Programming Languages of the Future

CS 242

December 6, 2017

Improving a PL

1. Determine what to improve
2. Determine how to improve it

Meta-problem: lack of good metrics

- Most research: "I or people I know have this problem"

- How do we know what matters in the real world?
  - Growing gap between industry and academia
  - Intellectually interesting doesn't mean important in practice!

- Need HCI for a principled approach

Survey says: PL features matter least

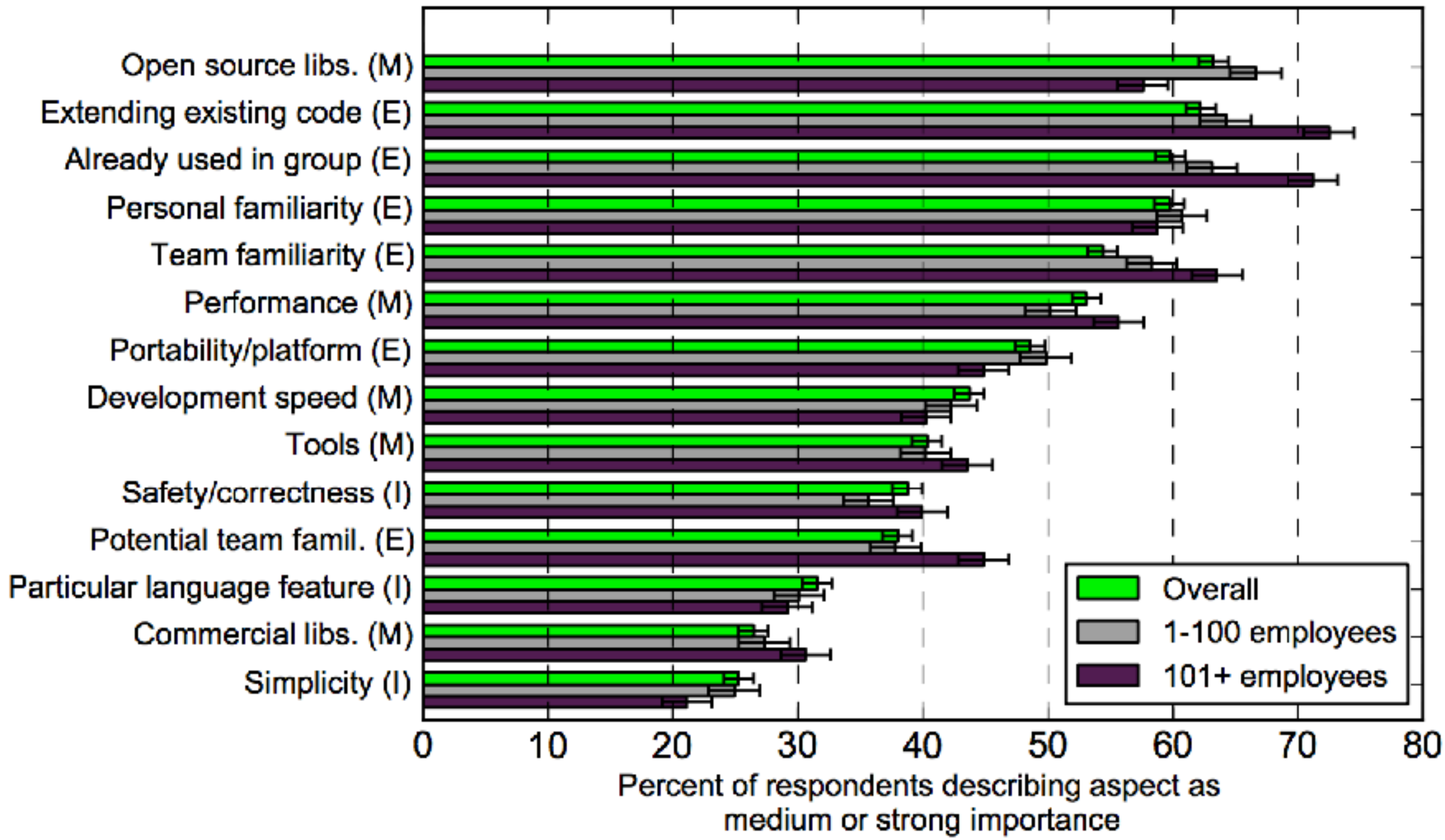


Figure 5: **Importance of different factors when picking a language**. Self-reported for every respondent's last project. Bars show standard error. E = Extrinsic factor, I = Intrinsic, M = Mixed. Shows results broken down by company size for respondents describing a work project and who indicated company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

[Meyerovich et al. '13]

Who needs PL improvements?

- Students?
  - Block-based vs text-based programming
  - "But in Java, you can like figure out how to do like, all the other stuff."
- Industry devs?
  - "Tools that help developers pick up where they left off"
  - "Tools that can generate documentation for legacy code"
- Academics? Library writers? Hardware devs?

Progress will be driven by applications

- Rust: Mozilla needed a faster web browser
- TypeScript: the world needed a better JavaScript
- Go: Google needed a faster Java for web servers

Hypothesis:

Interoperability is the most critical issue in programming languages today.

Interoperability is a problem

- There is no one true programming paradigm
  - Functional, imperative, declarative, dynamically typed, statically typed, low-level, high-level, …
  - They all have their time and place
- Languages are built in siloed ecosystems
  - No simple way to translate between values (e.g. Python list -> Java list)
  - How many people have to implement printf? JSON parsers?
- Programs need to either incorporate multiple paradigms or gradually move between them

Example #1: web programming

- SQL generated as strings –> SQL injection attacks
- Repeated features across multiple UI languages
  - HTML/CSS started life as external, wholly separate languages
  - “What if I want variables in my CSS?” –> LESS, SASS, Jade…
  - “What if I want to conditionally generate HTML?” -> PHP, Handlebars, Mustache, …

ReactJS

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

Example #2: evolving codebases

- As a startup, want dynamic scripting languages
  - e.g. Python
  - Fast iteration cycle
  - Partially broken code can still run
- As a big company, want type-checked compiled languages
  - Modules matter most—allow many teams to work independently
  - Correctness issues drastically reduce developer time, harder to debug across large code bases
- Today: completely different ecosystems
  - Can't just add types to a Python script (until recently)
  - Evolution means rewriting entire codebase
  - Too much of a competitive disadvantage

Example #3: game development

- Performance requirements: real-time, 60+ FPS, no freezes, 4K rendering, physics simulation, …

- Scripting requirements: high level, extensible, dynamic, interoperable with low-level interface

- Best example is Lua, but coding at the boundary still sucks
  - Programming interface turns into a stack machine language
  - Not trivial to deal with memory allocation
  - No simple type translation for composite structures

Option 1: Improve compatibility between existing languages

C is the lingua franca of PLs

- Many languages can convert to/from C types
  - Java JNI, Python ctypes, Go cgo
- C ABI becomes the lowest common denominator
- APIs are complex, fragile, can't capture memory management

Protobufs: serializable structs

Person.proto

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

PersonWriter.java

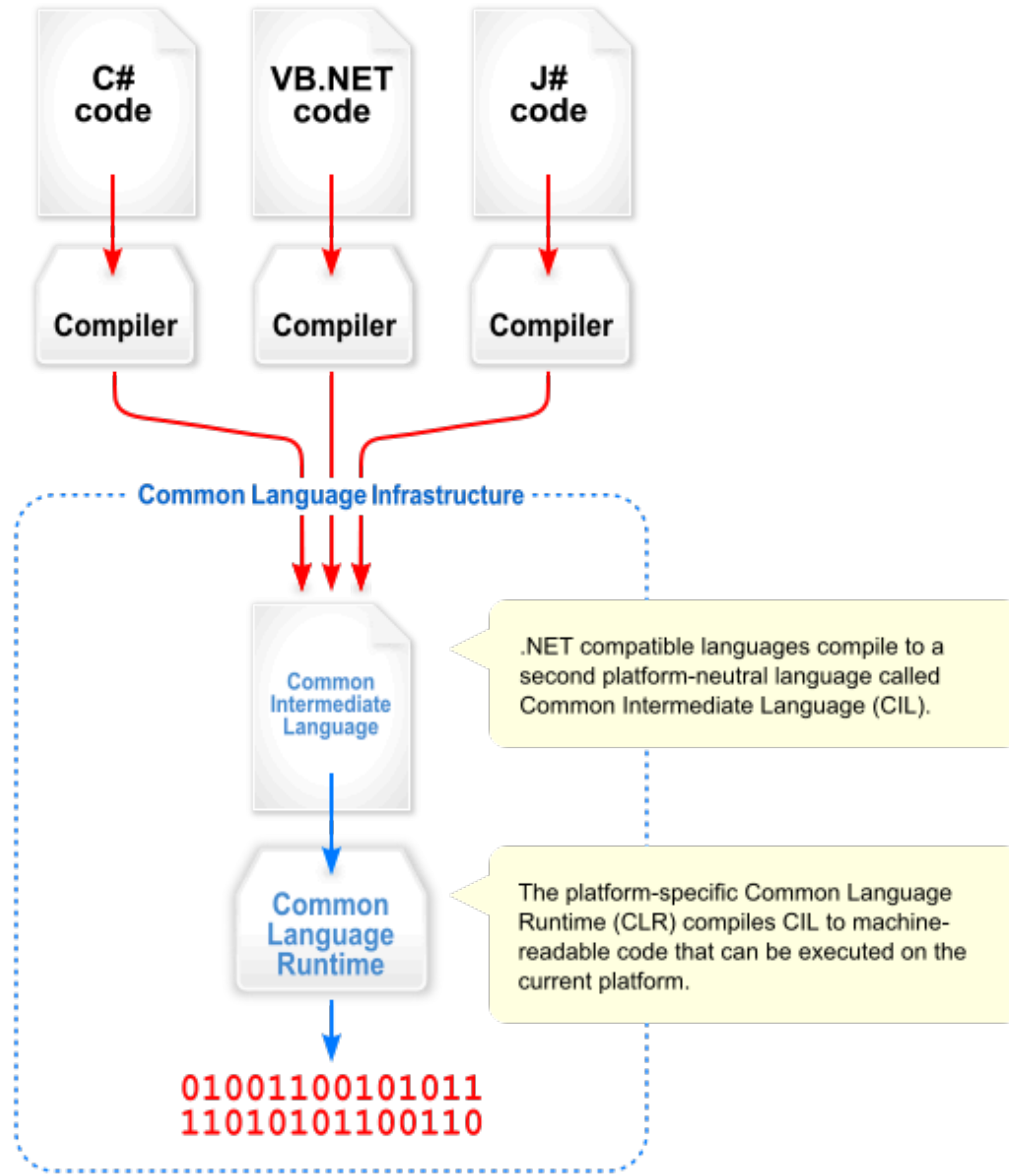
```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0])
john.writeTo(output);
```

PersonReader.cpp

```
Person john;
fstream input(argv[1],
    ios::in | ios::binary);
john.ParseFromIstream(&input);
id = john.id();
name = john.name();
email = john.email();
```

.NET: Common Language Infrastructure

- Provides classes, structs, enums, interfaces
- Requires using the full .NET stack



Option 2: Build a new language

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

How can this process be reflected in our programming languages?

Bad: programmer writes assertions

```
def incr(n):
    return n + 1
```

↓

```
def incr(n):
    assert(type(n) == int)
    return n + 1
```

Bad: programmer writes assertions

```
std::shared_ptr<int> x;
*x = 1;
```

```
int* x = new int;
*x = 1;
delete x;
```

Good: assertions part of the language

- Types: either annotatable or inferable
  - Ensures programmers don't forget to assert a type
  - Permits checking of code before it runs (static analysis is productive!)
- Memory: should be treated similarly
  - It's 2017, all languages should be memory safe
  - Question is whether data lifetimes should be determined at compile time (a la Rust) or run time (everything else)

Key difference is static analysis

- What distinguishes languages is the level of static analysis
  - Plus facilities for checking non-inferrable/annotatable info at runtime
  - Scripting: runtime types and memory
  - Functional: static types, runtime memory
  - Systems: static types and memory

- It's "easy" to defer static checks to runtime, but conceptual overhead increases
  - Rc<T> and Any in Rust
  - Obj.magic in OCaml

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

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

Fibonacci: 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())
    }
}
```

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
- Recall: 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

Let’s go implement these languages!
…But how much work is that?

Meta-problem:

Little reusable language infrastructure

Issue #1: Writing the compiler

- People love talking about and writing compilers
  - Billions of resources, many classes
  - But so much repeated code!!
- If you want to implement e.g. a statically typed, object oriented language, you have three options:
  1. LLVM or C
  2. Java bytecode
  3. .NET
- Potentially have to implement:
  - Lexer/parser, type system, code generator + JIT compiler, garbage collector

Possible solutions for reusable infra

- Solution #1: don't bother, write a prototype and let someone else take care of the rest
  - Cyclone ['02] language inspired Rust
  - Many modern langs (e.g. Swift) inspired by OCaml/Haskell
- Solution #2: compile to a higher-level language
  - Growing niche of compile-to-C languages for easier codegen
  - Hypothesis: "Rust is the new LLVM"
- Solution #3: build out generic language infrastructure
  - Most infra is tightly coupled to the language
  - Reusable type system? Reusable documentation generator?

Compile-to-lang = metaprogramming

- Active work on embedding DSLs into existing languages
  - Need a good macro system—also active research
  - Many languages are just a nice syntax on top of a normal library, e.g. HTML, SQL, TensorFlow
- Again, debuggability and blame arise
  - If you compile SQL to Rust and there's a type error, where in the SQL does it come from?

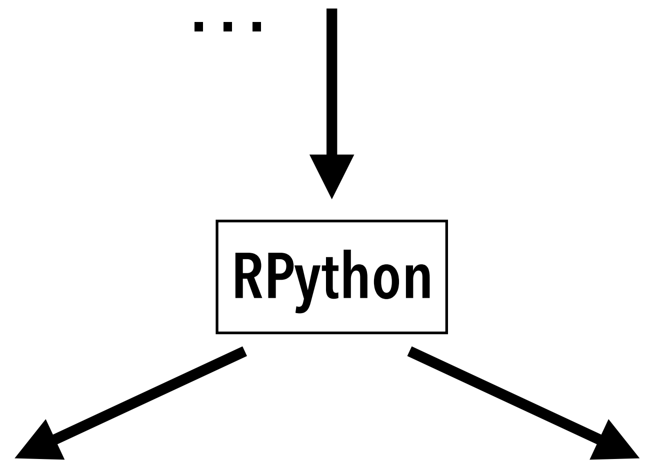
Composable, programmable macros

```
let imageBase : URL = <images.example.com>
let bgImage : URL = <%imageBase%/background.png>
new : SearchServer
  def resultsFor(searchQuery, page)
    serve(~) (* serve : HTML -> Unit *)
      >html
        >head
          >title Search Results
          >style ~
            body { background-image: url(%bgImage%) }
            #search { background-color: %darken(‘#aabbcc‘, 10pct)% }
        >body
          >h1 Results for <{HTML.Text(searchQuery)}:
          >div[id="search"]
            Search again: < SearchBox("Go!")
          < (* fmt_results : DB * SQLQuery * Nat * Nat -> HTML *)
             fmt_results(db, ~, 10, page)
               SELECT * FROM products WHERE {searchQuery} in title
```

[Omar '14]

RPython: JIT generator

```
def interpret():
    while True:
        instr = get_instruction()
        if instr == INSTR_ADD:
            push(pop() + pop())
        else:
            ...
```




```
void interpret() {
  while (true) {
    Instr instr = get_instruction();
    if (instr == INSTR_ADD) {
      push(pop() + pop());
    } else if (...) {
      ...
    }
  }
}
```

```
void jit(Instr* instructions) {
  std::string src;
  for (Instr instr : instructions) {
    if (instr == INSTR_ADD) {
      src += "push(pop() + pop());";
    } else if (...) {
      ...
    }
  }
  compile(src);
}
```

Issue #2: Everything else

- From Alex's lecture: devs need good tooling
  - Compiler, cross-platform code generation, package manager, documentation generator, release manager, debugger, editor integration, syntax formatter, standard library, websites, community outreach, …
- Some steps in this direction
  - Language Server Protocol helps with IDE integration
  - Compile-to-C can reuse tools like gdb with some effort