The Future of Programming Languages

Will Crichton

CS 242 — 11/28/18

Thesis:

The future of performance optimization is better programming models, not better optimizers.

Thesis:

programming languages

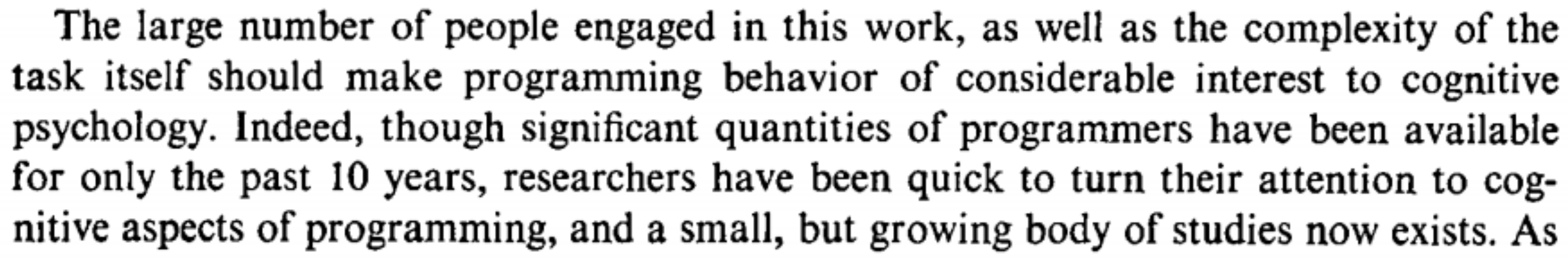
The future of ~~performance optimization~~ is better programming models, not better ~~optimizers~~.

compilers
program analyzers
runtimes

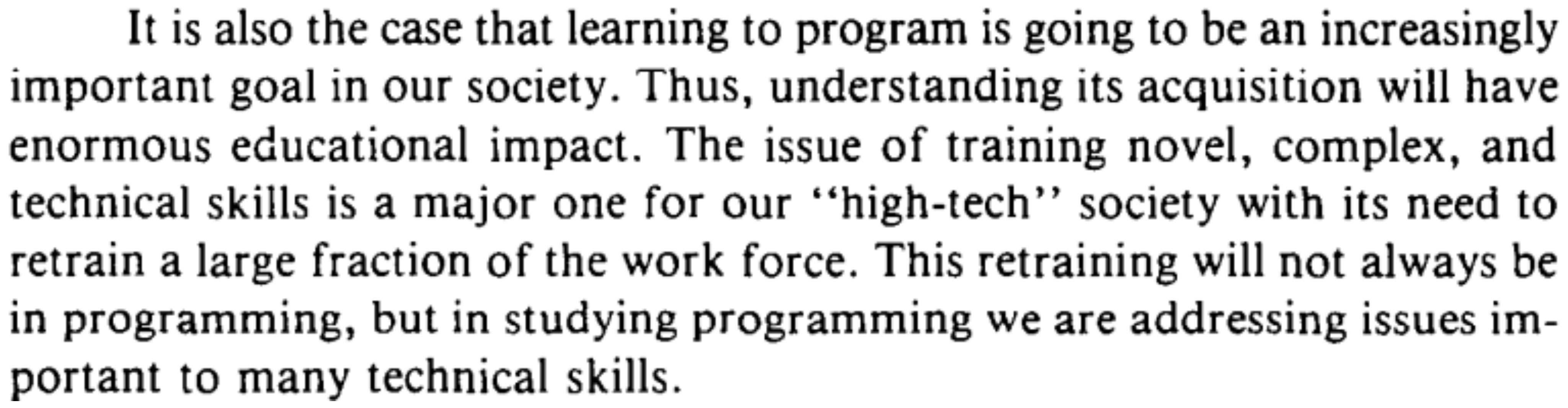
Hypothesis:

To build better programming models for people, we need better models of people programming.

Importance of cognition is well-known


The large number of people engaged in this work, as well as the complexity of the task itself should make programming behavior of considerable interest to cognitive psychology. Indeed, though significant quantities of programmers have been available for only the past 10 years, researchers have been quick to turn their attention to cognitive aspects of programming, and a small, but growing body of studies now exists. As

Ruven Brooks. "Towards a theory of the cognitive processes in computer programming." 1977


It is also the case that learning to program is going to be an increasingly important goal in our society. Thus, understanding its acquisition will have enormous educational impact. The issue of training novel, complex, and technical skills is a major one for our ''high-tech'' society with its need to retrain a large fraction of the work force. This retraining will not always be in programming, but in studying programming we are addressing issues important to many technical skills.

John Anderson et al. "Learning to Program in LISP." 1984

Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools

Brad A. Myers, Carnegie Mellon University
Andrew J. Ko, University of Washington
Thomas D. LaToza, George Mason University
YoungSeok Yoon, Google

Existing work is hard to generalize

Of the controlled experiments, only three show an effect large enough to have any practical significance. … Unfortunately, they all have issues that make it hard to draw a really strong conclusion.

In the Prechelt study, the populations were different between dynamic and typed languages, and the conditions for the tasks were also different. There was a follow-up study that … literally involves comparing code from Peter Norvig to code from random college students.

Dan Luu. "Static v. dynamic languages." 2015

Hypothesis:

Two main paths forward: large-scale analysis, and more rigorous cognitive science.

Survey says: PL features matter least

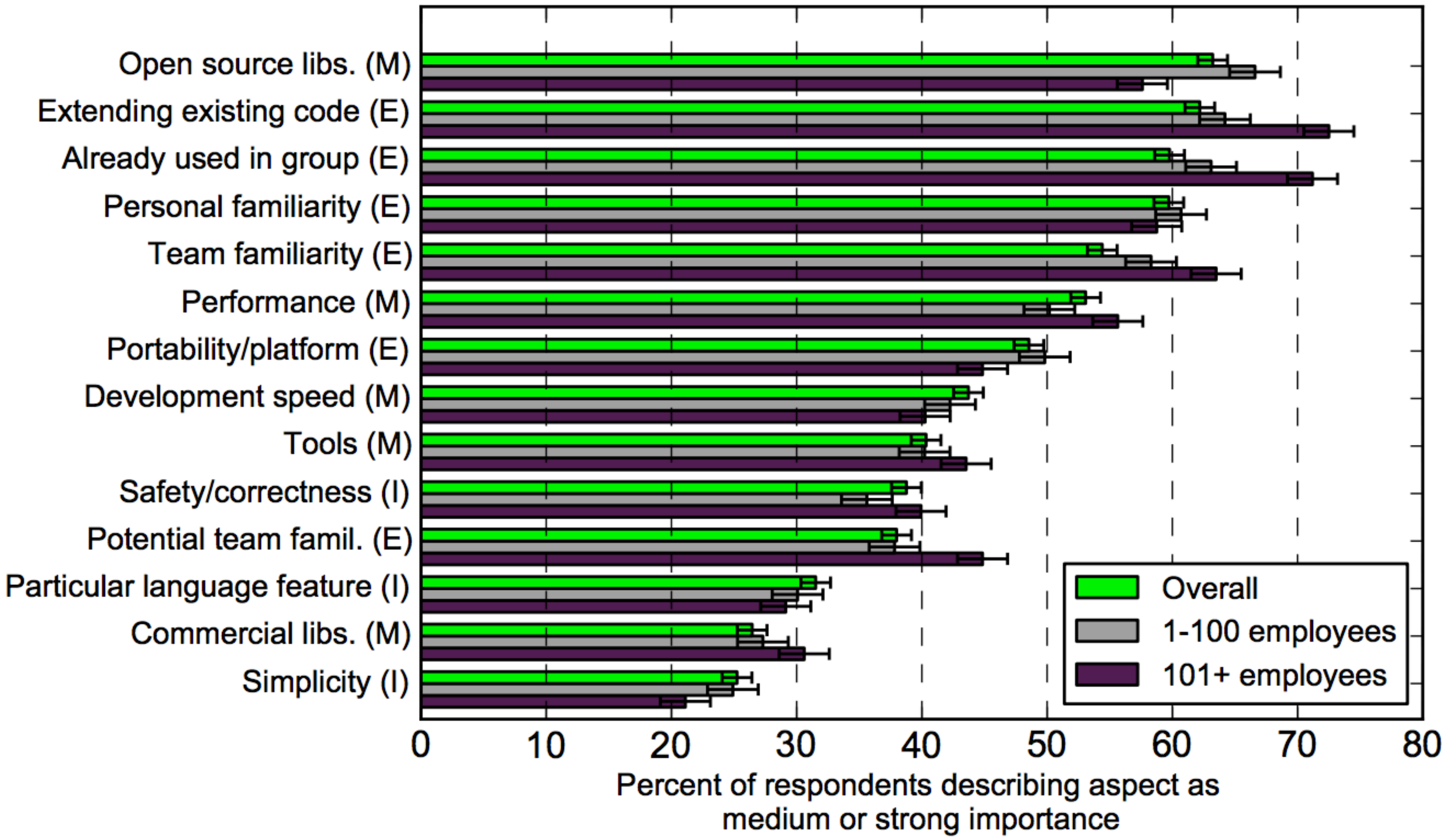


Figure 5: **Importance of different factors when picking a language**. Self-reported for every respondent's last project. Bars show standard error. E = Extrinsic factor, I = Intrinsic, M = Mixed. Shows results broken down by company size for respondents describing a work project and who indicated company size. (Slashdot, n = 1679)

Meyerovich et al. "Empirical Analysis of Programming Language Adoption". 2013

Survey says: Java is as hard as JS

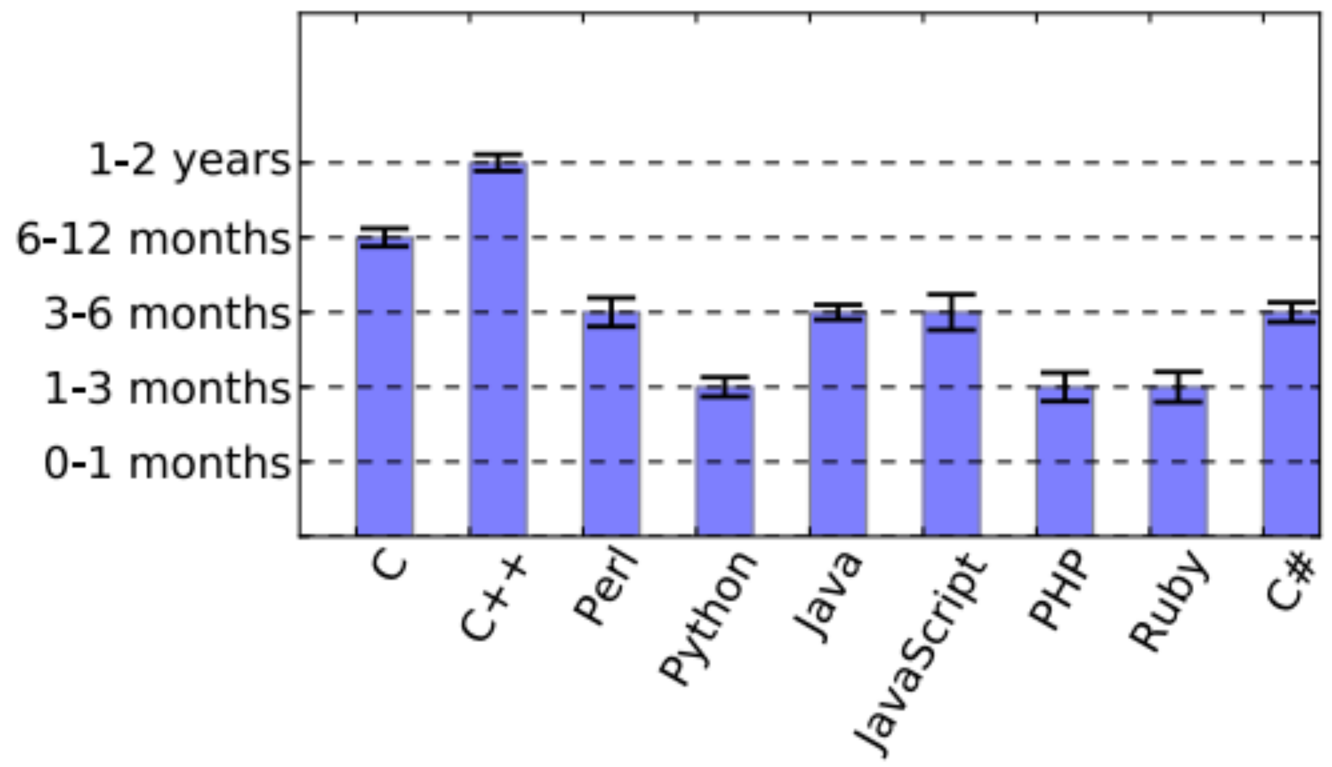
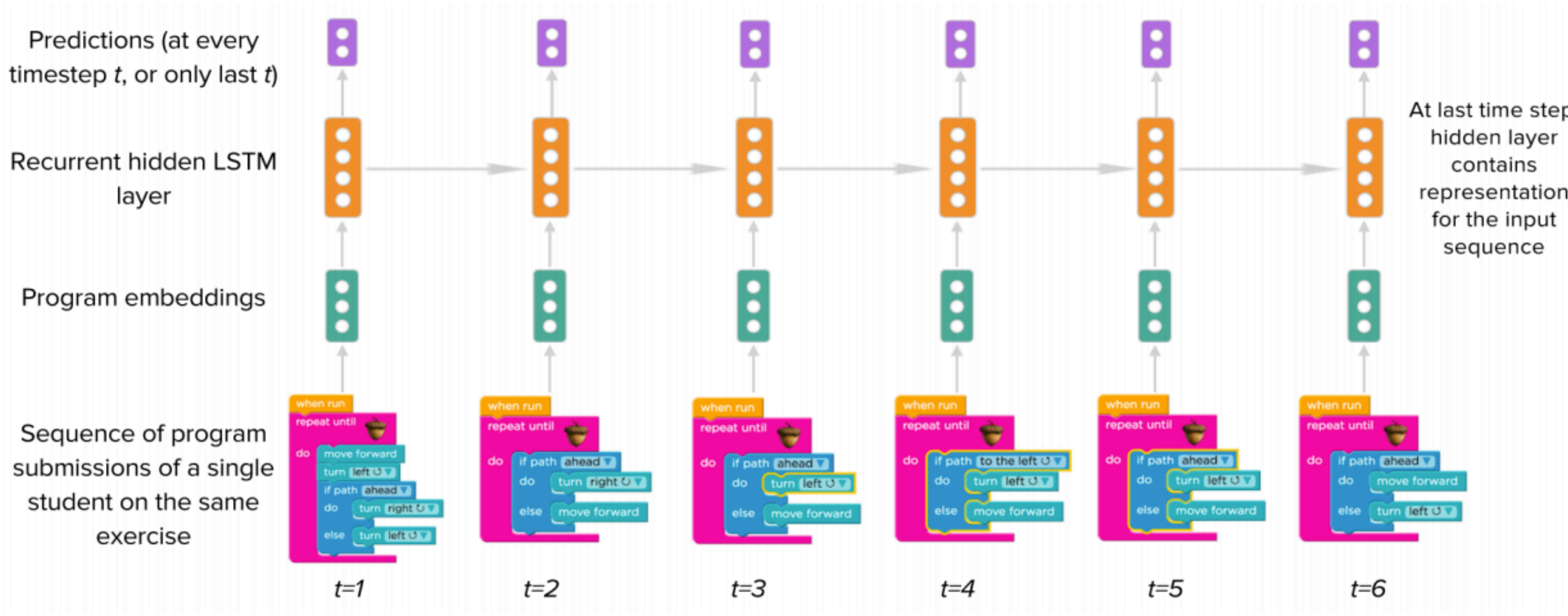


Figure 7: **Median reported speed of language acquisition.** Bars are standard error. (Slashdot, n = 1679)

Meyerovich et al. "Empirical Analysis of Programming Language Adoption". 2013

Deep knowledge tracing at scale

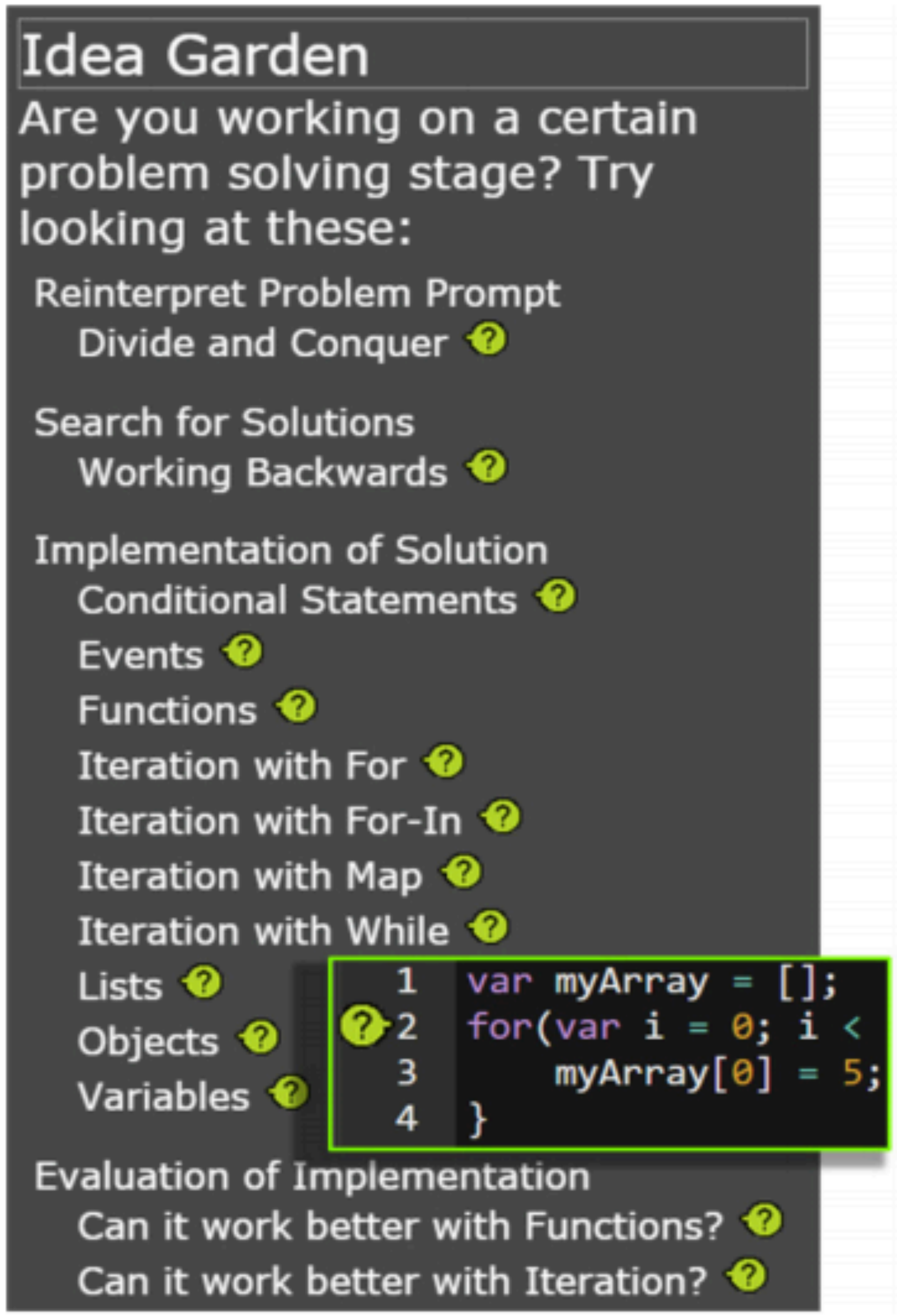

Predictions (at every timestep *t*, or only last *t*)
Recurrent hidden LSTM layer
Program embeddings
Sequence of program submissions of a single student on the same exercise
At last time step hidden layer contains representation for the input sequence
when run
repeat until
do
move forward
turn left
if path ahead
turn right
else
if path to the left
t=1
t=2
t=3
t=4
t=5
t=6

Wang et al. "Deep Knowledge Tracing on Programming Exercises." 2017

Metacognition in CS education

In this paper, we contribute an approach to promoting metacognitive awareness in introductory programming settings and investigate its effects on help requests, productivity, self-efficacy, and growth mindset.

Programming is not merely about language syntax and semantics, but more fundamentally about the iterative process of refining mental representations of computational problems and solutions and expressing those representations as code.



Idea Garden

Are you working on a certain problem solving stage? Try looking at these:

- Reinterpret Problem Prompt
  - Divide and Conquer
- Search for Solutions
  - Working Backwards
- Implementation of Solution
  - Conditional Statements
  - Events
  - Functions
  - Iteration with For
  - Iteration with For-In
  - Iteration with Map
  - Iteration with While
  - Lists
  - Objects
  - Variables
- Evaluation of Implementation
  - Can it work better with Functions?
  - Can it work better with Iteration?

```
var myArray = [];
for(var i = 0; i <
    myArray[0] = 5;
}
```

Loksa et al. “Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance”. 2016

Subvocalization for tracking cognition

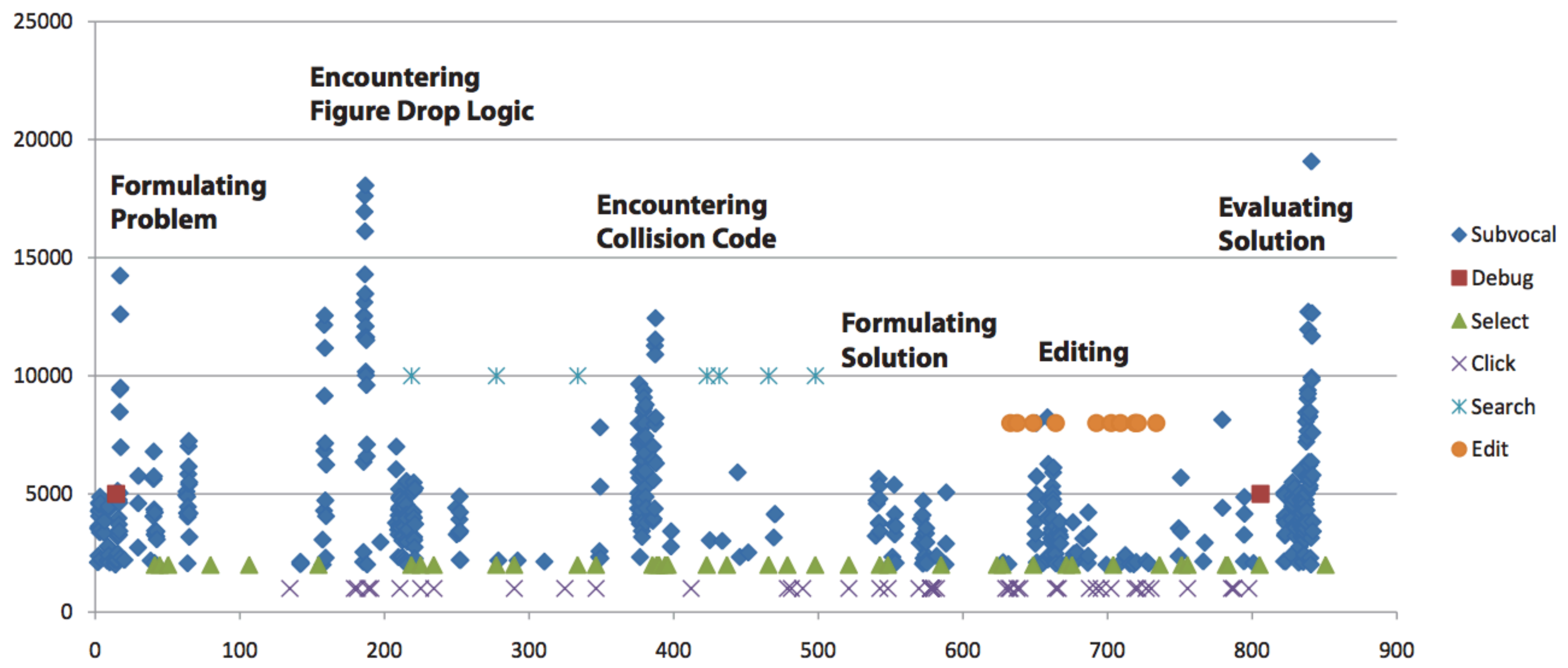


Figure 3. EMG and Programming events over 13 minutes of activity.

Chris Parnin. "Subvocalization – Towards Hearing the Inner Thoughts of Developers". 2011

Programming is … about the iterative process of refining mental representations of computational problems and solutions and expressing those representations as code.

Loksa et al. “Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance”. 2016

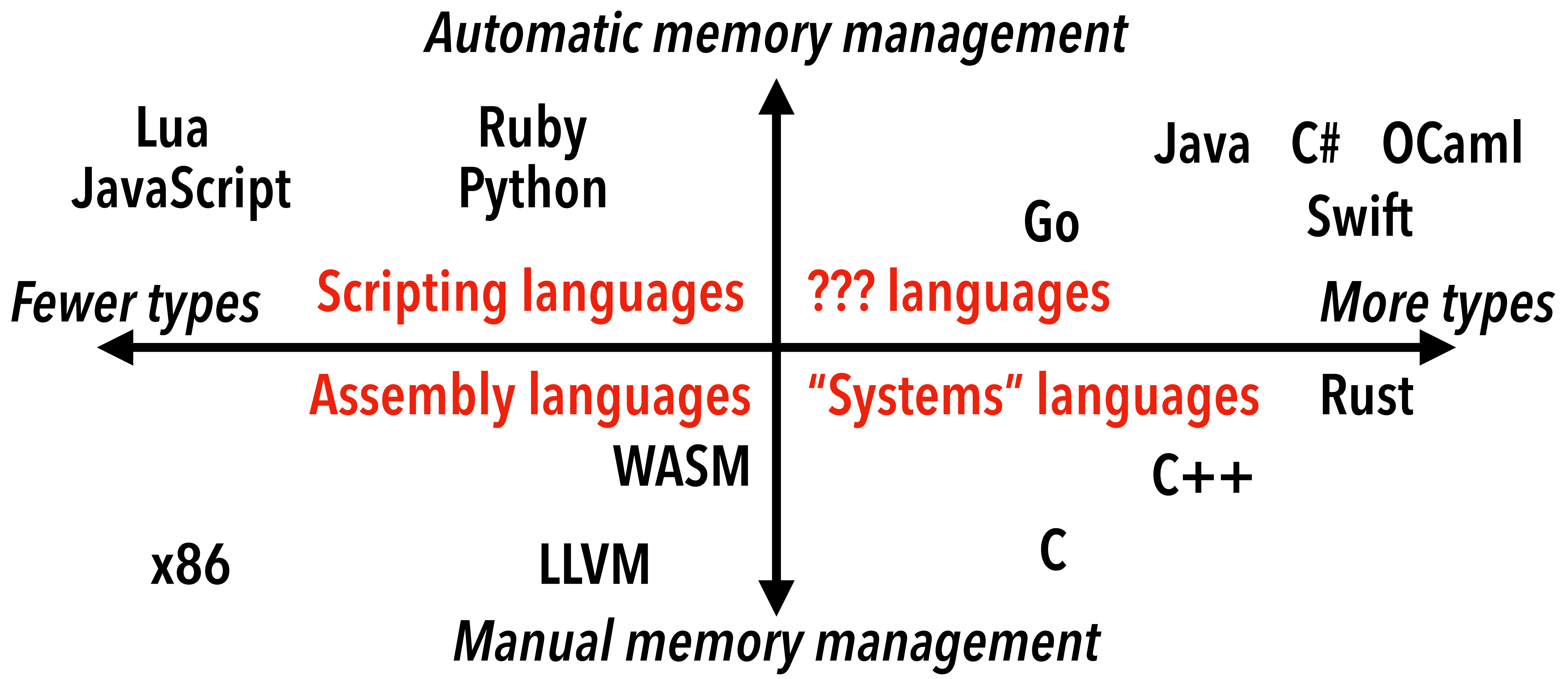
Programmers accumulate knowledge about their programs over time

- Programming a new system is touch-and-go
  - Don't know what the types should be, data schemas rapidly evolved
  - Code may be partially broken, but those paths won't be tested
  - "Almost right" is better than a compiler error

- Once you are more confident with types, write them down
  - And have the compiler enforce them

- Once you hit a bottleneck, add performant code
  - Manage memory yourself, don't rely on the garbage collector

Taxonomy of modern GPPLs



Fibonacci: Lua

```lua
function fib(n)
  if n == 0 or n == 1 then
    return n
  else
    return fib(n - 1) + fib(n - 2)
end
```

Fibonacci: OCaml

```ocaml
let rec fib (n : any) : any =
  let n : int = Obj.magic n in
  if n = 0 || n = 1 then
    n
  else
    Obj.magic (fib (Obj.magic (n - 1))) +
    Obj.magic (fib (Obj.magic (n - 2)))
```

Fibonacci: Rust

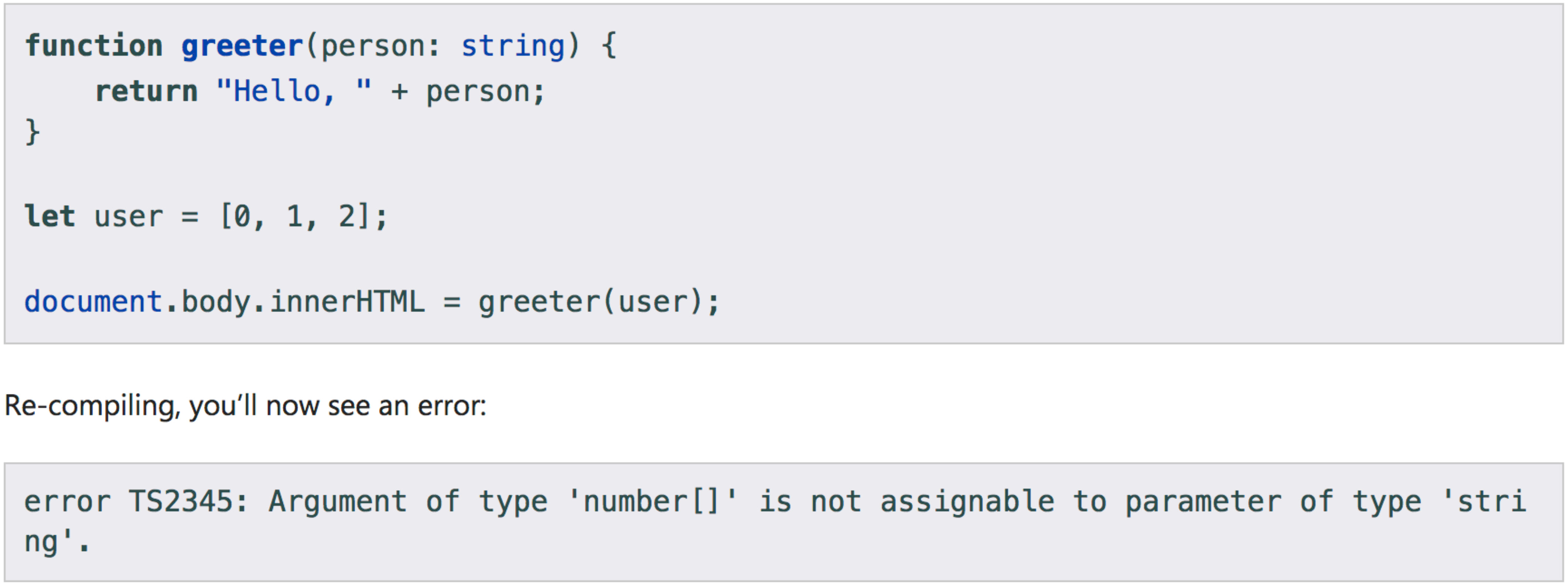
```rust
fn fib(n_dyn: Rc<Any>) -> Rc<Any> {
    let n_static: &i32 =
        n_dyn.downcast_ref::<i32>().unwrap();
    if *n_static == 0 {
        Rc::new(Box::new(*n_static))
    } else {
        let n1 = fib(Rc::new(Box::new(n_static - 1)));
        let n2 = fib(Rc::new(Box::new(n_static - 2)));
        Rc::new(
            n1.downcast_ref::<i32>().unwrap() +
            n2.downcast_ref::<i32>().unwrap())
    }
}
```

Key difference is static analysis

- What distinguishes languages is the level of static analysis
  - Plus facilities for checking non-inferrable/annotatable info at runtime
  - Tier 1 (“scripting”): runtime types and memory
  - Tier 2 (“functional”): static types, runtime memory
  - Tier 3 (“systems”): static types and memory
- It’s “easy” to defer static checks to runtime, but conceptual/ syntactic overhead increases
  - Rc<T> and Any in Rust
  - Obj.magic in OCaml

We need solutions to permit gradual migration from one to the other

Gradual typing crosses the type barrier

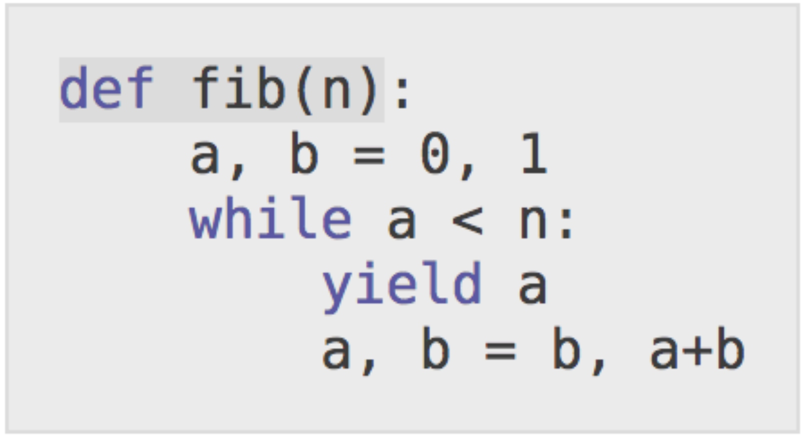


```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = [0, 1, 2];

document.body.innerHTML = greeter(user);
```

Re-compiling, you'll now see an error:

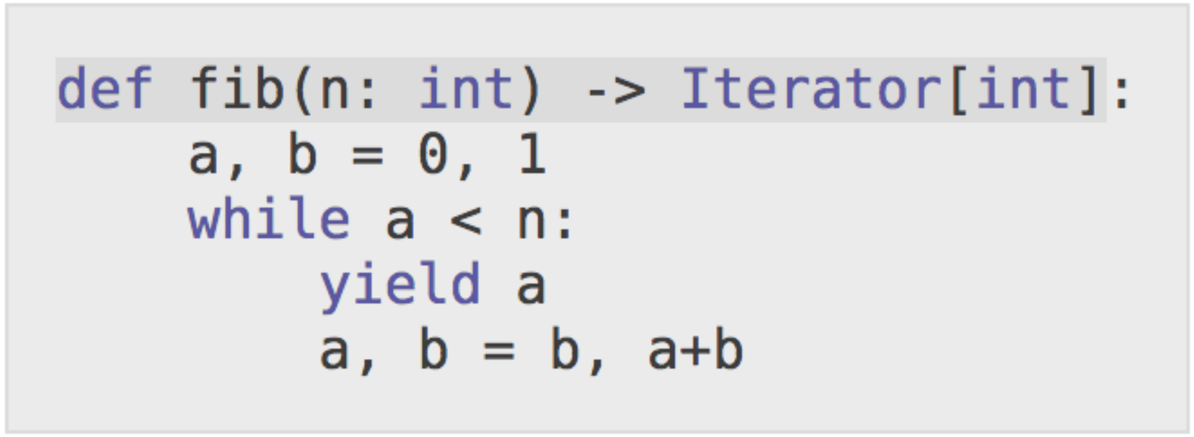
```
error TS2345: Argument of type 'number[]' is not assignable to parameter of type 'stri
ng'.
```

From Python...



```
def fib(n):
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

...to statically typed Python



```
def fib(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

Gradual memory management?

- No easy way to mix memory management solutions
  - C++/Rust make it possible to mix reference counting and lifetimes
  - But with heavy syntactic overhead
- Lua virtual stack solved this problem, but not easily
- Little/no published research here—open problem!

Issues in gradual systems

- Debuggability and blame
  - How do we know whether a value has had its type inferred or deferred? (Likely need to investigate IDE integration)
  - If an error occurs, what's the source of the cause? (Who's to blame?)
  - Broadly: when the compiler makes a decision for us, we need to understand that decision
- Performance
  - "Is Sound Gradual Typing Dead?" - 0.5x - 68x overhead relative to untyped code
  - No existing systems take advantage of potential perf benefits

Takeaways

- Understand the human to build better programming models
- Gradual programming is a promising PL technique that matches the human programming process