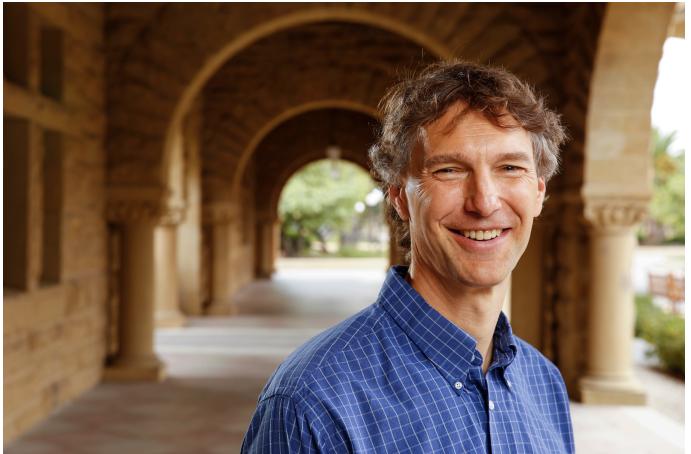


Programming Languages

CS242

Lecture 1

Course Staff



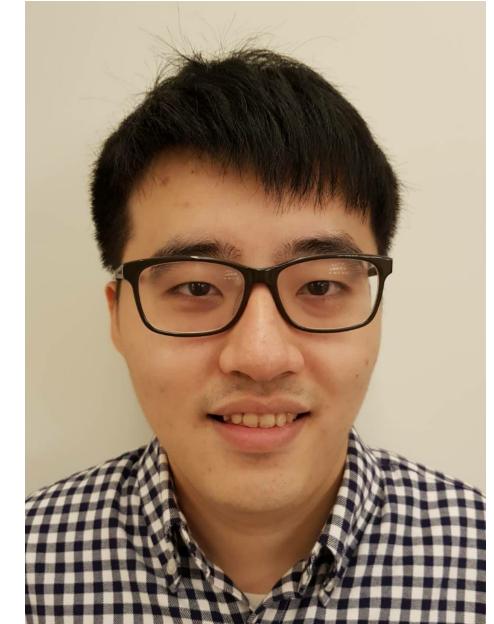
Alex Aiken



Wonyeol Lee



Jason Koenig



Cheng-Min Chiang

CS242: Programming Languages

- All things programming languages!
- Different families of programming languages
 - Including some you may have not seen before
 - In theory and practice
- Core ideas
 - Semantics
 - Type systems
 - Program analysis
 - Formal verification
 - Program synthesis
- Core features
 - Continuations
 - Monads
 - Concurrency
 - Parallelism

Course Prerequisites

- Theory: CS 103
 - First-order logic, induction, discrete math
- Systems: CS 107 + 110
- Comfortable with programming
 - And learning new programming systems

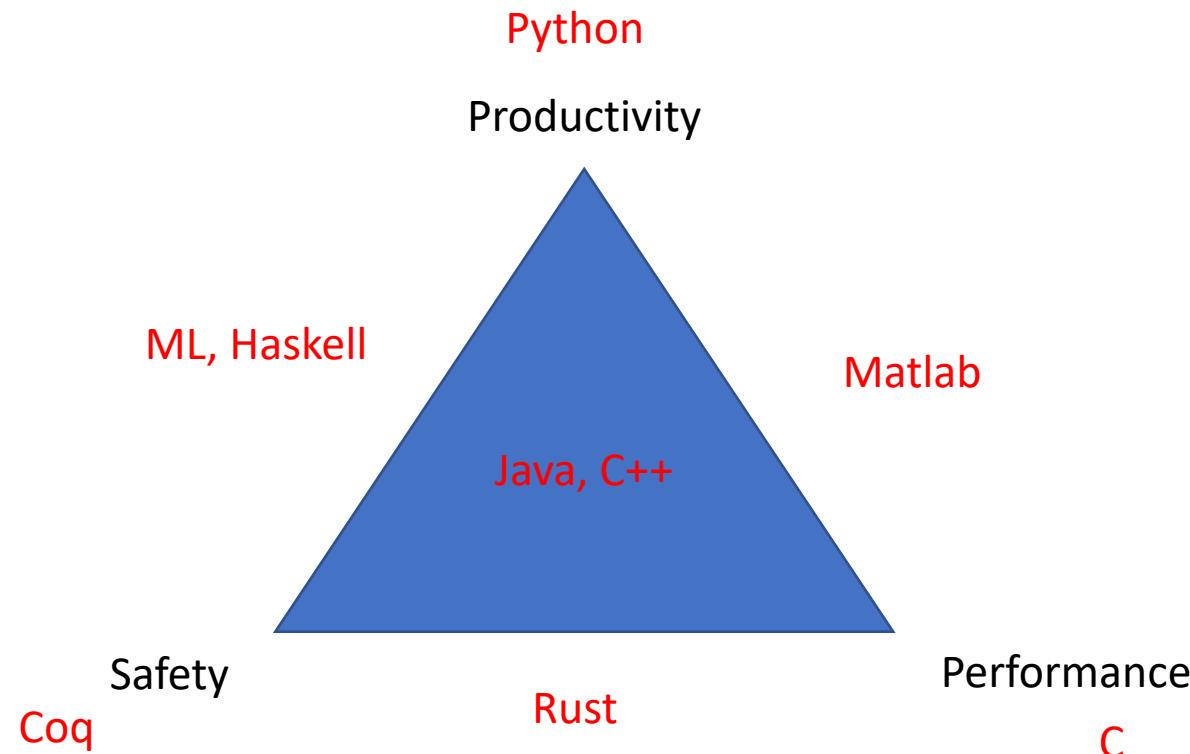
Course Structure

- Homeworks
 - Programming assignments
 - Roughly weekly, starting second week (50%)
 - Expect ~10 hours/week
- Readings
- Course project (50%)
 - Choose from a list of possible projects
 - TBD ...
- No exams

Takeaways

- Appreciation for programming languages as a technical field
 - The problems, the values, the “culture”
- Becoming a better programmer
 - Learn to think systematically about programming tools
- Understand the future
 - What we can do, what we can’t do, what we will likely be able to do
- Basics of active research areas
 - Preparation for research
- Thanks to Will Crichton for many of the slides in this lecture!

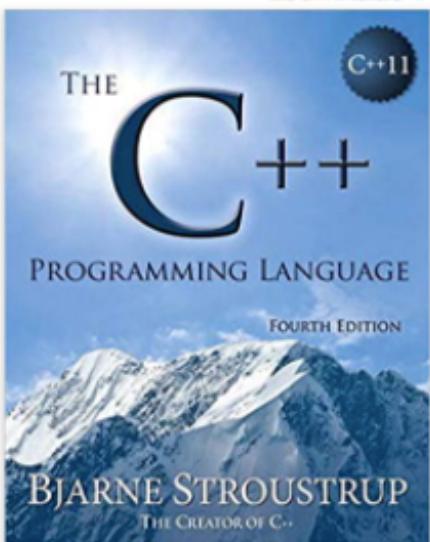
The Big Picture: Language Goals



Language Goals

- Every programming language has as goals
 - Performance
 - Productivity
 - Safety
- But there are tradeoffs
- And different designs make different choices
 - One of the reasons we have so many programming languages

Programming Languages are Complex Tools



Product details

Publisher : Addison-Wesley Professional; 4th edition (May 1, 2012)

Language: English

Paperback : 1376 pages

Item Weight : 2.73 pounds



Product details

Publisher : Wiley; 1st edition (July 8, 2014)

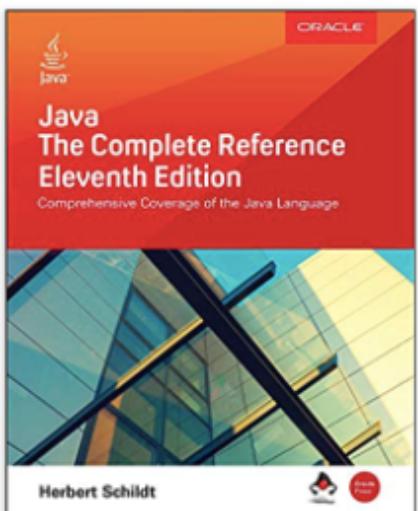
Language: English

Paperback : 1152 pages

ISBN-10 : 1118907442

ISBN-13 : 978-1118907443

Item Weight : 6.04 pounds



Publisher : McGraw-Hill Education;

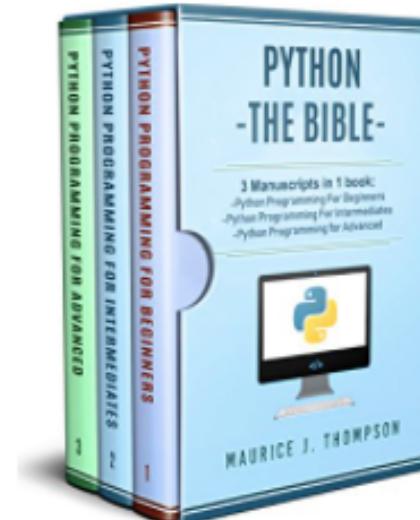
Language: English

Paperback : 1248 pages

ISBN-10 : 1260440230

ISBN-13 : 978-1260440232

Item Weight : 3.86 pounds



Product details

ASIN : B07CQPHC1N

Publication date : April 27, 2018

Language: English

File size : 70333 KB

Text-to-Speech : Enabled

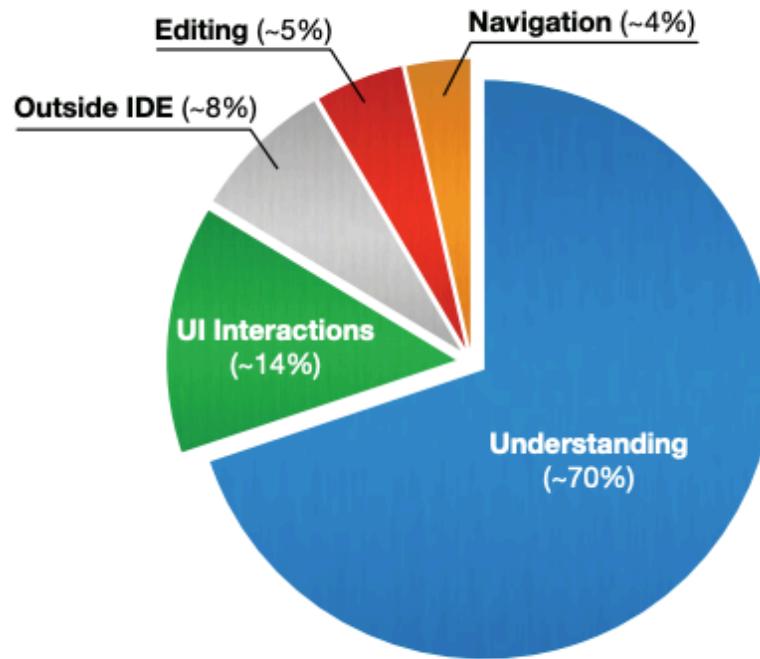
Enhanced typesetting : Enabled

X-Ray : Not Enabled

Word Wise : Not Enabled

Print length : 377 pages

Developers Are Reading, Not Writing



Minelli et al. "I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time" ICPC '15.

Project	Comprehension	Navigation	Editing	Others
Average	57.62%	23.96%	5.02%	13.40%

Xia et al. "Measuring Program Comprehension: A Large-Scale Field Study with Professionals." IEEE Trans. Softw. Eng, 2018.

Complexity



std::move_if_noexcept

Defined in header `<utility>`

```
template< class T >
typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept(T& x) noexcept;
template< class T >
constexpr typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept(T& x) noexcept;
```

(since C++11)
(until C++14)

(since C++14)

`move_if_noexcept` obtains an rvalue reference to its argument if its move constructor does not throw exceptions or if there is no copy constructor (move-only type), otherwise obtains an lvalue reference to its argument. It is typically used to combine move semantics with strong exception guarantee.

Parameters

`x` - the object to be moved or copied

Return value

`std::move(x)` or `x`, depending on exception guarantees.

Notes

This is used, for example, by `std::vector::resize`, which may have to allocate new storage and then move or copy elements from old storage to new storage. If an exception occurs during this operation, `std::vector::resize` undoes everything it did to this point, which is only possible if `std::move_if_noexcept` was used to decide whether to use move construction or copy construction. (unless copy constructor is not available, in which case move constructor is used either way and the strong exception guarantee may be waived)

Example

[Run this code](#)

```
#include <iostream>
#include <utility>
```

6.7.3.1 Formal definition of `restrict`

- 1 Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a `restrict`-qualified pointer to type **T**.
- 2 If **D** appears inside a block and does not have storage class **extern**, let **B** denote the block. If **D** appears in the list of parameter declarations of a function definition, let **B** denote the associated block. Otherwise, let **B** denote the block of **main** (or the block of whatever function is called at program startup in a freestanding environment).
- 3 In what follows, a pointer expression **E** is said to be *based* on object **P** if (at some sequence point in the execution of **B** prior to the evaluation of **E**) modifying **P** to point to a copy of the array object into which it formerly pointed would change the value of **E**.¹¹⁹⁾ Note that “based” is defined only for expressions with pointer types.
- 4 During each execution of **B**, let **L** be any lvalue that has `&L` based on **P**. If **L** is used to access the value of the object **X** that it designates, and **X** is also modified (by any means), then the following requirements apply: **T** shall not be const-qualified. Every other lvalue used to access the value of **X** shall also have its address based on **P**. Every access that modifies **X** shall be considered also to modify **P**, for the purposes of this subclause. If **P** is assigned the value of a pointer expression **E** that is based on another restricted pointer object **P2**, associated with block **B2**, then either the execution of **B2** shall begin before the execution of **B**, or the execution of **B2** shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.
- 5 Here an execution of **B** means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration

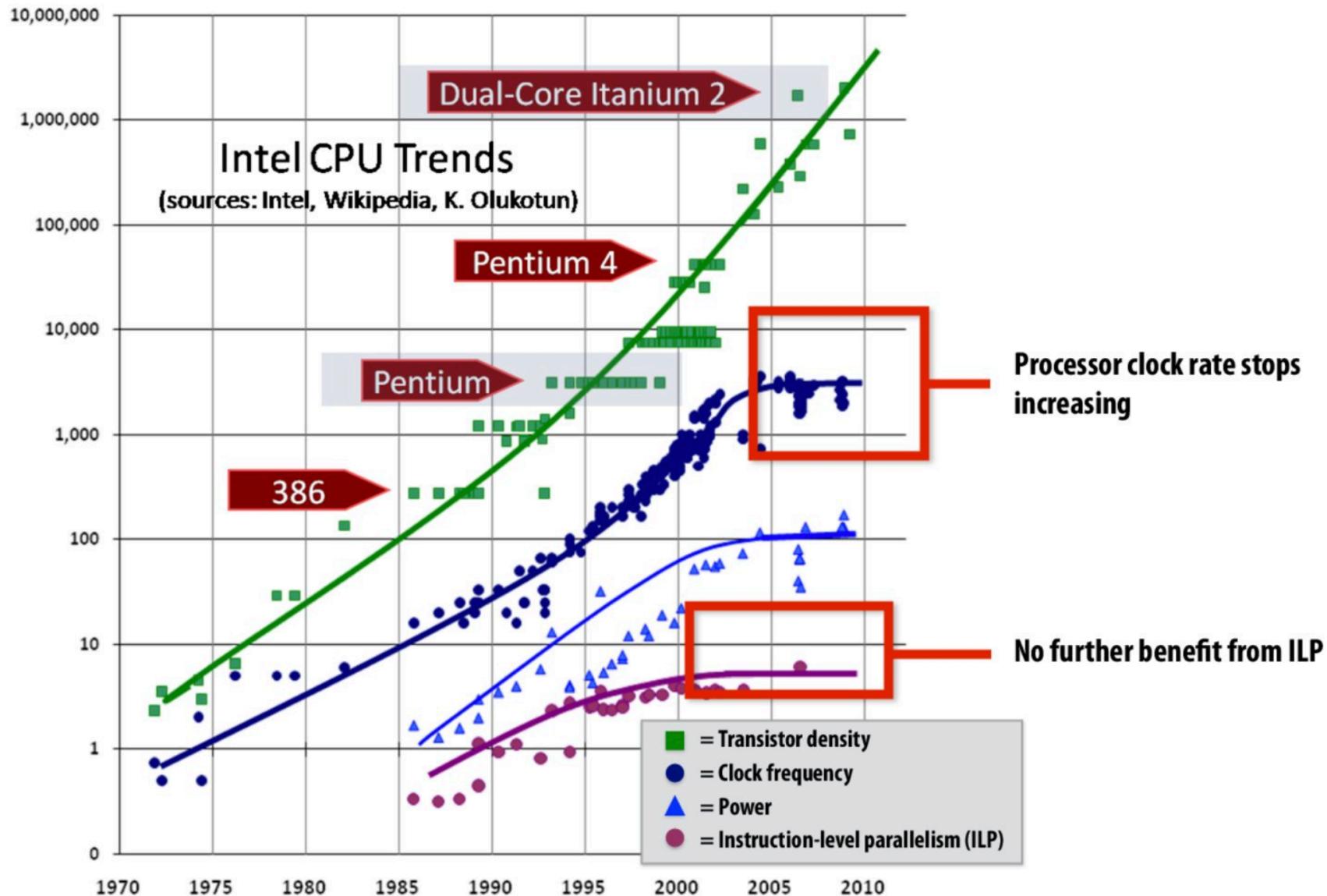


Image credit: "The free Lunch is Over" by Herb Sutter, Dr. Dobbs 2005

CMU 15-418/618, Spring 2017

Memory usage continues to scale

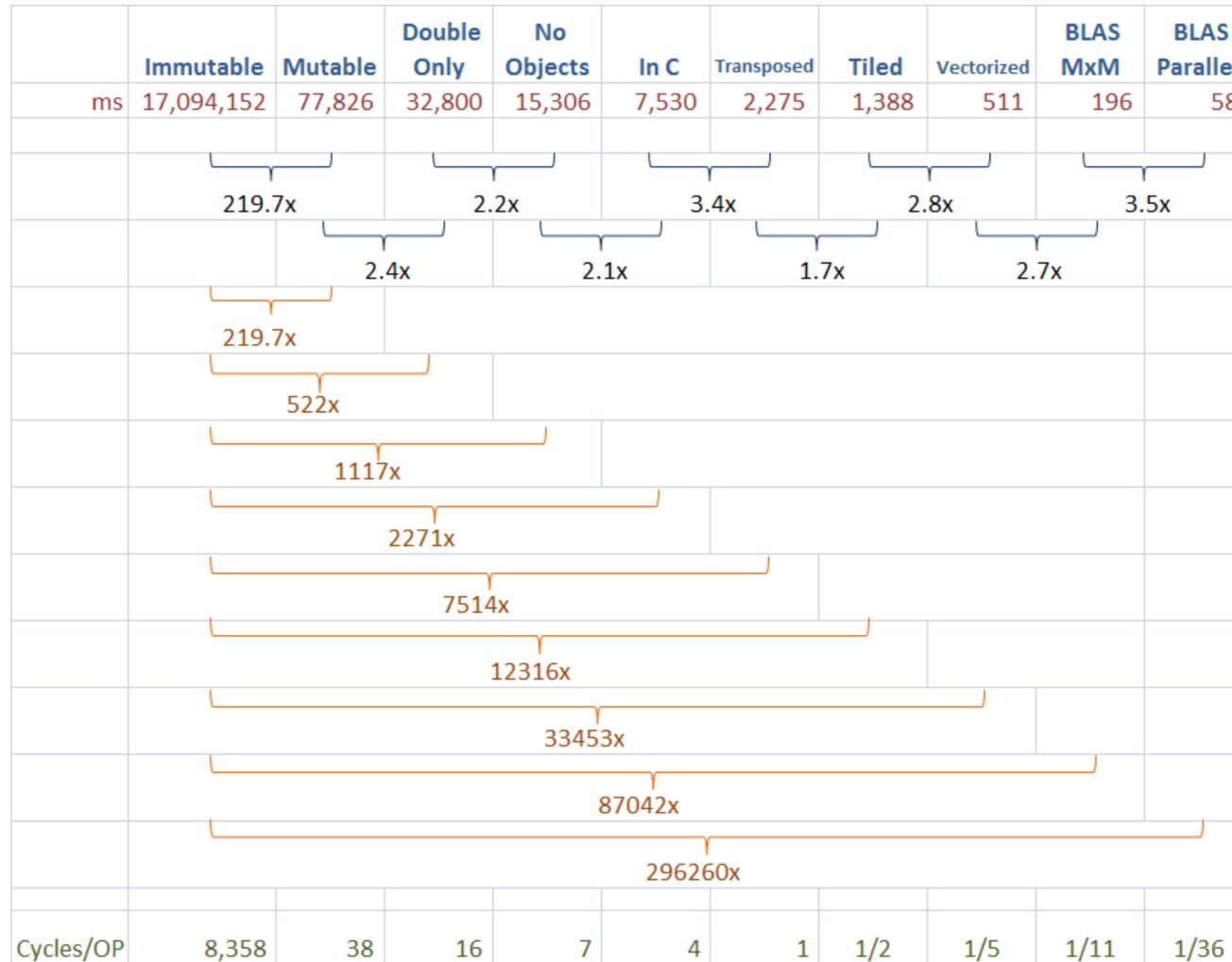
Process Name	Memory
Slack Helper	365.8 MB
Slack Helper	234.9 MB
Slack	91.8 MB
Slack Helper	89.8 MB

700 MB!

Process Name	Memory	Co
Google Chrome Helper	738.8 MB	
Google Chrome Helper	419.8 MB	
Google Chrome Helper	399.4 MB	
Google Chrome Helper	366.4 MB	
Google Chrome	317.8 MB	
Google Chrome Helper	57.2 MB	
Google Chrome Helper	54.9 MB	
Google Chrome Helper	53.5 MB	
Google Chrome Helper	45.2 MB	
Google Chrome Helper	44.5 MB	
Google Chrome Helper	44.0 MB	
Google Chrome Helper	43.8 MB	
Google Chrome Helper	43.6 MB	
Google Chrome Helper	43.5 MB	
Google Chrome Helper	43.3 MB	
Google Chrome Helper	43.2 MB	
Google Chrome Helper	43.1 MB	
Google Chrome Helper	42.8 MB	
Google Chrome Helper	40.0 MB	
Google Chrome Helper	30.1 MB	
Google Chrome Helper	18.8 MB	
Google Chrome Helper	18.4 MB	
Google Chrome Helper	17.3 MB	

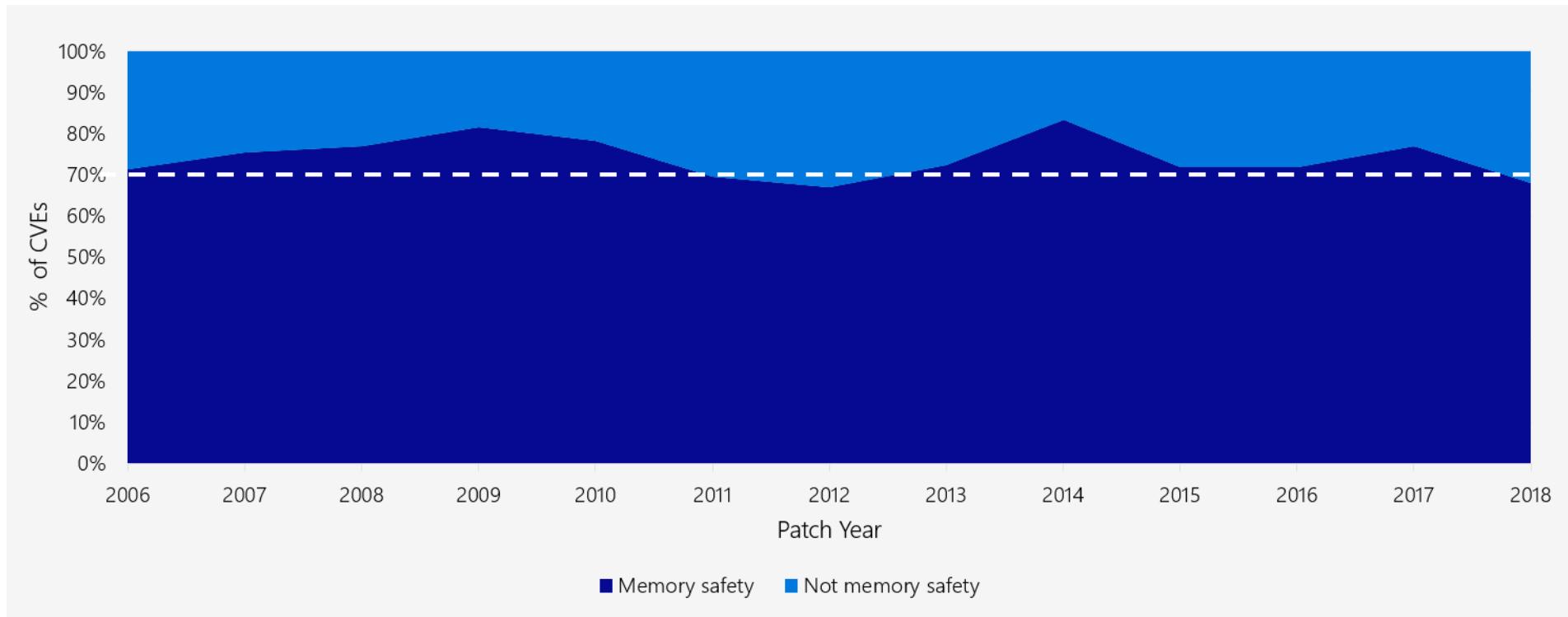
3 GB!!!

Huge Performance Gaps



And Persistent Correctness Gaps

“The majority of vulnerabilities fixed and with a CVE assigned are caused by developers inadvertently inserting memory corruption bugs into their C and C++ code.”



Recap

1. Developers mostly comprehend/debug code
2. Software ecosystems are complex at all scales
3. The hardware is changing and software often doesn't exploit it well
4. Bugs aren't getting any better

The Long View

Alan Turing



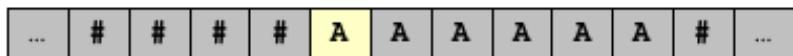
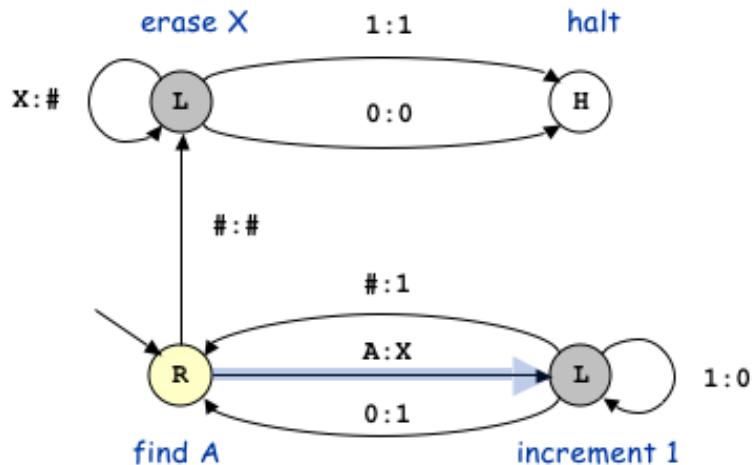
“On Computable
Numbers” 1937

Alonzo Church



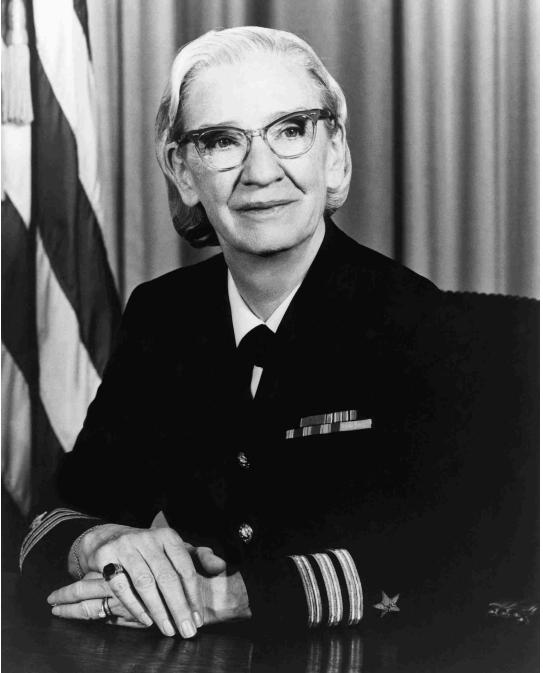
“A set of postulates for
the foundation of logic”,
1932

Turing machines



Lambda calculus

$$\begin{array}{ll} [x \rightarrow y] x & = y \\ [x \rightarrow y] z & = z \\ [x \rightarrow y] \lambda z . x & = \lambda z . y \\ [x \rightarrow y] \lambda y . x y & = \lambda y' . y y' \\ [x \rightarrow y] x (\lambda x . x) & = y (\lambda x . x) \end{array}$$



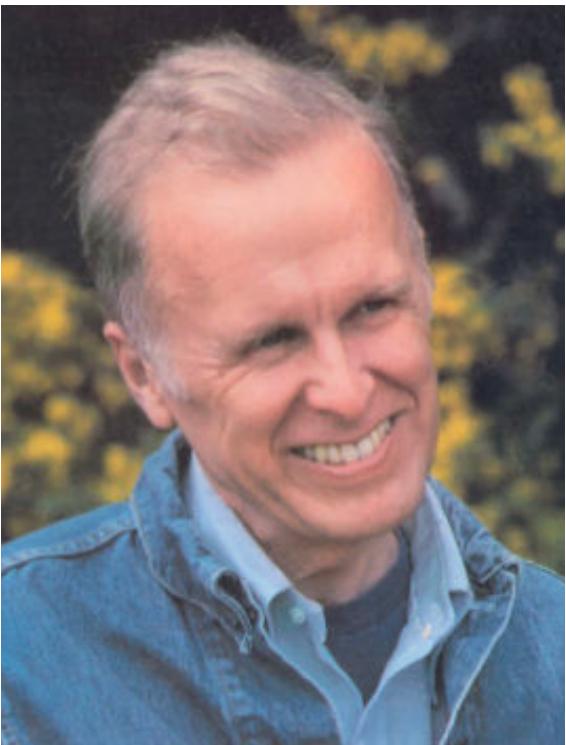
Grace Hopper
HOPL Keynote, 1978

On assembly:

I think I spent 20 years fighting the “Establishment.” In the early years of programming languages, the most frequent phrase we heard was that the only way to program a computer was in octal. Of course, a few years later a few people admitted that maybe you could use assembly language.

On compilers:

At that time, the Establishment told us that a computer could not write a program; it was totally impossible; that all that computers could do was arithmetic; that it had none of the imagination and dexterity of a human being. I kept trying to explain that we were wrapping up the human being's dexterity in the program that he wrote.



John Backus,
“History of FORTRAN”
1978

Prior to FORTRAN, most source language operations were not machine operations. Large inefficiencies in looping and computing addresses were masked by time spent in floating point subroutines.

The advent of the 704 with built-in floating point and indexing radically altered the situation. ...It increased the problem of generating efficient programs by an order of magnitude by speeding up floating point operations by a factor of ten and thereby leaving inefficiencies nowhere to hide. This caused us to regard the design of the translator as the real challenge, not the simple task of designing the language.



Alan Perlis,
“The American Side of the
Development of Algol”
1978

Algol introduced into programming languages such terms as type, declaration, identifier, for statement, while, if then else, switch, the begin end delimiters, block, call by value and call by name, typed procedures, declaration scope, dynamic arrays, side effects, global and local variables.

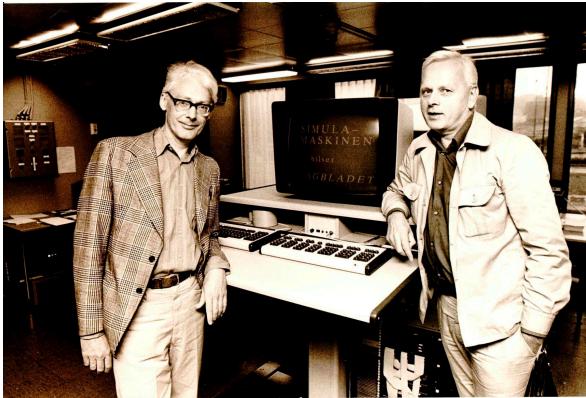
Algol was strongly derived from FORTRAN and its contemporaries. The logic, arithmetic and data organizations were close to those then being designed into real computers. Certain simple generalizations of computer instructions such as switch, for statement, and if statements were included because their semantics and computer processing were straight-forward consequences of single statement processing.



Ken Thompson,
Dennis Ritchie
“The Development
of the C Language”
1993

C fits firmly in the traditional procedural family typified by Fortran and Algol 60. It is ‘close to the machine’ in that the abstractions it introduces are readily grounded in the concrete data types and operations supplied by conventional computers.

The most important [historical accident] has been the tolerance of C compilers to errors in type. C evolved from typeless languages. It did not suddenly appear to its earliest users as an entirely new language with its own rules; instead we continually had to adapt existing programs as the language developed, and make allowance for an existing body of code.



Kristen Nygaard and Ole-Johan Dahl
“The Development of SIMULA Languages”, 1978

Our first main task was to carry out resonance absorption calculations related to the construction of Norway's first nuclear reactor. Monte Carlo simulation methods were successfully introduced instead in 1949-1950. The necessity of using simulation and the need of a language for system description was the direct stimulus for SIMULA.

When writing simulation programs we had observed that processes often shared a number of common properties, both in data attributes and actions, but were structurally different in other respects so that they had to be described by separate declarations. Such partial similarity fairly often applied to processes in different simulation models, indicating that programming effort could be saved by somehow preprogramming the common properties.



Bjarne Stroustrup
“A History of C++:
1979-1991”, 1993

[Simula’s] class concept allowed me to map my application concepts into the language constructs in a direct way. The way Simula classes can act as coroutines made the inherent concurrency of my application easy to express.

The implementation of Simula, however, did not scale in the same way and as a result the whole project came close to disaster. The cost arose from several language features and their interactions: run-time type checking, guaranteed initialization of variables, concurrency support, and garbage collection of both user-allocated objects and procedure activation records.



Sun Microsystems,
1997

The team originally considered using C++, but rejected it for several reasons. They decided that C++'s complexity led to developer errors. The language's lack of garbage collection meant that programmers had to manually manage system memory, a challenging and error-prone task. The team also worried about the C++ language's lack of portable facilities for security, distributed programming, and threading. Finally, they wanted a platform that would port easily to all types of devices.

— Wikipedia



Guido van Rossum
“The Making of
Python”, 2003

My initial goal for Python was to serve as a second language for people who were C or C++ programmers, but who had work where writing a C program was just not effective.

Maybe it was something you'd do only once. It was the sort of thing you'd prefer to write a shell script for, but when you got into the writing details, you found that the shell was not the ideal language—you needed more data structures, more namespaces, or maybe more performance. The first sound bite I had for Python was, "Bridge the gap between the shell and C."

Turing languages

1957 — FORTRAN

1959 — ALGOL

1962 — SIMULA

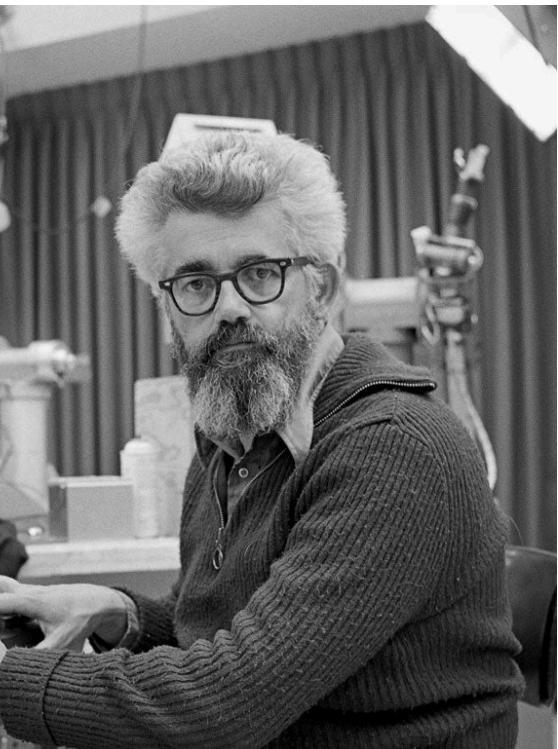
1972 — C

1979 — C++

1991 — Python

1995 — Java





John McCarthy,
“History of LISP” 1978

My own research in artificial intelligence [in 1958]... involved representing information about the world by sentences in a suitable formal language and a reasoning program that would decide what to do by making logical inferences. Representing sentences by list structure seemed appropriate and a list-processing language also seemed appropriate for programming the operations involved in deduction.

...One needs a notation for functions, and it seemed natural to use the λ -notation of Church (1941). I didn't understand the rest of his book, so I wasn't tempted to try to implement his more general mechanism for defining functions.



Peter Landin,
“The Next 700
Programming
Languages” 1966

The languages people use to communicate with computers differ in their intended aptitudes, towards either a particular application area, or a particular phase of computer use (high level programming, program assembly, job scheduling, etc). The question arises, do the idiosyncrasies reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments?

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ -notation."



Robin Milner,
“A Metalanguage for
Interactive Proof in
LCF” 1978

The principal aims in designing ML were to make it impossible to prove non-theorems yet easy to program strategies for performing proofs.

A strategy—or recipe for proof—could be something like “induction on f and g , followed by assuming antecedents and doing case analysis, all interleaved with simplification”. This is imprecise—analysis of what cases? what kind of induction, etc, etc.—but these in turn may well be given by further recipes, still in the same style.



Hudak et al.
“A History of Haskell:
Being Lazy with
Class” 2007

The simplicity and elegance of functional programming captivated the present authors. Lazy evaluation— with its direct connection to the pure, call-by-name lambda calculus, the remarkable possibility of representing and manipulating infinite data structures, and addictively simple and beautiful implementation techniques—was like a drug.

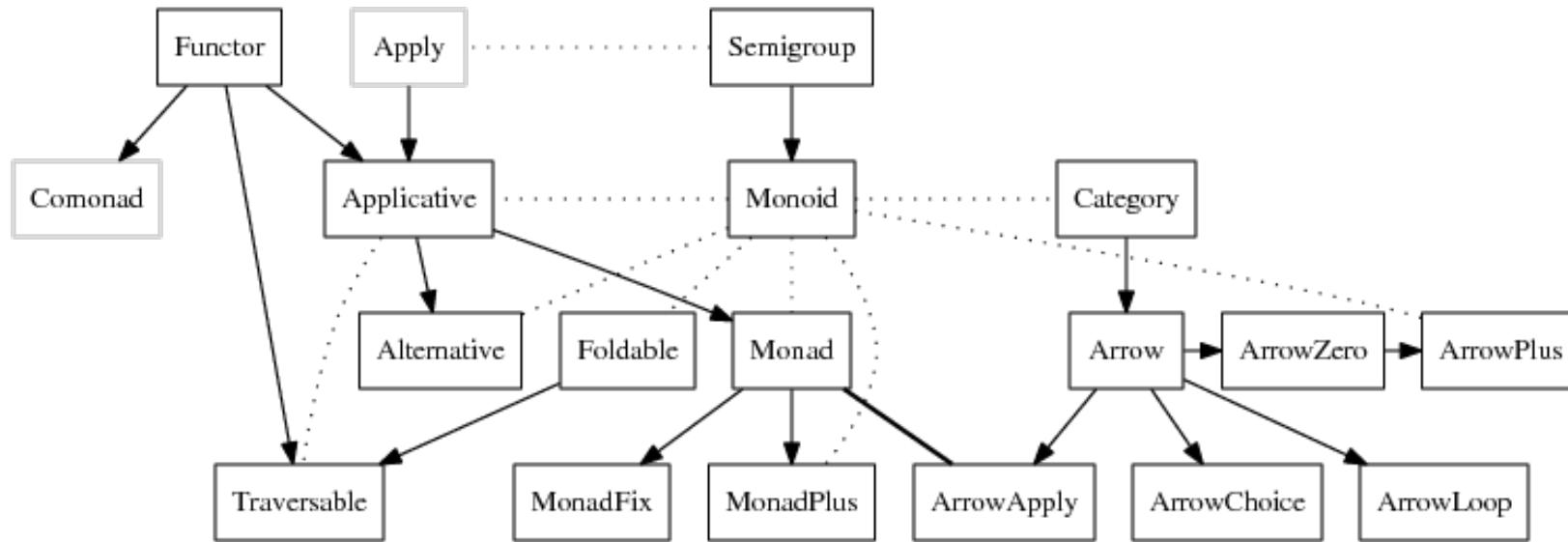
Turing languages

- 1957 — FORTRAN
- 1959 — COBOL, ALGOL
- 1962 — SIMULA
- 1972 — C, Smalltalk
- 1979 — C++
- 1991 — Python
- 1995 — Java

Church languages

- 1959 — LISP
- 1966 — ISWIM
- 1972 — Prolog
- 1978 — ML, FP
- 1985 — Miranda
- 1990 — Haskell

Church Languages Are Unfamiliar



```
Couldn't match type `k0' with `b'  
because type variable `b' would escape its scope  
This (rigid, skolem) type variable is bound by  
the type signature for  
groupBy :: Ord b => (a -> b) -> Set a -> Set (b, [a])  
The following variables have types that mention k0
```

PL Theory is Unfamiliar (and Dense)

$$\Sigma_0; \Gamma_1 \vdash A = \mathbf{D} \bar{v} : \mathbf{Set}_n \quad \Gamma = \Gamma_1(x : A)\Gamma_2$$

data $\mathbf{D} \Delta : \mathbf{Set}_n$ where $\overline{c_i \Delta_i [\bar{j}_i \mid b_i]} \in \Sigma_0 \quad \exists k. [\bar{j}_k \mid b_k] \neq []$

$$\left(\begin{array}{ll} \Delta'_i = \Delta_i(\bar{j}_i : \mathbb{I})[\bar{v} / \Delta] & \bar{q}_i = \hat{\Delta}_i[\bar{j}_i \mid b_i][\bar{v} / \Delta] \\ \rho_i = \mathbf{1}_{\Gamma_1} \uplus [c_i \bar{q}_i / x] & \rho'_i = \rho_i \uplus \mathbf{1}_{\Gamma_2} \\ \Theta_i = \text{BOUNDARY}(\bar{j}_i); \Theta & \\ \Sigma_{i-1}; \Gamma_1 \Delta'_i (\Gamma_2 \rho_i) \vdash f \bar{q} \rho'_i := Q_i : C \rho'_i \mid \Theta_i \rightsquigarrow \Sigma_i \end{array} \right)_{i=1\dots n}$$

$$\frac{\Delta_{\mathbf{hc}} = (r : \mathbb{I})(u : \mathbb{I} \rightarrow \text{Partial } r (\mathbf{D} \bar{v}))(u_0 : \mathbf{D} \bar{v} [r \mapsto u \ i0]) \\ \rho_{\mathbf{hc}} = \mathbf{1}_{\Gamma_1} \uplus [\mathbf{hcomp} \ r \ u \ u_0 / x] \quad \rho'_{\mathbf{hc}} = \rho_{\mathbf{hc}} \uplus \mathbf{1}_{\Gamma_2} \\ \Sigma_n; \Gamma_1 \Delta_{\mathbf{hc}} (\Gamma_2 \rho_{\mathbf{hc}}) \vdash f \bar{q} \rho'_{\mathbf{hc}} := Q_{\mathbf{hc}} : C \rho'_{\mathbf{hc}} \mid (r = 1); \Theta \rightsquigarrow \Sigma_{n+1}}{\Sigma_0; \Gamma \vdash f \bar{q} := \mathbf{case}_x \left\{ \begin{array}{l} c_1 \bar{q}_1 \mapsto Q_1; \dots; c_n \bar{q}_n \mapsto Q_n \\ \mathbf{hcomp} \ r \ u \ u_0 \mapsto Q_{\mathbf{hc}} \end{array} \right\} : C \mid \Theta \rightsquigarrow \Sigma_{n+1}}$$

The Future

Change Is Coming

- Software systems tend to be big, slow, and buggy
- There are broad forces at work that are compelling changes
 - Security and the increasing dependence on software
 - Revolution in the underlying hardware
 - A perpetual shortage of skilled programmers

Change Is Coming

- Software systems are big, slow, and buggy
 - And getting more so
- There are broad forces at work that are compelling changes
 - Security and the increasing dependence on software
 - Software verification
 - Revolution in the underlying hardware
 - Moving beyond Turing languages
 - A perpetual shortage of skilled programmers
 - Automation of programming

This Course

- Convey ethos of programming languages as a topic of study
 - And some of the important techniques
- Illustrate these ideas with examples
 - From current practice and research
- See the future before it happens

Tackling Complexity Through Modularity

Modular design is the key to successful programming. When writing a modular program to solve a problem, one first divides the problem into subproblems, then solves the subproblems, and finally combines the solutions.

The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language.

— John Hughes, “Why Functional Programming Matters” 1989

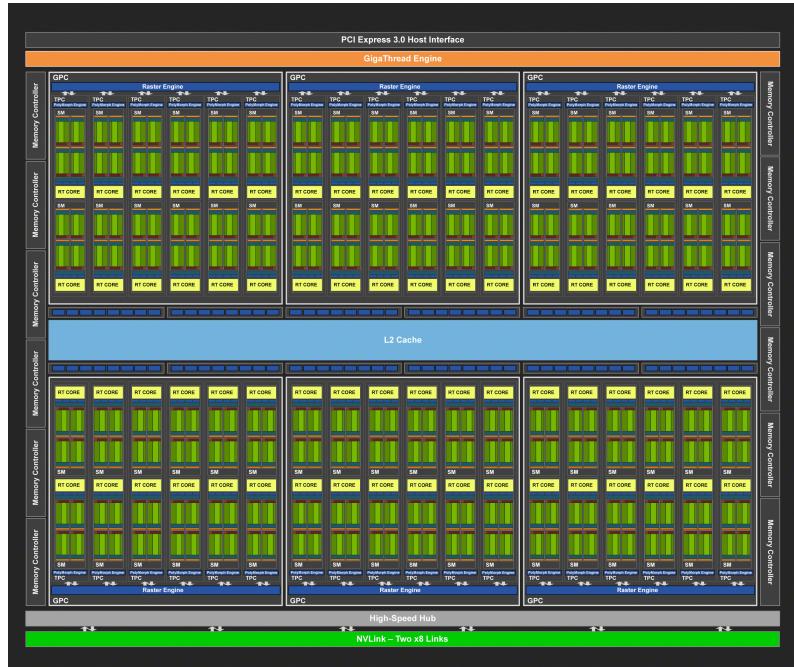
```
fun innerproduct(a, b, n):  
    c := 0  
    for i := 1 step 1 until n do  
        c := c + a[i] * b[i]  
    return c
```

- Statements operate on invisible state
 - Computes word-at-a-time by repetition of assignment/modification
 - Requires names for arguments, iterator, return value
-

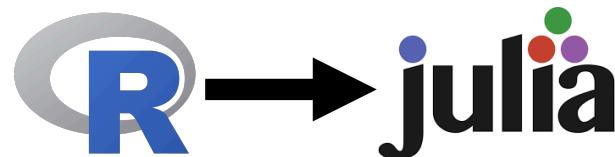
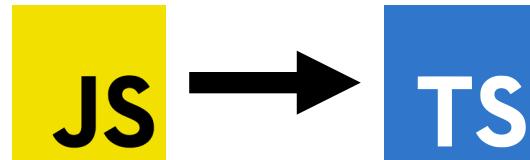
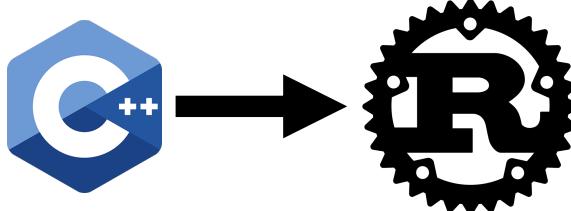
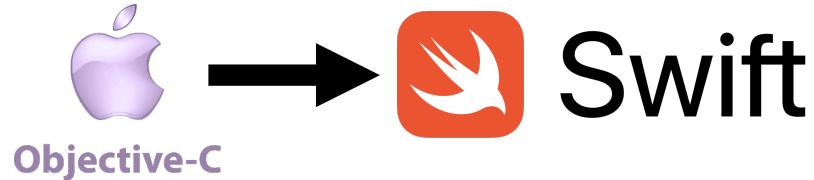
```
let innerproduct = zip |> (map *) |> (reduce +)
```

- Built from composable functions (map, reduce, pipe)
- Operates on whole conceptual units (lists), no repeated steps
- No names for arguments or temporaries

Radically Different Hardware is Here



New Kinds of Glue are Coming



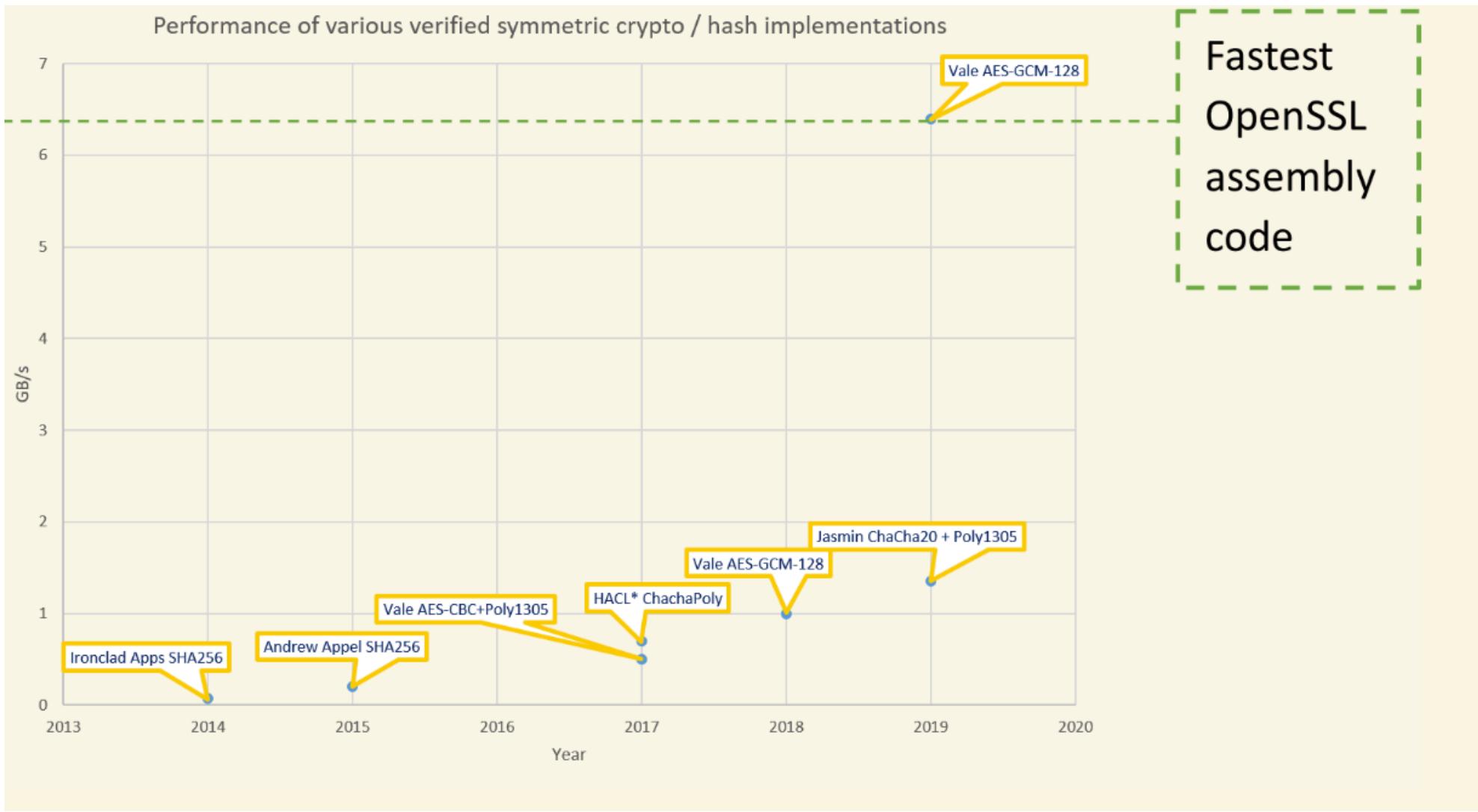
More Reliable Software is Coming

Messenger used to receive bugs reports on a daily basis; since the introduction of Reason, there have been a total of 10 bugs (that's during the whole year, not per week)! Refactoring speed went from days to hours to dozens of minutes.

— “Messenger.com Now 50% Converted to Reason” 2017

Being able to encode constraints of your application in the type system makes it possible to refactor, modify, or replace large swaths of code with confidence. Rust's error model forces developers to handle every corner case. [Our system] needs very little attention. We were able to leave it running without any issues through the holiday break.

— “Rust at OneSignal” 2017



Summary

- The world of software will change significantly
- The changes are driven by
 - New ideas in programming
 - Changes in underlying hardware
 - Changes in needs (e.g., security)
- In this course we will focus on
 - The new programming ideas
 - And the intellectual tools to understand the next generation of ideas