not as simple as it might have been
- Like Fortress, we mistook familiarity for simplicity

# Operator Precedence

- The operators * and / have higher precedence than + and -
  - because in arithmetic, multiplication & division have precedence over addition & subtraction.
  - precedence is independent of the methods that *, /, +, and - may cause to be executed
- Smalltalk is simpler: left to right execution

# Traits and Classes

- Grace has both Traits and Classes
  - Classes, because we wanted a form of inheritance familiar to instructors
    - We did eliminate *super-requests*
  - Traits, because single class inheritance was inadequate for building our own libraries
- I believe (now) that we could have devised a traits-only mechanism that was both simpler and more powerful than our hybrid

# What keeps me coming back?

- I like *fixing things*
  - there's plenty to fix in programming!
- I like *helping others* to succeed
  - Programming languages are an *enabler*
    - for others (3R, EPL)
    - for programmers (Traits)
    - for students (*Grace*)

# Why Do PLs Matter?

# Why Do PLs Matter?

A quick survey of the members of IFIP WG 2.16 on language design ...

# Creating

"The power to create out of pure thought"

*Jonathan Edwards*

"In the beginning was the word"

*Cristina Lopes*

*Michelangelo: The Creation of Adam*

# Magic

Programmers are like wizards ... except that the magic is real!

PLs are “spell systems”

*Sean McDirmid*

“Any sufficiently-advanced technology is indistinguishable from magic”

*Arthur C . Clarke*

# Foundational

- Software is the most important infrastructure for … basically everything
- Software is totally dependent on programming languages
- Hence: programming languages are the most important infrastructure for anything and everything!

*James Noble*

# Are we there yet?

# Are we there yet? No!

# Are we there yet? No!

Since Fortran, people have been saying that we don't need new languages.

Yet, languages continue to evolve ... and few of us would want to go back to Fortran.

*Roberto Ierusalimschy*

# Language as “Law Enforcement”

# Language as "Law Enforcement"

The value of a language can be in what it *prevents* you from doing

Hence: libraries are not the answer

# Language as "Law Enforcement"

The value of a language can be in what it *prevents* you from doing

## Hence: libraries are not the answer

- No library is ever going to ensure that there are no race conditions in my Java program

# Languages shape thought

# Languages shape thought

## Whorfianism, or “Linguistic Relativity”

# Languages shape thought

Whorfianism, or “Linguistic Relativity”

Learning a new language “changes the path of least resistance”

*Tom van Cutsem*

# Languages shape thought

Whorfianism, or “Linguistic Relativity”

Learning a new language “changes the path of least resistance”

*Tom van Cutsem*

# Languages shape thought

# Languages shape thought

# Languages shape thought

"You can't trust the opinions of others, because of the Blub paradox: they're satisfied with whatever language they happen to use, because it dictates the way they think about programs."

*Paul Graham*

# Languages shape thought

# Languages shape thought

# Languages shape thought

# Languages shape thought

"A language that doesn't affect the way you think about programming, is not worth knowing"

*Alan Perlis*

# Languages shape thought

# Languages shape thought

# Languages shape thought

"A programming system has two parts. The programming 'environment' is the part that's installed on the computer. The programming 'language' is the part that's installed in the programmer's head."

*Brett Victor*

# Languages shape thought

# Languages shape thought

My Recommendation:

# Languages shape thought

My Recommendation:

- *Do* program in a pure functional language
- *Do* program with pure objects (Smalltalk)

# Languages shape thought

My Recommendation:

- *Do* program in a pure functional language
- *Do* program with pure objects (Smalltalk)
- *Do* program with CSP

# Languages shape thought

My Recommendation:

- *Do* program in a pure functional language
- *Do* program with pure objects (Smalltalk)
- *Do* program with CSP
- *Do* try Logic Programming (but not Prolog!)

# Languages shape thought

My Recommendation:

- *Do* program in a pure functional language
- *Do* program with pure objects (Smalltalk)
- *Do* program with CSP
- *Do* try Logic Programming (but not Prolog!)

Use them for a serious project

# PL Reading List

1. *Notation as a tool of thought*. Iverson
2. *Programming as Theory-building*. Naur
3. *Beating the Averages*. Graham (and commentary thereon at c2.com)
4. *The Development of the Emerald Programming Language*. Black *et al*. HoPL III
5. *Algol 60 Report*. Naur *et al*
6. *Smalltalk*. **BYTE** Magazine, August 1981
7. *Lisp: Good News, Bad News, How to Win Big*. Gabriel
8. *Babel-17*. Delany
9. *An exploration of program as language*. Baniassad and Myers

# References

A. P. Black and V. Rayward-Smith. Proposals for Algol H — a superlanguage of Algol 68. Algol Bulletin, 42:36–49, May 1978.

E. W. Dijkstra. A Discipline of Programming. Prentice Hall, 1976.

E. Hehner. do considered od: A contribution to the programming calculus. Acta Informatica, 11(4):287– 304, 1979.

N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In L. Cardelli (ed), ECOOP, LNCS vol 2743, pages 248–274, Darmstadt, Germany, 2003.