

The com.lihaoyi Ecosystem Executable Scala Pseudocode that's Easy, Boring and Fast!

Li Haoyi, Scaladays Seattle 7 June 2023

About Haoyi

Been using Scala since 2012

Work on the Databricks Developer Platform

Lots of Scala OSS work: Scala.js, Mill,
Ammonite, uPickle, Fastparse, ...

Author of Hands-on Scala Programming



Executable Scala Pseudocode that's Easy, Boring and Fast!

1. What is the com.lihaoyi ecosystem?
2. What is the com.lihaoyi ecosystem not?
3. Principles of the com.lihaoyi ecosystem
4. Case Study: How Mill Makes Builds Great

Executable Scala Pseudocode that's Easy, Boring and Fast!

1. **What is the com.lihaoyi ecosystem?**
2. What is the com.lihaoyi ecosystem not?
3. Principles of the com.lihaoyi ecosystem
4. Case Study: How Mill Makes Builds Great

1.1 What is the com.lihaoyi Ecosystem?

- (2012) Scalatags: *HTML Generation Library*
- (2014) FastParse: *Fast Parser Combinators*
- (2014) uTest: *Minimal Testing Library*
- (2014) uPickle: *JSON & Binary Serialization Library*
- (2015) Ammonite: *Fancy Scala REPL*
- (2016) Sourcecode: *source file-name/lines/etc.*
- (2017) Mill: *Better Scala Build Tool*
- (2017) PPrint: *Pretty-Printing Library*
- (2018) OS-Lib: *Filesystem/Subprocess Library*
- (2018) Requests-Scala: *HTTP Request Library*
- (2018) Cask: *HTTP Micro-Framework*
- (2020) MainArgs: *CLI Argument Parsing Library*

The com.lihaoyi ecosystem is a collection of small libraries and tools that I've built up over the past decade. It includes many of the fundamentals that you would need to do any sort of software engineering work: data serialization, HTTP clients and servers, filesystem operations, and so on

1.1 What is the com.lihaoyi Ecosystem?

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: *Minimal Testing Library*

(2014) uPickle: *JSON & Binary Serialization Library*

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: *HTTP Request Library*

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: *CLI Argument Parsing Library*

All under <https://github.com/com-lihaoyi>,
separate from <https://github.com/lihaoyi>



These libraries, originally on my personal github and published under my personal Maven Central account, are under a com-lihaoyi github organization, with a number of maintainers.

- (2012) Scalatags: *HTML Generation Library*
- (2014) FastParse: *Fast Parser Combinators*
- (2014) uTest: *Minimal Testing Library*
- (2014) uPickle: *JSON & Binary Serialization Library*
- (2015) Ammonite: *Fancy Scala REPL*
- (2016) Sourcecode: *source file-name/lines/etc.*
- (2017) Mill: *Better Scala Build Tool*
- (2017) PPrint: *Pretty-Printing Library*
- (2018) OS-Lib: *Filesystem/Subprocess Library*
- (2018) Requests-Scala: *HTTP Request Library*
- (2018) Cask: *HTTP Micro-Framework*
- (2020) MainArgs: *CLI Argument Parsing Library*

1.1 What is the com.lihaoyi Ecosystem?

All under <https://github.com/com-lihaoyi>,
separate from <https://github.com/lihaoyi>



We get about 18 million downloads a month on Maven Central. This is certainly far less than other more well-known Scala ecosystems, but it does indicate that these libraries do get a good amount of usage in the wild.

(2012) Scalatags: *HTML Generation Library*
(2014) FastParse: *Fast Parser Combinators*
(2014) uTest: *Minimal Testing Library*
(2014) uPickle: *JSON & Binary Serialization Library*
(2015) Ammonite: *Fancy Scala REPL*
(2016) Sourcecode: *source file-name/lines/etc.*
(2017) Mill: *Better Scala Build Tool*
(2017) PPrint: *Pretty-Printing Library*
(2018) OS-Lib: *Filesystem/Subprocess Library*
(2018) Requests-Scala: *HTTP Request Library*
(2018) Cask: *HTTP Micro-Framework*
(2020) MainArgs: *CLI Argument Parsing Library*

1.1 What is the com.lihaoyi Ecosystem?

All under <https://github.com/com-lihaoyi>,
separate from <https://github.com/lihaoyi>



Largely a self-contained ecosystem

Lastly, the ecosystem is largely self-contained. We do not pull in heavy dependencies on things like Akka or Cats or ZIO. This keeps the total amount of “stuff” you need to know to use these libraries tightly bounded. This makes them an easy way to get started productively with Scala before perhaps graduating to more sophisticated frameworks.

This helps make these libraries broadly applicable. Whether you come from an Akka background, or Pure FP, or learning Scala for the first time, there is definitely something here you could make use of

1.2 What is the com.lihaoyi Ecosystem?

(2012) Scalatags: HTML Generation Library

(2014) FastParse: *Fast Parser Combinators*
(2014) uTest: *Minimal Testing Library*
(2014) uPickle: *JSON & Binary Serialization Library*
(2015) Ammonite: *Fancy Scala REPL*
(2016) Sourcecode: *source file-name/lines/etc.*
(2017) Mill: *Better Scala Build Tool*
(2017) PPrint: *Pretty-Printing Library*
(2018) OS-Lib: *Filesystem/Subprocess Library*
(2018) Requests-Scala: *HTTP Request Library*
(2018) Cask: *HTTP Micro-Framework*
(2020) MainArgs: *CLI Argument Parsing Library*

```
body(  
  div(  
    h1(id="title", "This is a title"),  
    p("This is a big paragraph of text")  
  )  
)  
  
<body>  
  <div>  
    <h1 id="title">This is a title</h1>  
    <p>This is a big paragraph of text</p>  
  </div>  
</body>
```

Let's do a short tour of the com.lihaoyi libraries. I have listed the most notable ones here on the left, starting from Scalatags which was released in 2012

Scalatags is a small HTML generation library, letting you write nested method calls (body, div, h1, etc.) with attributes (such as id) that render into HTML snippets you can include on your website

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: *Minimal Testing Library*

(2014) uPickle: *JSON & Binary Serialization Library*

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: *HTTP Request Library*

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: *CLI Argument Parsing Library*

1.3 What is the com.lihaoyi Ecosystem?

```
def eval(tree: (Int, Seq[(String, Int)])): Int = ???

def number[$: P] =
  P(CharIn("0-9").rep(1).!).map(_.toInt)

def parens[$: P] = P("(" ~/ addSub ~ ")")
def factor[$: P] = P(number | parens)

def divMul[$: P] =
  P(factor ~ (CharIn("*/").! ~/ factor).rep).map(eval)

def addSub[$: P] =
  P(divMul ~ (CharIn("+\\-").! ~/ divMul).rep).map(eval)

def expr[$: P]: P[Int] = P(addSub ~ End)

fastparse.parse("((1+1*2)+(3*4*5))/3", expr(_))
// Parsed.Success(21, _)
```

We have FastParse, introduced in 2014. This is a parser combinator library, similar to Scala's old standard library parser combinators: you can basically write out the grammar of the thing you want to parse - number is a char in 0-9 repeated more than once, parens is an "(" followed by something followed by a ")". FastParse stands out over other parsing libraries with great error messages, and great performance: ~300x faster than the std lib parser combinators.

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: Minimal Testing Library

(2014) uPickle: *JSON & Binary Serialization Library*

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: *HTTP Request Library*

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: *CLI Argument Parsing Library*

1.4 What is the com.lihaoyi Ecosystem?

```
object HelloTests extends TestSuite{
  val tests = Tests{
    test("test1"){
      throw new Exception("test1")
    }

    test("test2"){
      val a = List[Byte](1, 2)
      assert(a(0) == 1)
    }
  }
}
```

We have uTest, a minimal testing framework for writing and running unit tests.

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: *Minimal Testing Library*

(2014) uPickle: JSON & Binary Serialization Library

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: *HTTP Request Library*

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: *CLI Argument Parsing Library*

1.5 What is the com.lihaoyi Ecosystem?

```
// JSON
write(Seq(1, 2, 3))      ==> "[1,2,3]"

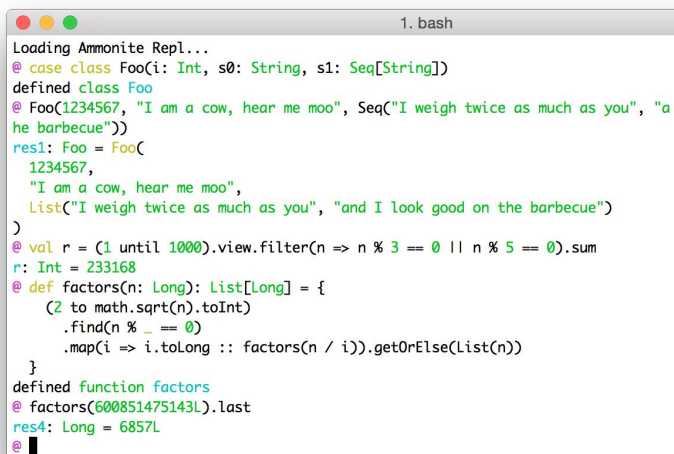
read[Seq[Int]]("[1,2,3]") ==> List(1, 2, 3)

// MsgPack
writeBinary(Seq(1, 2, 3)) ==>
  Array[Byte](0x93, 0x01, 0x02, 0x03)
```

uPickle, a JSON and binary data serialization library for all your Scala data structures and case classes

- (2012) Scalatags: *HTML Generation Library*
- (2014) FastParse: *Fast Parser Combinators*
- (2014) uTest: *Minimal Testing Library*
- (2014) uPickle: *JSON & Binary Serialization Library*
- (2015) Ammonite: *Fancy Scala REPL***
- (2016) Sourcecode: *source file-name/lines/etc.*
- (2017) Mill: *Better Scala Build Tool*
- (2017) PPrint: *Pretty-Printing Library*
- (2018) OS-Lib: *Filesystem/Subprocess Library*
- (2018) Requests-Scala: *HTTP Request Library*
- (2018) Cask: *HTTP Micro-Framework*
- (2020) MainArgs: *CLI Argument Parsing Library*

1.6 What is the com.lihaoyi Ecosystem?



```
1. bash
Loading Ammonite Repl...
@ case class Foo(i: Int, s0: String, s1: Seq[String])
defined class Foo
@ Foo(1234567, "I am a cow, hear me moo", Seq("I weigh twice as much as you", "a
he barbecue"))
res1: Foo = Foo(
  1234567,
  "I am a cow, hear me moo",
  List("I weigh twice as much as you", "and I look good on the barbecue")
)
@ val r = (1 until 1000).view.filter(n => n % 3 == 0 || n % 5 == 0).sum
r: Int = 233168
@ def factors(n: Long): List[Long] = {
  (2 to math.sqrt(n).toInt)
    .find(n % _ == 0)
    .map(i => i.toLong :: factors(n / i)).getOrElse(List(n))
}
defined function factors
@ factors(600851475143L).last
res4: Long = 6857L
@
```

Ammonite, a fancy Scala REPL: with syntax-highlighted multi-line input, syntax-highlighted nicely-formatted multi-line output, and many other quality of life features

1.7 What is the com.lihaoyi Ecosystem?

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: *Minimal Testing Library*

(2014) uPickle: *JSON & Binary Serialization Library*

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: *HTTP Request Library*

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: *CLI Argument Parsing Library*

```
def log(foo: String)
  (implicit line: sourcecode.Line,
   file: sourcecode.File) = {

  println(
    s"${file.value}:${line.value} $foo"
  )
}

log("Foooooo")
// src/test/scala/Tests.scala:86 Foooooo
```

Sourcecode: implicits that provide source-level information such as line numbers and file names of a callsite. Great for things like logging libraries, letting you immediately see where in the codebase a logging message is coming from rather than needing to grep the message string

1.8 What is the com.lihaoyi Ecosystem?

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: *Minimal Testing Library*

(2014) uPickle: *JSON & Binary Serialization Library*

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: *HTTP Request Library*

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: *CLI Argument Parsing Library*

```
object foo extends ScalaModule {  
  def scalaVersion = "2.13.8"  
  def ivyDeps = Agg(  
    ivy"com.lihaoyi::scalatags:0.8.2",  
    ivy"com.lihaoyi::mainargs:0.4.0"  
  )  
}
```

```
$ ./mill foo.compile  
compiling 1 Scala source...
```

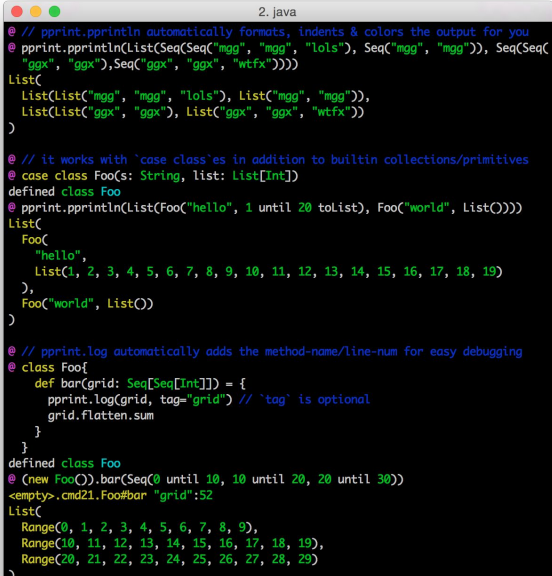
```
$ ./mill foo.assembly
```

```
$ ./out/foo/assembly.dest/out.jar --text hello  
<h1>hello</h1>
```

Mill: a build tool that lets you write normal Scala objects, traits, and method definitions to build/test/run your project in an incremental and parallel fashion

1.9 What is the com.lihaoyi Ecosystem?

- (2012) Scalatags: *HTML Generation Library*
- (2014) FastParse: *Fast Parser Combinators*
- (2014) uTest: *Minimal Testing Library*
- (2014) uPickle: *JSON & Binary Serialization Library*
- (2015) Ammonite: *Fancy Scala REPL*
- (2016) Sourcecode: *source file-name/lines/etc.*
- (2017) Mill: *Better Scala Build Tool*
- (2017) PPrint: *Pretty-Printing Library***
- (2018) OS-Lib: *Filesystem/Subprocess Library*
- (2018) Requests-Scala: *HTTP Request Library*
- (2018) Cask: *HTTP Micro-Framework*
- (2020) MainArgs: *CLI Argument Parsing Library*



```
2. java
@ // pprint.println automatically formats, indents & colors the output for you
@ pprint.println(List(Seq(Seq("mgg", "mgg", "lols"), Seq("mgg", "mgg")), Seq(Seq(
  "ggx", "ggx"),Seq("ggx", "ggx", "wtfx"))))
List(
  List(List("mgg", "mgg", "lols"), List("mgg", "mgg")),
  List(List("ggx", "ggx"), List("ggx", "ggx", "wtfx"))
)

@ // it works with `case class`es in addition to builtin collections/primitives
@ case class Foo(s: String, list: List[Int])
defined class Foo
@ pprint.println(List(Foo("hello", 1 until 20 toList), Foo("world", List())))
List(
  Foo(
    "hello",
    List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
  ),
  Foo("world", List())
)

@ // pprint.log automatically adds the method-name/line-num for easy debugging
@ class Foo{
  def bar(grid: Seq[Seq[Int]]) = {
    pprint.log(grid, tag="grid") // `tag` is optional
    grid.flatten.sum
  }
}
defined class Foo
@ (new Foo()).bar(Seq(0 until 10, 10 until 20, 20 until 30))
<empty>.cmd21.Foo#bar "grid":52
List(
  Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9),
  Range(10, 11, 12, 13, 14, 15, 16, 17, 18, 19),
  Range(20, 21, 22, 23, 24, 25, 26, 27, 28, 29)
)
```

PPrint: the output formatting logic of Ammonite extracted into a standalone library, so you can easily add syntax highlighting and nice multi-line formatting for e.g. logging in your own project

(2012) Scalatags: *HTML Generation Library*
(2014) FastParse: *Fast Parser Combinators*
(2014) uTest: *Minimal Testing Library*
(2014) uPickle: *JSON & Binary Serialization Library*
(2015) Ammonite: *Fancy Scala REPL*
(2016) Sourcecode: *source file-name/lines/etc.*
(2017) Mill: *Better Scala Build Tool*
(2017) PPrint: *Pretty-Printing Library*
(2018) OS-Lib: *Filesystem/Subprocess Library*
(2018) Requests-Scala: *HTTP Request Library*
(2018) Cask: *HTTP Micro-Framework*
(2020) MainArgs: *CLI Argument Parsing Library*

1.10 What is the com.lihaoyi Ecosystem?

```
val wd = os.pwd / "out" / "splash"

// Read/write files
os.write(wd / "file.txt", "hello")
os.read(wd / "file.txt") ==> "hello"

// Perform filesystem operations
os.copy(wd / "file.txt", wd / "new.txt")

// Invoke subprocesses
val invoked = os
  .proc("cat", wd / "file.txt", wd / "new.txt")
  .call(cwd = wd)
```

OS-Lib: a library to make it very easy to manipulate files and subprocesses

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: *Minimal Testing Library*

(2014) uPickle: *JSON & Binary Serialization Library*

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: HTTP Request Library

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: *CLI Argument Parsing Library*

1.11 What is the com.lihaoyi Ecosystem?

```
val r = requests.get(
  "https://api.github.com/users/lihaoyi"
)

r.statusCode
// 200

r.headers("content-type")
// Buffer("application/json; charset=utf-8")

r.text
// {"login":"lihaoyi",
//  "Id":934140,
//  "node_id":"MDQ6VXNlcjkzNDE0MA==",
//  ...
```

Requests-Scala: a library for making HTTP requests

1.12 What is the com.lihaoyi Ecosystem?

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: *Minimal Testing Library*

(2014) uPickle: *JSON & Binary Serialization Library*

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: *HTTP Request Library*

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: *CLI Argument Parsing Library*

```
object Application extends cask.MainRoutes{
  @cask.get("/")
  def hello() = {
    "Hello World!"
  }

  @cask.post("/do-thing")
  def doThing(request: cask.Request) = {
    request.text().reverse
  }

  initialize()
}
```

Cask: a small HTTP MicroFramework, letting you just annotate a few methods to turn them into a web server

1.13 What is the com.lihaoyi Ecosystem?

(2012) Scalatags: *HTML Generation Library*

(2014) FastParse: *Fast Parser Combinators*

(2014) uTest: *Minimal Testing Library*

(2014) uPickle: *JSON & Binary Serialization Library*

(2015) Ammonite: *Fancy Scala REPL*

(2016) Sourcecode: *source file-name/lines/etc.*

(2017) Mill: *Better Scala Build Tool*

(2017) PPrint: *Pretty-Printing Library*

(2018) OS-Lib: *Filesystem/Subprocess Library*

(2018) Requests-Scala: *HTTP Request Library*

(2018) Cask: *HTTP Micro-Framework*

(2020) MainArgs: CLI Argument Parsing Library

```
object Main{
  @main
  def run(foo: String,
         num: Int = 2,
         bool: Flag) = {

    println(foo * myNum + " " + bool.value)
  }

  def main(args: Array[String]): Unit =
    ParserForMethods(this).runOrExit(args)
}

$ ./mill example.hello --foo hello
hellohello false

$ ./mill example.hello --foo hello --num 3 --bool
hellohellohello true
```

MainArgs: a CLI argument parsing library, letting you take a normal Scala method and annotate it with `@main` to turn it into a proper CLI main method with – flags, optional flags, help text, and so on.

There are other libraries in the com.lihaoyi ecosystem, but these are the most notable and interesting ones. Does anyone here use any of them?

Executable Scala Pseudocode that's Easy, Boring and Fast!

1. **What is the com.lihaoyi ecosystem?**
2. What is the com.lihaoyi ecosystem not?
3. Principles of the com.lihaoyi ecosystem
4. Case Study: How Mill Makes Builds Great

We have done a quick tour of what the com.lihaoyi ecosystem is,

Executable Scala Pseudocode that's Easy, Boring and Fast!

1. What is the com.lihaoyi ecosystem?
2. **What is the com.lihaoyi ecosystem not?**
3. Principles of the com.lihaoyi ecosystem
4. Case Study: How Mill Makes Builds Great

Let's take a look at what it is not

2.1 Python Fanfic?

- (2012) Scalatags: HTML Generation Library
- (2014) FastParse: Fast Parser Combinators
- (2014) uTest: Minimal Testing Library
- (2014) uPickle: JSON & Binary Serialization Library
- (2015) Ammonite: Fancy Scala REPL
- (2016) Sourcecode: source file-name/lines/etc.
- (2017) Mill: Better Scala Build tool
- (2017) PPrint: Pretty-Printing Library
- (2018) OS-Lib: Filesystem/Subprocess Library
- (2018) Requests-Scala: HTTP Request Library
- (2018) Cask: HTTP Micro-Framework
- (2020) MainArgs: CLI Argument Parsing Library

Is the com.lihaoyi ecosystem Python fanfic, for people who want to write Python but are stuck writing Scala?

(2012) Scalatags: HTML Generation Library
(2014) FastParse: Fast Parser Combinators
(2014) uTest: Minimal Testing Library
(2014) uPickle: JSON & Binary Serialization Library
(2015) Ammonite: Fancy Scala REPL
(2016) Sourcecode: source file-name/lines/etc.
(2017) Mill: Better Scala Build tool
(2017) PPrint: Pretty-Printing Library
(2018) OS-Lib: Filesystem/Subprocess Library
(2018) Requests-Scala: HTTP Request Library
(2018) Cask: HTTP Micro-Framework
(2020) MainArgs: CLI Argument Parsing Library

2.1 Python Fanfic?

```
write(Seq(1, 2, 3))      ==> "[1,2,3]"

pprint.pprintln(Seq(1, 2, 3))

val r = requests.get(
  "https://api.github.com/users/lihaoyi"
)

@cask.get("/")
def hello() = {
  "Hello World!"
}
```

Yes, it is true that many libraries are inspired by python, or even carbon-copies of their Python equivalent. uPickle is a clone of python's pickle/json, Ammonite is a clone of iPython. PPrint is a clone of, well pprint and requests-scala is a clone of Python's requests.

(2012) Scalatags: HTML Generation Library

(2014) FastParse: Fast Parser Combinators

(2014) uTest: Minimal Testing Library

(2014) uPickle: JSON & Binary Serialization Library

(2015) Ammonite: Fancy Scala REPL

(2016) Sourcecode: source file-name/lines/etc.

(2017) Mill: Better Scala Build tool

(2017) PPrint: Pretty-Printing Library

(2018) OS-Lib: Filesystem/Subprocess Library

(2018) Requests-Scala: HTTP Request Library

(2018) Cask: HTTP Micro-Framework

(2020) MainArgs: CLI Argument Parsing Library

2.1 Python Fanfic?

```
body(  
  div(  
    h1(id="title", "This is a title"),  
    p("This is a big paragraph of text")  
  )  
)  
  
def number[$: P] =  
  P(CharIn("0-9").rep(1).!).map(_.toInt)  
  
def parens[$: P] = P("(" ~/ addSub ~ ")")  
def factor[$: P] = P(number | parens)  
  
object foo extends RootModule with ScalaModule
```

But there are a lot of libraries in com.lihaoyi that are most definitely not Pythonic. You would not find Scalatags-style HTML templating in Python, nor would you find Fastparse-style parser combinators. The example at the bottom demonstrates how Mill lets you define modules in your project build by using stackable traits. These are all theoretical possible in Python - Python does support nested method calls, operator overloading, and multiple inheritance - but it is not a Pythonic thing to do, and these libraries are definitely not something cloned from the Python ecosystem

2.2 Scala.js Libraries?

- (2012) Scalatags: HTML Generation Library
- (2014) FastParse: Fast Parser Combinators
- (2014) uTest: Minimal Testing Library
- (2014) uPickle: JSON & Binary Serialization Library
- (2015) Ammonite: Fancy Scala REPL
- (2016) Sourcecode: source file-name/lines/etc.
- (2017) Mill: Better Scala Build tool
- (2017) PPrint: Pretty-Printing Library
- (2018) OS-Lib: Filesystem/Subprocess Library
- (2018) Requests-Scala: HTTP Request Library
- (2018) Cask: HTTP Micro-Framework
- (2020) MainArgs: CLI Argument Parsing Library

How about Scala.js? Is the com.lihaoyi ecosystem just for use in Scala.js?

2.2 Scala.js Libraries?

(2012) Scalatags: HTML Generation Library

(2014) FastParse: Fast Parser Combinators

(2014) uTest: Minimal Testing Library

(2014) uPickle: JSON & Binary Serialization Library

(2015) Ammonite: Fancy Scala REPL

(2016) Sourcecode: source file-name/lines/etc.

(2017) Mill: Better Scala Build tool

(2017) PPrint: Pretty-Printing Library

(2018) OS-Lib: Filesystem/Subprocess Library

(2018) Requests-Scala: HTTP Request Library

(2018) Cask: HTTP Micro-Framework

(2020) MainArgs: CLI Argument Parsing Library

In the past, yes: the initial impetus for many of these libraries was to provide a Scala.js ecosystem, at a time when there was none.

In fact, for many years, libraries like Scalatags, uTest, and uPickle were the **only** ways to do HTML templating, testing and serializing data in Scala.js

2.2 Scala.js Libraries?

- (2012) Scalatags: HTML Generation Library
- (2014) FastParse: Fast Parser Combinators
- (2014) uTest: Minimal Testing Library
- (2014) uPickle: JSON & Binary Serialization Library
- (2015) Ammonite: Fancy Scala REPL
- (2016) Sourcecode: source file-name/lines/etc.
- (2017) Mill: Better Scala Build tool
- (2017) PPrint: Pretty-Printing Library
- (2018) OS-Lib: Filesystem/Subprocess Library
- (2018) Requests-Scala: HTTP Request Library
- (2018) Cask: HTTP Micro-Framework
- (2020) MainArgs: CLI Argument Parsing Library

But that's no longer the case. While com.lihaoyi continues to support Scala.js whenever possible, many - or even most - of the libraries are not Scala.js specific. Ammonite, OS-Lib, MainArgs, etc. don't even work on Scala.js. Even for the original set of libraries, their importance to Scala.js has also waned over, as more of the broader ecosystem has also picked up support for the Scala.js platform.

Even though com.lihaoyi started off as *the* Scala.js ecosystem, both it and Scala.js has since grown beyond that

Executable Scala Pseudocode that's Easy, Boring and Fast!

1. What is the com.lihaoyi ecosystem?
2. **What is the com.lihaoyi ecosystem not?**
3. Principles of the com.lihaoyi ecosystem
4. Case Study: How Mill Makes Builds Great

Executable Scala Pseudocode that's Easy, Boring and Fast!

1. What is the com.lihaoyi ecosystem?
2. What is the com.lihaoyi ecosystem not?
3. **Principles of the com.lihaoyi ecosystem**
4. Case Study: How Mill Makes Builds Great

3. Principles of the com.lihaoyi ecosystem

Executable Scala Pseudocode that's Easy, Boring and Fast!

These principles have actually been hiding in plain sight, from the first slide. To summarize the com.lihaoyi ecosystem in one line, it is to use Scala as Executable Pseudocode that's Easy, Boring, and Fast!

This isn't just a catchy tagline. Let's break it down.

3.1 Principles of the com.lihaoyi ecosystem

Executable Scala Pseudocode that's Easy, Boring and Fast!

First, executable pseudocode. Pseudocode is code that may not actually run, but matches closely with your mental model of the problem. This makes it excellent for exploring a problem space or discussing things with other people. “Executable” pseudocode means you can take what it's your head, write it down, and directly run it without first translating it into a “real” programming language.

Executable pseudocode is great not just because it's easy to write, but because it's easy to read: someone who comes to your quick script 10 years in future who may not even know the same programming language could guess what your code is doing, and guess correctly.

When you say “Executable Pseudocode”, people often think of Python.

Scala is as concise as Python. In many ways, it's even more concise. It's better at the “Executable” part: it executes faster, so you don't need to rewrite parts of your program in C when you need to productionize it. It has better tooling on the JVM, with a good story for deployment and distribution. It has compile-time checks, so is safer to execute without blowing up at runtime due to dumb typos.

Why not Scala?

3.1.1 Executable Pseudocode: HTTP Requests

Imagine for a moment you are in an interview. Maybe you just got laid off, maybe you're just exploring the job market. You are in a small room with someone who just asked you to write a web crawler or a github issue migrator or some other such programming challenge.

You've sketched out the skeleton of your solution, and it's time to make a HTTP request. It's just a whiteboard, so it doesn't need to actually work, but you'd just write pseudocode: what makes the most sense both to *you* and to the interviewer. What do you write?

3.1.1 Executable Pseudocode: HTTP Requests

```
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.Behaviors

implicit val system =
  ActorSystem(Behaviors.empty, "SingleRequest")

implicit val executionContext = system.executionContext
```

Do you start off by importing a bunch of stuff, defining an actorsystem, configuring it's behaviors, and then extracting it's executioncontext?

3.1.1 Executable Pseudocode: HTTP Requests

```
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.Behaviors
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._

implicit val system =
  ActorSystem(Behaviors.empty, "SingleRequest")

implicit val executionContext = system.executionContext

val responseFuture = Http()
  .singleRequest(HttpRequest(uri = "http://akka.io"))
```

Then importing a HTTP DSL, that you can use to make a HTTP request?

3.1.1 Executable Pseudocode: HTTP Requests

```
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.Behaviors
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import scala.util.{ Failure, Success }

implicit val system =
  ActorSystem(Behaviors.empty, "SingleRequest")

implicit val executionContext = system.executionContext

val responseFuture = Http()
  .singleRequest(HttpRequest(uri = "http://akka.io"))

responseFuture.onComplete {
  case Success(res) => println(res)
  case Failure(_)   => sys.error("something wrong")
}
```

And then importing Success and Failure cases and then finally doing something with the result of the request?

Would you write all this on whiteboard? Of course not. All this stuff is irrelevant to the initial problem at hand. Sure you might need to configure it later, tweaking thread counts or sharing execution contexts and so on, but that's not essential to the problem we want to discuss. If this is not what you would write on the whiteboard, what would you write?

3.1.1 Executable Pseudocode: HTTP Requests

```
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.Behaviors
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import scala.util.{ Failure, Success }

val res = requests.get("https://akka.io")

println(res)

implicit val system =
  ActorSystem(Behaviors.empty, "SingleRequest")

implicit val executionContext = system.executionContext

val responseFuture = Http()
  .singleRequest(HttpRequest(uri = "http://akka.io"))

responseFuture.onComplete {
  case Success(res) => println(res)
  case Failure(_)   => sys.error("something wrong")
}
```

Probably you would write something like this. You need to make a GET request, so call `requests.get` and pass it the URL. Then you'd use the response. End of story.

This is executable code, using `requests-scala`.

3.1.2 Executable Pseudocode: CLI Entrypoint

Imagine you just hired a new intern, and their first job is to write a small command line tool. You're trying to explain what this command line tool is going to do, what flags it will take, on a piece of paper at your intern's desk. What do you write on this piece of paper?

3.1.2 Executable Pseudocode: CLI Entrypoint

```
case class Config(foo: String = null,  
                 num: Int = 2,  
                 bool: Boolean = false)
```

You might start off defining a Config case class defining the inputs.

3.1.2 Executable Pseudocode: CLI Entrypoint

```
case class Config(foo: String = null,  
                 num: Int = 2,  
                 bool: Boolean = false)  
val builder = OParser.builder[Config]
```

Would you follow that up by creating a builder that is a parser builder for that config?

3.1.2 Executable Pseudocode: CLI Entrypoint

```
case class Config(foo: String = null,
                 num: Int = 2,
                 bool: Boolean = false)
val builder = OParser.builder[Config]
val parser1 = {
  import builder._
  OParser.sequence(
    programName("run"),
    opt[String]("foo")
      .required()
      .action((x, c) => c.copy(foo = x)),
    opt[Int]("num")
      .action((x, c) => c.copy(num = x)),
    opt[Unit]("bool")
      .action((_, c) => c.copy(bool = true))
  )
}
```

And then define another lower-case parser, separate from the first capital-P parser, that then imports the builder, then use the capital-P parser again to do a whole bunch of stuff that mostly duplicates the code you wrote already in the config case class?

3.1.2 Executable Pseudocode: CLI Entrypoint

```
case class Config(foo: String = null,
                 num: Int = 2,
                 bool: Boolean = false)
val builder = OParser.builder[Config]
val parser1 = {
  import builder._
  OParser.sequence(
    programName("run"),
    opt[String]("foo")
      .required()
      .action((x, c) => c.copy(foo = x)),
    opt[Int]("num")
      .action((x, c) => c.copy(num = x)),
    opt[Unit]("bool")
      .action((_, c) => c.copy(bool = true))
  )
}
val parsed = OParser.parse(parser1, args, Config())
for(Config(foo, num, bool) <- parsed){
  println(foo * num + " " + bool.value)
}
```

And then finally using the capital-P parser to parse the lowercase-p parser along with the arguments, and a dummy config value, to finally parse out the values you can use

Now, this all makes sense from a software engineering perspective: builder patterns, functional combinators, separation of concerns, etc.. But many of those things are “internal” details: relevant to the *implementation* of the CLI parsing library, but irrelevant to the task at hand. That’s why you’re not going to write all this down on the piece of paper you give your intern to use as a reference for what they need to do.

So what would you write?

3.1.2 Executable Pseudocode: CLI Entrypoint

```
case class Config(foo: String = null,
                  num: Int = 2,
                  bool: Boolean = false)
def run

val builder = OParser.builder[Config]
val parser1 = {
  import builder._
  OParser.sequence(
    programName("run"),
    opt[String]("foo")
      .required()
      .action((x, c) => c.copy(foo = x)),
    opt[Int]("num")
      .action((x, c) => c.copy(num = x)),
    opt[Unit]("bool")
      .action((_, c) => c.copy(bool = true))
  )
}
val parsed = OParser.parse(parser1, args, Config())
for(Config(foo, num, bool) <- parsed){
  println(foo * num + " " + bool.value)
}
```

If you're like me, since you're trying to explain what the CLI tool's main method is meant to do, you would start by defining the method. That means "def", followed by a name. Maybe "main" or "run"

3.1.2 Executable Pseudocode: CLI Entrypoint

```
case class Config(foo: String = null,
                 num: Int = 2,
                 bool: Boolean = false)
def run(foo: String, num: Int = 2, bool: Flag)

val builder = OParser.builder[Config]
val parser1 = {
  import builder._
  OParser.sequence(
    programName("run"),
    opt[String]("foo")
      .required()
      .action((x, c) => c.copy(foo = x)),
    opt[Int]("num")
      .action((x, c) => c.copy(num = x)),
    opt[Unit]("bool")
      .action((_, c) => c.copy(bool = true))
  )
}
val parsed = OParser.parse(parser1, args, Config())
for(Config(foo, num, bool) <- parsed){
  println(foo * num + " " + bool.value)
}
```

You'd then define what the arguments to the CLI tool are, what their types are, and if any of them are optional and take default values

3.1.2 Executable Pseudocode: CLI Entrypoint

```
case class Config(foo: String = null,
                 num: Int = 2,
                 bool: Boolean = false)

val builder = OParser.builder[Config]
val parser1 = {
  import builder._
  OParser.sequence(
    programName("run"),
    opt[String]("foo")
      .required()
      .action((x, c) => c.copy(foo = x)),
    opt[Int]("num")
      .action((x, c) => c.copy(num = x)),
    opt[Unit]("bool")
      .action((_, c) => c.copy(bool = true))
  )
}

val parsed = OParser.parse(parser1, args, Config())
for(Config(foo, num, bool) <- parsed){
  println(foo * num + " " + bool.value)
}
```

```
def run(foo: String, num: Int = 2, bool: Flag) = {
  println(foo * num + " " + bool.value)
}
```

And then you'd use it to do the thing you want to do. End of story.

3.1.2 Executable Pseudocode: CLI Entrypoint

```
case class Config(foo: String = null,
                 num: Int = 2,
                 bool: Boolean = false)
val builder = OParser.builder[Config]
val parser1 = {
  import builder._
  OParser.sequence(
    programName("run"),
    opt[String]("foo")
      .required()
      .action((x, c) => c.copy(foo = x)),
    opt[Int]("num")
      .action((x, c) => c.copy(num = x)),
    opt[Unit]("bool")
      .action((_, c) => c.copy(bool = true))
  )
}
val parsed = OParser.parse(parser1, args, Config())
for(Config(foo, num, bool) <- parsed){
  println(foo * num + " " + bool.value)
}
```

```
@main
def run(foo: String, num: Int = 2, bool: Flag) = {
  println(foo * num + " " + bool.value)
}

ParserForMethods(this).runOrExit(args)
```

That's basically what the `com.lihaoyi.mainargs` library lets you do.

Sure you have to annotate it with `@main`, and you have to at some point call something to pass the args from the *real* main method to it, but overall it lets you take pseudocode you may sketch out on a piece of paper during your intern's onboarding, annotate it, and turn it into executable code. That's executable pseudocode.

3.1.3 Executable Pseudocode:

```
case class Config(foo: String = null,
                 num: Int = 2,
                 bool: Boolean = false)
val builder = OParser.builder[Config]
val parser1 = {
  import builder._
  OParser.sequence(
    programName("run"),
    opt[String]("foo")
      .required()
      .action((x, c) => c.copy(foo = x)),
    opt[Int]("num")
      .action((x, c) => c.copy(num = x)),
    opt[Unit]("bool")
      .action((_, c) => c.copy(bool = true))
  )
}
val parsed = OParser.parse(parser1, args, Config())
for(Config(foo, num, bool) <- parsed){
  println(foo * num + " " + bool.value)
}
```

```
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.Behaviors
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import scala.util.{Failure, Success }

implicit val system =
  ActorSystem(Behaviors.empty, "SingleRequest")

implicit val executionContext = system.executionContext

val responseFuture = Http()
  .singleRequest(HttpRequest(uri = "http://akka.io"))

responseFuture.onComplete {
  case Success(res) => println(res)
  case Failure(_)   => sys.error("something wrong")
}
```

These examples perhaps highlight the traditional problem with Scala as Executable Pseudocode. The language is fine, and can be made concise and understandable. But the libraries have traditionally been verbose and clunky.

So while snippets of pure-scala on slides look great, defining a linked list or fibonacci sequence or something, any Scala that actually has to do something is just as ugly and verbose as Java code! People like making fun of Java for being verbose, but ArgParse4j would actually look not much uglier than the Sconfig code on the left, and Apache HTTP Client would look not much more verbose than the Akka HTTP code on the right.

It's a different style of ugly, with a different set of design patterns making it verbose - implicits and combinators rather than builder patterns and visitors - but ugly and verbose all the same. Scala needs to do better

3.1.4 Executable Pseudocode in the com.lihaoyi libraries

```
// mainargs
@mainargs.main
def run(foo: String, num: Int = 2, bool: Flag) = {
  println(foo * num + " " + bool.value)
}

// cask
@cask.get("/user/:userName")
def showUserProfile(userName: String) = {
  s"User $userName"
}

// uPickle
upickle.default.write(Seq(1, 2, 3))

// requests
val resp = requests.get("http://akka.io")

// os-lib
os.proc("grep", "Data")
  .call(
    stdin = resp,
    stdout = os.pwd / "Out.txt"
  )

// Mill
def lineCount = T{
  allSourceFiles()
    .map(f => os.read.lines(f.path).size)
    .sum
}
```

com.lihaoyi libraries all look very different from traditional Scala libraries, as shown by these examples. They also look very similar to each other. By and large, they revolve around Scala method defs and Scala method calls, with a sprinkling of annotations for customization

These all match more or less how a developer would think of the problem:

1. A main method taking CLI arguments is a method that takes parameters.
2. HTTP get and post endpoints are methods that take parameters.
3. Making HTTP requests or subprocess operations are just method calls that you pass parameters

Note what we *don't* see here: DSLs, implicits, inheritance, builder patterns, config objects, wrappers, adapters, registries, visitors, etc. Even *imports* are minimized

That's not to say that these things aren't used: all of these libraries make heavy use of the those design patterns in their implementation. But they make sure to wrap them up nicely such that for users who don't *want* to care about these implementation detail, they don't *need* to. A typical user gets reasonable defaults and a simple API, letting them write code almost as simple as the code they write on a whiteboard. Advanced customization is available, but not forced upon those users who do not need it.

3.1 Principles of the com.lihaoyi ecosystem

Executable Scala Pseudocode that's Easy, Boring and Fast!

3.2 Principles of the com.lihaoyi ecosystem

Executable Scala Pseudocode that's **Easy**, Boring and Fast!

The next point to discuss is Easy.

“Simple Not Easy” is a meme that originated in a talk by the author of the Clojure programming language. What it means “Simplicity” is more about fundamental simplicity, that the whole system from top-to-bottom is made of a small number of orthogonal concepts. “Easy” is more about familiarity, about polish, and molding your product closely to a specific use case.

Some people say Scala is already very Simple. Martin Odersky likes saying that Scala has a very small grammar, and he's right. Scala does have a relatively small number of orthogonal features. It's already Simple. What's missing from Scala is *Easy*

3.2.1 Easy, not Simple

```
def string[$: P] =
  P( space ~ "\"" ~/ (strChars | escape).rep.! ~ "\"" )
  .map(Js.Str.apply)

def array[$: P] =
  P( "[" ~/ jsonExpr.rep(sep=",") ~/ space ~ "]" )
  .map(Js.Arr(_:_*))

def pair[$: P] = P( string.map(_.value) ~/ ":" ~/ jsonExpr )

def obj[$: P] =
  P( "{" ~/ pair.rep(sep=",") ~/ space ~ "}" )
  .map(Js.Obj(_:_*))
```

Consider the following FastParse code, a snippet from the example JSON parser. It might look a bit cryptic at first, but it's not that different from other Scala parser combinator libraries, e.g. the one that used to be in the standard library. `string` is made of a quote followed by string characters followed by a closing quote. `array` is made of an open bracket followed by comma-separated expressions followed by a closing bracket. And so on. There are some details involved: `.!`'s to capture things, `.map`'s to turn them into case classes, some spaces sprinkled around so you can parse JSON that's not minified. But overall it still closely matches how you would think of a formal grammar

3.2.1 Easy, not Simple

```
def string[$: P] =
  P( space ~ "\"" ~/ (strChars | escape).rep.! ~ "\"" )
  .map(Js.Str.apply)

def array[$: P] =
  P( "[" ~/ jsonExpr.rep(sep=",") ~ space ~ "]" )
  .map(Js.Arr(_:_*))

def pair[$: P] = P( string.map(_._value) ~/ ":" ~/ jsonExpr )

def obj[$: P] =
  P( "{" ~/ pair.rep(sep=",") ~ space ~ "}" )
  .map(Js.Obj(_:_*))

@ fastparse.parse("""["1", "2", ]""", array(_))
Failure at index: 11, found: ...""
expected: (obj | array | string | true | false | null | number)
```

Just by writing what's effectively the language grammar, FastParse doesn't just give you a high-performance parser, but it also gives you excellent error reporting, for free. For example, if I forget an entry in my array and put a closing bracket after the comma, I immediately get an error message with the offset in the string, what characters it found, and what it was expecting to find.

Simple parser definition, fast performance (~300x faster than scala-parser-combinators), good error messages. That is exactly what someone parsing something wants. That's **Easy**

3.2.1 Easy, not Simple

```
def string[$: P] =
  P( space ~ "\"" ~/ (strChars | escape).rep.! ~ "\"" )
  .map(Js.Str.apply)

def array[$: P] =
  P( "[" ~/ jsonExpr.rep(sep=",") ~ space ~ "]" )
  .map(Js.Arr(_:_*))

def pair[$: P] = P( string.map(_._value) ~/ ":" ~/ jsonExpr )

def obj[$: P] =
  P( "{" ~/ pair.rep(sep=",") ~ space ~ "}" )
  .map(Js.Obj(_:_*))

@ fastparse.parse("""["1", "2", ]""", array(_))
Failure at index: 11, found: "...]"
expected: (obj | array | string | true | false | null | number)
```

```
final class ParsingRun[+T](
  val input: ParserInput,
  val startIndex: Int,
  val originalParser: ParsingRun[_] => ParsingRun[_],
  val traceIndex: Int,
  val instrument: Instrument,
  // Mutable vars below:
  var terminalMsgs: Msgs,
  var aggregateMsgs: Msgs,
  var shortMsg: Msgs,
  var lastFailureMsg: Msgs,
  var failureStack: List[(String, Int)],
  var isSuccess: Boolean,
  var logDepth: Int,
  var index: Int,
  var cut: Boolean,
  var successValue: Any,
  var verboseFailures: Boolean,
  var noDropBuffer: Boolean,
  val misc: collection.mutable.Map[Any, Any])
```

Internally, Fastparse is not simple. On the right is the data structure representing the internal workings of FastParse: the parsing, backtracking, error reporting, etc.

3.2.1 Easy, not Simple

```
def string[$: P] =
  P( space ~ "\"" ~/ (strChars | escape).rep.! ~ "\"" )
  .map(Js.Str.apply)

def array[$: P] =
  P( "[" ~/ jsonExpr.rep(sep=",") ~ space ~ "]" )
  .map(Js.Arr(_:_*))

def pair[$: P] = P( string.map(_._value) ~/ ":" ~/ jsonExpr )

def obj[$: P] =
  P( "{" ~/ pair.rep(sep=",") ~ space ~ "}" )
  .map(Js.Obj(_:_*))

@ fastparse.parse("""["1", "2", ]""", array(_))
Failure at index: 11, found: "...)"
expected: (obj | array | string | true | false | null | number)
```

```
final class ParsingRun[+T](
  val input: ParserInput,
  val startIndex: Int,
  val originalParser: ParsingRun[_] => ParsingRun[_],
  val traceIndex: Int,
  val instrument: Instrument,
  // Mutable vars below:
  var terminalMsgs: Msgs,
  var aggregateMsgs: Msgs,
  var shortMsg: Msgs,
  var lastFailureMsg: Msgs,
  var failureStack: List[(String, Int)],
  var isSuccess: Boolean,
  var logDepth: Int,
  var index: Int,
  var cut: Boolean,
  var successValue: Any,
  var verboseFailures: Boolean,
  var noDropBuffer: Boolean,
  val misc: collection.mutable.Map[Any, Any])
```

Everyone here knows that dealing with mutable state is not simple. FastParse internally uses tons of mutable state. I've highlighted it in red just so you can't miss it. FastParse internals are not simple, and are actually really confusing, precisely due to all this mutable state. To top things off, all this is hidden under a thick layer of Macros, implemented twice for Scala 2 and Scala 3.

If you watched Daniel's talk on micro-optimization yesterday, FastParse uses all the tricks internally, and many more, while still presenting a facade of a pure-functional combinator API

However, this complexity is not for nothing: it is precisely this complexity that makes FastParse so much faster than the alternatives, 300x faster than Scala Parser Combinators, while providing such good error messages. FastParse is not **Simple**, but using FastParse is definitely **Easy**

3.2.2 Easy, not Simple

```
@main
def run(foo: String, num: Int = 2, bool: Flag) = {
  println(foo * num + " " + bool.value)
}

ParserForMethods(this).runOrExit(args)
```

Next, let's revisit the mainargs example I showed earlier: we annotate a method with `@main`, and then call `ParserForMethods.runOrExit`. That's easy enough. But what's happening under the hood?

3.2.2 Easy, not Simple

```
@main
def run(foo: String, num: Int = 2, bool: Flag) = {
  println(foo * num + " " + bool.value)
}
```

```
ParserForMethods(this).runOrExit(args)
```

```
def run(foo: String, num: Int = 2, bool: Flag) = {
  println(foo * num + " " + bool.value)
}

ParserForMethods(
  Seq(
    MainData(
      name = "run",
      argSigs0 = Seq(
        ArgSig[String](Some("foo"), default = None),
        ArgSig[Int](Some("num"), default = Some(2)),
        ArgSig[Flag](Some("bool"), default = None),
      ),
      invoke = {
        case Seq(foo: String, num: Int, bool: Boolean) =>
          this.run(foo, num, bool)
      }
    )
  )
).runOrExit(args)
```

Under the hood, `ParserForMethods` expands into the following code (slightly simplified). As you can see, it's a relatively verbose, but straightforward data structure: the name of the method, a list of `ArgSig`'s with types and metadata, an `invoke` callback which lets you take the parsed arguments and passes them to the actual `run` method we want to execute

3.2.2 Easy, not Simple

```
val builder = OParser.builder[Config]
val parser1 = {
  import builder._
  OParser.sequence(
    programName("run"),
    opt[String]("foo")
      .required()
      .action((x, c) => c.copy(foo = x)),
    opt[Int]("num")
      .action((x, c) => c.copy(num = x)),
    opt[Unit]("bool")
      .action((_, c) => c.copy(bool = true))
  )
}
val parsed = OParser.parse(parser1, args, Config())
for(Config(foo, num, bool) <- parsed){
  println(foo * num + " " + bool.value)
}
```

```
def run(foo: String, num: Int = 2, bool: Flag) = {
  println(foo * num + " " + bool.value)
}

ParserForMethods(
  Seq(
    MainData(
      name = "run",
      argSigs0 = Seq(
        ArgSig[String](Some("foo"), default = None),
        ArgSig[Int](Some("num"), default = Some(2)),
        ArgSig[Flag](Some("bool"), default = None),
      ),
      invoke = {
        case Seq(foo: String, num: Int, bool: Boolean) =>
          this.run(foo, num, bool)
      }
    )
  )
).runOrExit(args)
```

In fact, if you look at the generated code for mainargs on the right, it is just as complicated as the example code from Scott I showed you earlier, on the left. It's actually almost exactly the same data structure, just spelled a bit differently! Mainargs' value is not being simpler than the alternative; if anything, these additional macro transformations make it even less simple. The point of mainargs is to make things *easy*, which means we bundle up all this verbose data structure definition into a thin macro facade that does what a user probably wants

3.2 Principles of the com.lihaoyi ecosystem

Executable Scala Pseudocode that's **Easy**, Boring and Fast!

Those two examples illustrate what I mean by *Easy*: going the extra mile, taking on extra complexity, to fit closely with what a user is already familiar with and what they want to do. Even if that means making the library less simple: with mutable state, macros, etc.

com.lihaoyi libraries aim to be easy, not simple.

3.3 Principles of the com.lihaoyi ecosystem

Executable Scala Pseudocode that's Easy, **Boring** and Fast!

Next, let's look at *Boring*

3.3 Boring Scala

A lot of Scala projects have traditionally been extremely ambitious.

3.3 Boring Scala

Distributed Computing Architectures of the Future

Novel Functional Programming Paradigms

Creative DSLs

Research-worthy Innovations

3.3 Boring Scala

Distributed Computing Architectures of the Future

Novel Functional Programming Paradigms

Creative DSLs

Research-worthy Innovations

JSON/Binary Serialization

HTTP Clients/Servers

HTML Generation

Parsing

Filesystem/Subprocess Operations

Pretty-printing

CLI Argument Parsing

Build Tooling

When compared to the com.lihaoyi projects, there's a stark difference: the com.lihaoyi projects are **boring**!

While the projects on the left are the kind you may give keynote conference presentations about, the projects on the right are the kind you would add to your build file, use, and forget about. And that's intentional!

3.3 Boring Scala

Distributed Computing Architectures of the Future

Novel Functional Programming Paradigms

Creative DSLs

Research-worthy Innovations

JSON/Binary Serialization

HTTP Clients/Servers

HTML Generation

Parsing

Filesystem/Subprocess Operations

Pretty-printing

CLI Argument Parsing

Build Tooling



One way I like thinking about this is that traditional Scala projects have often been Cathedrals: huge efforts, aesthetically beautiful, doing amazing things nobody thought was possible.

But being a cathedral also has its downsides. If I want a small house to stay in, or a bridge, or a grain silo, having a huge cathedral is a poor fit. You also can't mix two cathedrals together.

That's always been the problem with Scala: if you want super-high-concurrency low-latency distributed computation, then Akka is perfect. If you want to write pure-functional referentially-transparent code, then one of the FP frameworks is a great fit. If you want anything else, you're out of luck. And you can't mix these options: you're not going to use Akka in your SBT build definition, or use Cats-Effect in your Akka streams, etc.

3.3 Boring Scala

Distributed Computing Architectures of the Future

Novel Functional Programming Paradigms

Creative DSLs

Research-worthy Innovations

JSON/Binary Serialization

HTTP Clients/Servers

HTML Generation

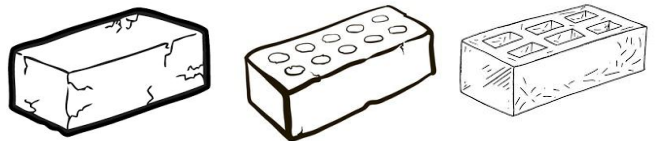
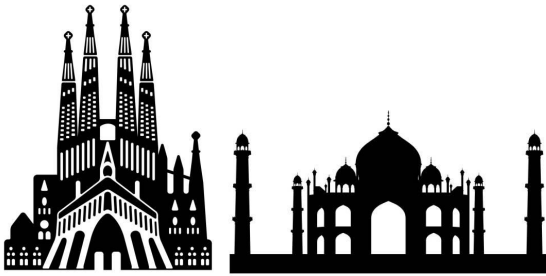
Parsing

Filesystem/Subprocess Operations

Pretty-printing

CLI Argument Parsing

Build Tooling



The `com.lihaoyi` libraries aim to be bricks. Boring, uninteresting, but solid and reliable. And you're *meant* to combine them together to build things, bricks don't have an opinion on what you are building. Whether you're building a house or a bridge or a grain silo, or even another cathedral, you can use bricks.

`com.lihaoyi` libraries are meant to be usable everywhere. You can use Scalatags to render HTML in your personal blog/static-site generator, in your Akka/Play backend server, or you can use it to render HTML in HTTP4S. You can use the Ammonite REPL to test your Akka code or your Cats code or to test your ZIO code. You can use uPickle to serialize data in your Mill build file, in your one-off research project or you can use it for your 10,000 QPS API server.

3.3 Boring Scala

Distributed Computing Architectures of the Future

Novel Functional Programming Paradigms

Creative DSLs

Research-worthy Innovations

JSON/Binary Serialization

HTTP Clients/Servers

HTML Generation

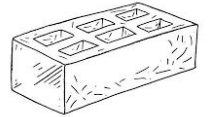
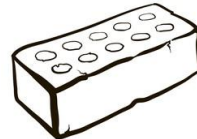
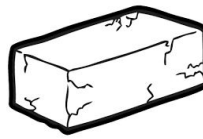
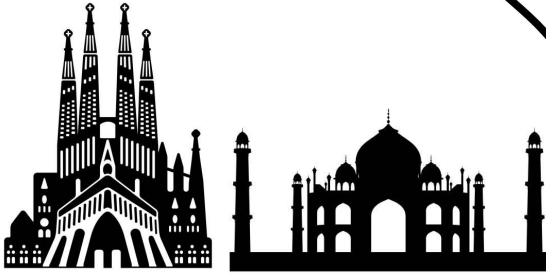
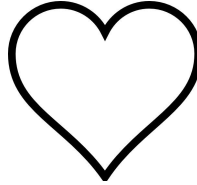
Parsing

Filesystem/Subprocess Operations

Pretty-printing

CLI Argument Parsing

Build Tooling



com.lihaoyi libraries don't care where they are used, and aim to be friendly with everyone and integrate nicely into whatever framework or style you may be using. They aim to be bricks: boring, to be used without comment or controversy, quietly supporting some of the load or complexity while staying out of the limelight

3.3 Principles of the com.lihaoyi ecosystem

Executable Scala Pseudocode that's Easy, **Boring** and Fast!

3.4 Principles of the com.lihaoyi ecosystem

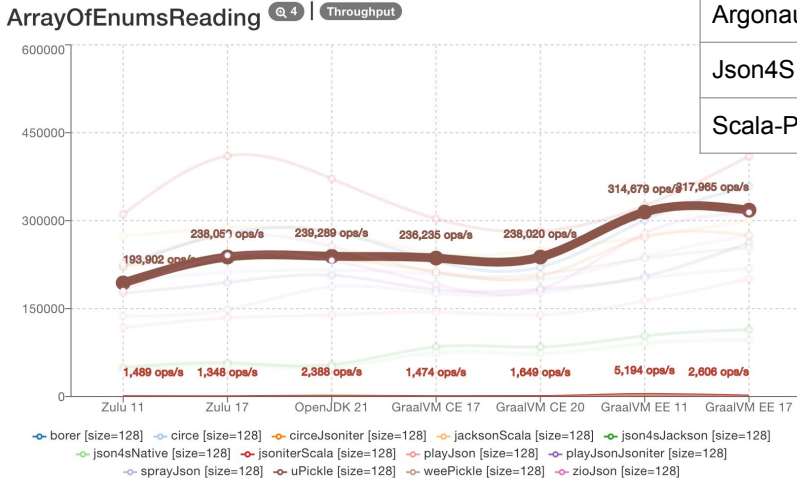
Executable Scala Pseudocode that's Easy, Boring and **Fast!**

The last point to touch on, is “Fast”. The com.lihaoyi ecosystems aim to be Fast.

Not the *Fastest*. There will always be someone willing to run one more profile, implement one more optimization, bash one more bit. But JVM performance is great. People build huge systems in Python and Ruby, that are reasonably snappy, despite the languages and runtimes being 1-2 orders of magnitude slower than bytecode running on the JVM. Apart from some core limitations like memory footprint and warmup time, Scala running on the JVM has no excuse to be slow

3.4.2 Aiming for Second Best

Parser	Parses/60s
Circe	332
play-json	227
Fastparse JSON	160
Argonaut	149
Json4S	101
Scala-Parser-Comb...	0.9



Template Engine	Renders/60s
Scalatags	7436041
scala-xml	3794707
Twirl	1902274
Scalate-Mustache	500975
Scalate-Jade	396224

Here is a rough collection of benchmarks that illustrates this philosophy:

1. Bottom left, we can see uPickle’s rough performance, not at the top of the benchmark, but definitely in the upper half among other JSON libraries
2. Top right, we can see the performance of Fastparse’s example JSON parser. It is competitive to hand-crafted hand-optimized parsers, even though it’s not the top. Notably, it’s over 150 times faster than the equivalent JSON parser written using Scala-Parser-Combinators. Which shows what can easily happen for libraries that do not care about performance at all
3. Bottom right, we can see the performance of the Scalatags templating engine: a few times faster than scala-xml/twirl and 15-20 times faster than Scalate templates

Some of these benchmarks are recent, while others are a bit out of date. But the point here is that while com.lihaoyi libraries are not top dog in terms of performance, they aim to perform pretty well, and generally do. Well enough that “the library is slow” is generally not going to be an issue in the systems you use them in

3.4.3 Fast is Easy, Fast is Boring

Having fast tools and libraries is good, even if you do not care about performance.

In fact, having fast building blocks is often what *lets* you not care about performance!

3.4.3 Fast is Easy, Fast is Boring

1. Fast building blocks means you can spend *less* time caring about perf

If your building blocks are slow, performance becomes a problem, and you have to spend time on fancy algorithms, optimizations, caching, incremental computation, parallelism, distribution. Maybe you need to swap out your std lib priority queue with a hand-rolled Van Em Boas data structure from CLRS. This adds a whole bunch of cool, challenging, interesting work that is exactly what you do not want to care about when you are trying to add a feature or serve a customer.

Having fast building blocks lets you skip all of this entirely

3.4.3 Fast is Easy, Fast is Boring

1. Fast building blocks means you can spend *less* time caring about perf
2. It means simpler code can still satisfying performance requirements

If your building blocks are fast, often doing the dumb thing is fast enough. Rather than building a complicated hierarchical caching system for your HTML template partials, Scalatags is fast enough you can usually just re-render the whole thing every time, and chrome will crash before your webserver gets perf issues.

3.4.3 Fast is Easy, Fast is Boring

1. Fast building blocks means you can spend *less* time caring about perf
2. It means simpler code can still satisfying performance requirements
3. Fast gives robustness: to poor algorithms, architecture, or configuration

Fast building blocks gives robustness to mistakes. For example, everyone knows you're not meant to be serving large static files from your high-QPS web and API servers. If you start serving huge binary blobs from your Python webservers, things are going to fall down hard. If you're serving huge binary blobs from your Cask webservers, from experience things can keep chugging along remarkably well, and rather than an "outage" you instead have a roadmap item to "optimize static file serving by end of quarter" or even "end of year"

3.4.3 Fast is Easy, Fast is Boring

1. Fast building blocks means you can spend *less* time caring about perf
2. It means simpler code can still satisfying performance requirements
3. Fast gives robustness: to poor algorithms, architecture, or configuration
4. It means you can go from prototype to production without a massive rewrite

Fast building blocks means fewer rewrites. Rather than building your language prototype in scala-parser-combinators and then re-writing it as hand-rolled recursive descent after, you can just use FastParse once and be done with it.

Fast is easy, fast is boring. In the case of com.lihaoyi tools and libraries, being fast is not something that is done at the *expense* of ease of use, but is done to *enhance* it. The less developers have to deal with performance issues in their underlying building blocks, the more they can spend their time and energy on the actually problem that they are trying to solve.

3.4 Principles of the com.lihaoyi ecosystem

Executable Scala Pseudocode that's Easy, Boring and **Fast!**

Executable Scala Pseudocode that's Easy, Boring and Fast!

1. What is the com.lihaoyi ecosystem?
2. What is the com.lihaoyi ecosystem not?
3. **Principles of the com.lihaoyi ecosystem**
4. Case Study: How Mill Makes Builds Great

Executable Scala Pseudocode that's Easy, Boring and Fast!

1. What is the com.lihaoyi ecosystem?
2. What is the com.lihaoyi ecosystem not?
3. Principles of the com.lihaoyi ecosystem
4. **Case Study: How Mill Makes Builds Great**

For the last section of this talk, we will explore the Mill build tool.

Mill is a build tool that uses these principles to make the experience of configuring your build a pleasant one. Users like complaining about build tools: about Maven, about Gradle, about SBT. But users generally have nice things to say about Mill! We will dive into how Mill works, and how it use Scala as Executable Pseudocode to let you define your build in a way that is easy, boring, and fast.

Mill is probably one of the most mature alternatives to SBT today. All the projects in the com.lihaoyi ecosystem have been developed, built, and published using Mill for over half a decade. Coursier, which you all use to resolve and download third-party dependencies, is built using Mill. Scala-CLI is built using Mill. I personally haven't touched SBT for more than 5 years now. Mill works **great**, and now we'll dive into **why**

4.1 Mill as Executable Pseudocode

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}
```

Mill lets you define your project's sub-modules as `object`s: in this case `object foo` will represent the module in the `foo/` folder, normally with sources in `foo/src/`, resources in `foo/resources/` and output in subfolders of `out/foo/`.

In this case, we define a new build target called `lineCount`, that we compute by taking all the source files and counting how many lines are within them. We then override the `resources` of the module to include a generated text file containing the line count, which is written to a `T.dest` folder which (automatically assigned as `out/foo/resources`)

4.1 Mill as Exec. Pseudocode

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}
```

```
// foo/src/Foo.scala
object Foo{
  def main(args: Array[String]): Unit = {
    val lineCount = scala.io.Source
      .fromResource("line-count.txt")
      .mkString

    println(s"Line Count: $lineCount")
  }
}
```

This generated resource file can then be read at runtime, by the source code in the `foo/src/` folder.

4.1 Mill as Exec. Pseudocode

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}
```

```
// foo/src/Foo.scala
object Foo{
  def main(args: Array[String]): Unit = {
    val lineCount = scala.io.Source
      .fromResource("line-count.txt")
      .mkString

    println(s"Line Count: $lineCount")
  }
}
```

```
$ ./mill foo.run
Line Count: 10
```

And you can use `foo.run` to print it out

4.1 Mill as Exec. Pseudocode

```
import mill._, scalalib._
```

```
object foo extends ScalaModule {  
  def scalaVersion = "2.13.8"  
  
  /** Total number of lines in module's source files */  
  def lineCount = T{  
    allSourceFiles().map(f => os.read.lines(f.path).size).sum  
  }  
  
  /** Generate resources using lineCount of sources */  
  override def resources = T{  
    os.write(T.dest / "line-count.txt", "" + lineCount())  
    super.resources() ++ Seq(PathRef(T.dest))  
  }  
}
```

```
// foo/src/Foo.scala  
object Foo{  
  def main(args: Array[String]): Unit = {  
    val lineCount = scala.io.Source  
      .fromResource("line-count.txt")  
      .mkString  
  
    println(s"Line Count: $lineCount")  
  }  
}
```

```
$ ./mill foo.run  
Line Count: 10
```

```
$ ./mill show foo.lineCount  
10
```

`foo.show` to print out the value of any specified intermediate target

4.1 Mill as Exec. Pseudocode

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}

// foo/src/Foo.scala
object Foo{
  def main(args: Array[String]): Unit = {
    val lineCount = scala.io.Source
      .fromResource("line-count.txt")
      .mkString

    println(s"Line Count: $lineCount")
  }
}

$ ./mill foo.run
Line Count: 10

$ ./mill show foo.lineCount
10

$ ./mill inspect foo.lineCount
foo.lineCount(build.sc:6)
  Total number of lines in module's source files

Inputs:
  foo.allSourceFiles
```

Or `inspect` to see the metadata information of a particular build target

This is a synthetic example, but it is enough to give you an idea of how Mill works. In Mill, you write more-or-less plain Scala code: you have objects that extend traits, method defs that call or override other method defs, super, and so on. That's all Mill needs to schedule them to be built in the right order, automatically caching things, invalidating, parallelizing, and letting you inspect the build to debug it. For example, if I don't edit any source files, my **lineCount** computation here will not be re-evaluated unnecessarily.

These things don't matter so much for a toy example, but become much more important when the size and complexity of the build grows. For example, in large SBT builds it's very common for people being puzzled why it's slow, and why certain tasks are being re-executed even when nothing material changed, or why one task causes another to be run. That confusion generally does not happen in Mill.

4.1 Mill as Exec. Pseudocode

```
import mill._, scalalib._
```

```
object foo extends ScalaModule {  
  def scalaVersion = "2.13.8"  
  
  /** Total number of lines in module's source files */  
  def lineCount = T{  
    allSourceFiles().map(f => os.read.lines(f.path).size).sum  
  }  
  
  /** Generate resources using lineCount of sources */  
  override def resources = T{  
    os.write(T.dest / "line-count.txt", "" + lineCount())  
    super.resources() ++ Seq(PathRef(T.dest))  
  }  
}
```

```
import sbt._, Keys._
```

```
lazy val lineCount = taskKey[Int](  
  "Total number of lines in module's source files"  
)  
  
lazy val foo = project.in(file("."))  
  .settings(  
    name := "foo",  
    scalaVersion := "2.13.8",  
    lineCount := {  
      val srcFiles = (Compile / sources).value  
      srcFiles.map(f => IO.readLines(f).size).sum  
    },  
    Compile / resource += {  
      val dest = (Compile / resourceManaged).value  
      val count = lineCount.value  
      val lineCountFile = dest / "line-count.txt"  
      IO.write(lineCountFile, count.toString)  
      lineCountFile  
    }  
  )
```

One thing that's worth mentioning is how this compares to the equivalent SBT build.

The SBT code contains most of the same concepts. However, the same concept - e.g. defining namespaces, defining values, etc. - look very different in the SBT build on the right when compared to the Mill example on the left, and very different from what you would expect to see in "normal" Scala code.

1. Rather than `object`s, we have `(project in file).settings`s.
2. Rather than `def lineCount =`, we need to do `lazy val lineCount = taskKey` and then `lineCount :=` later.
3. Rather than normal scaladoc comments, we need to put the lineCount documentation as a string

We can see from this that it's not the conciseness or verbosity that makes or breaks executable pseudocode: the SBT example is marginally more verbose than the Mill example, but not terribly so. Rather, it is the fact that SBT code - while nominally Scala - does not look like any Scala code you would write anywhere else. Mill code, does.

As a Scala developer, you already know how objects, traits, defs, overrides, supers, and method calls work. Mill takes all that stuff you already know, and then automatically augments it with all the stuff you probably want in your build pipelines: caching, parallelism, queryability, etc. That's why Mill code looks so familiar, especially compared to other build tools like SBT or Gradle; it's executable pseudocode!

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}
```

Many people have commented that Mill is not *Simple*. And it's true: Mill aims to be *Easy*, not *Simple*. In order to give the seamless “write what you think, we'll take care of the rest” experience shown earlier, Mill has to jump through a *lot* of hoops! Let's go through some of them

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  def lineCount = new Target(
    T.zipMap(Seq(allSourceFiles)) { case Seq(allSourceFiles) =>
      allSourceFiles.map(f => os.read.lines(f.path).size).sum
    }
  )

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}
```

```
$ ./mill inspect foo.lineCount
foo.lineCount(build.sc:6)
  Total number of lines in module's source files

Inputs:
  foo.allSourceFiles
```

First, `T{}` blocks with `()` inside are macros, that expand into a `new Target` with a `zipMap` call, effectively turning the direct-style code into a free applicative. This free applicative graph structure gives us the scheduling, parallelizability, introspectability, and other things that people want from their build tool.

For example, you can easily use `mill inspect` to list out dependencies,

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  def lineCount = new Target(
    T.zipMap(Seq(allSourceFiles)) { case Seq(allSourceFiles) =>
      allSourceFiles.map(f => os.read.lines(f.path).size).sum
    }
  )

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}
```

```
$ ./mill inspect foo.lineCount
foo.lineCount(build.sc:6)
  Total number of lines in module's source files

Inputs:
  foo.allSourceFiles

$ ./mill path foo.run foo.lineCount
foo.lineCount
foo.resources
foo.localClasspath
foo.runClasspath
foo.run
```

`mill path` to find how one target depends on another, and so on.

All these are things that you cannot do with the callgraph of “normal” code.

`T{}` is a macro. People say these macro transformations are not simple, and it's true: they're not. But it does make things *easier*.

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  @Scaladoc("Total number of lines in module's source files")
  def lineCount = new Target(
    T.zipMap(Seq(allSourceFiles)) { case Seq(allSourceFiles) =>
      allSourceFiles.map(f => os.read.lines(f.path).size).sum
    },
  )

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}
```

```
$ ./mill inspect foo.lineCount
foo.lineCount(build.sc:6)
  Total number of lines in module's source files

Inputs:
  foo.allSourceFiles
```

We also use a compiler plugin to save the Scaladoc as an annotation, that we then fish out later using Java reflection. This is what lets us print out the Scaladoc when you inspect things from the command line.

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  @Scaladoc("Total number of lines in module's source files")
  def lineCount = new Target(
    T.zipMap(Seq(allSourceFiles)) { case Seq(allSourceFiles) =>
      allSourceFiles.map(f => os.read.lines(f.path).size).sum
    },
    line = sourcecode.Line(6),
    fileName = sourcecode.FileName("build.sc")
  )

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}
```

```
$ ./mill inspect foo.lineCount
foo.lineCount(build.sc:6)
  Total number of lines in module's source files

Inputs:
  foo.allSourceFiles
```

We use implicit macros from the `com.lihaoyi:sourcecode` library to inject the line number and file name, so when you `inspect` the target at the command line you can see where it is defined.

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule(path = outer.path / "foo") {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  @Scaladoc("Total number of lines in module's source files")
  def lineCount = new Target(
    T.zipMap(Seq(allSourceFiles)) { case Seq(allSourceFiles) =>
      allSourceFiles.map(f => os.read.lines(f.path).size).sum
    },
    line = sourcecode.Line(6),
    fileName = sourcecode.FileName("build.sc"),
    path = foo.path / "lineCount"
  )

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
  }
}
```

```
$ ./mill foo.run
Line Count: 10

$ ./mill show foo.lineCount
10
```

Next, we use implicits macros to grab the name of the `object`s and `def`. That is how Mill knows that the `foo.run` or `foo.lineCount` commands correspond to those modules and targets,

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule(path = outer.path / "foo") {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  @Scaladoc("Total number of lines in module's source files")
  def lineCount = new Target(
    T.zipMap(Seq(allSourceFiles)) { case Seq(allSourceFiles) =>
      allSourceFiles.map(f => os.read.lines(f.path).size).sum
    },
    line = sourcecode.Line(6),
    fileName = sourcecode.FileName("build.sc"),
    path = foo.path / "lineCount"
  )

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
  }
}

$ ./mill foo.run
Line Count: 10

$ ./mill show foo.lineCount
10

foo/src/Foo.scala
out/foo/lineCount.json
{
  "value": 10,
  "valueHash": 1291200293,
  "inputsHash": 1717538688
}
```

And that they should read sources from the `foo/` folder and output their caches and other metadata in the `out/foo/` folder e.g. `out/foo/lineCount.json`

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule(path = outer.path / "foo") {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  @Scaladoc("Total number of lines in module's source files")
  def lineCount = new Target(
    T.zipMap(Seq(allSourceFiles)) { case Seq(allSourceFiles) =>
      allSourceFiles.map(f => os.read.lines(f.path).size).sum
    },
    line = sourcecode.Line(6),
    fileName = sourcecode.FileName("build.sc"),
    path = foo.path / "lineCount",
    readWriter = upickle.default.readwriter[Int]
  )

  /** Generate resources using lineCount of sources */
  override def resources = T{
```

Lastly, we use implicits to resolve a `upickle.default.ReadWriter`, that is used to cache things on disk. This means if we don't keep a persistent Mill process open, a future Mill process is still able to pick things up from the cache where the past Mill process left off.

4.2 Mill is Easy, not Simple

```
import mill._, scalalib._

object foo extends ScalaModule {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  /** Generate resources using lineCount of sources */
  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    super.resources() ++ Seq(PathRef(T.dest))
  }
}

import mill._, scalalib._

object foo extends ScalaModule(path = outer.path / "foo") {
  def scalaVersion = "2.13.8"

  /** Total number of lines in module's source files */
  @Scaladoc("Total number of lines in module's source files")
  def lineCount = new Target(
    T.zipMap(Seq(allSourceFiles)) { case Seq(allSourceFiles) =>
      allSourceFiles.map(f => os.read.lines(f.path).size).sum
    },
    line = sourcecode.Line(6),
    fileName = sourcecode.FileName("build.sc"),
    path = foo.path / "lineCount",
    readWriter = upickle.default.readwriter[Int]
  )

  /** Generate resources using lineCount of sources */
  override def resources = T{
```

Overall, if you look at the thing on the right, the transformations that let Mill do what it does are certainly not **Simple**: java reflection, compiler plugins, implicits, macros, implicit macros.

However, all this complexity is towards one purpose: to make life **Easy** for the person configuring the build! To them, they just need to write their Scala code as it were pseudocode from an intro-to-Scala class - with objects, traits, and defs. Methods that call other methods. Override and super. All the other “stuff” around build tooling - incremental computation, caching, parallelism, introspectability, and so on is done **for** them, rather than **by** them. Mill uses its complexity budget to ensure there’s **fewer** things that a user would need to care about, rather than **more**.

4.3 Mill is Fast

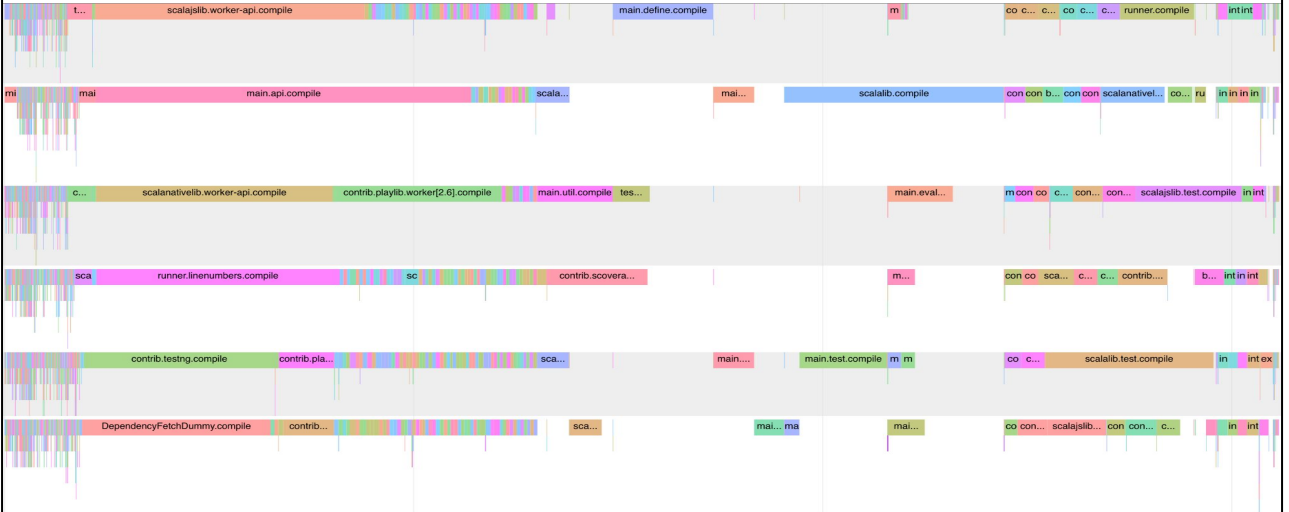
```
./mill -j 6 __.compile
```

Lastly, Mill makes it very easy to make your builds fast via parallelism. You can pass in how many parallel jobs you want, e.g. `-j 6` gives you 6 parallel jobs

4.3 Mill is Fast

```
./mill -j 6 __.compile
```

```
# out/mill-chrome-profile.json
```



Mill will then run everything using 6 threads, scheduling things appropriately in dependency order, while generating a nice profile you can load into chrome to see where your time is being spent.

This makes it easy to see where the bottlenecks are: e.g. the blue segment in the top-middle which is running alone after all the segments on the left run in parallel, and needs to complete before all the segments on the right run in parallel.

While Mill goes to great efforts to make its own internal implementation reasonably fast, in any build tool the bottleneck often is the user-defined build steps or build actions. Mill makes it easy to triage the performance of your build, so you can figure out what parts of your own Mill build are slow, and do something about it.

Executable Scala Pseudocode that's Easy, Boring and Fast!

1. What is the com.lihaoyi ecosystem?
2. What is the com.lihaoyi ecosystem not?
3. Principles of the com.lihaoyi ecosystem
4. **Case Study: How Mill Makes Builds Great**

com.lihaoyi, Executable Scala Pseudocode that's Easy, Boring and Fast!

Now for some parting references.

com.lihaoyi, Executable Scala Pseudocode that's Easy, Boring and Fast!

com-lihaoyi Github Organization: <https://github.com/com-lihaoyi>

- #com-lihaoyi on the Scala Discord

If any of you want to look at these libraries and try them out, they are all available under the github.com/com-lihaoyi github organization. These are open source projects, and we always need more contributors helping to use, improve and maintain these libraries. Even if you are already using Cats or Akka or ZIO or something else, these small libraries can fit right in

com.lihaoyi, Executable Scala Pseudocode that's Easy, Boring and Fast!

com-lihaoyi Github Organization: <https://github.com/com-lihaoyi>

- #com-lihaoyi on the Scala Discord

Hands-on Scala Programming <https://www.handsonscala.com/>

- Code Examples <https://github.com/handsonscala/handsonscala>

If anyone wants a more structured way of learning about these tools and libraries, you can check out my book Hands-On Scala Programming, at www.handsonscala.com. This book is a systematic tour of how to use the Scala - supported by the com.lihaoyi ecosystem - to build a lot of projects with real-world applicability. While other Scala books might have you re-implementing linked lists and fibonacci functions, Hands-on Scala has you implementing parallel web-crawlers, realtime web-sites, networked Scala applications, programming language interpreters, and many other things that your employer may pay real money for you to do.

The book's example projects are available for free online. Even without the book, this is a great resource for the question of "how do I do X" using the com.lihaoyi ecosystem libraries. All 143 examples are free online, complete with test suites

com.lihaoyi, Executable Scala Pseudocode that's Easy, Boring and Fast!

com-lihaoyi Github Organization: <https://github.com/com-lihaoyi>

- #com-lihaoyi on the Scala Discord

Hands-on Scala Programming <https://www.handsonscala.com/>

- Code Examples <https://github.com/handsonscala/handsonscala>

Mill Build Tool <https://github.com/com-lihaoyi/mill>

Lastly, if anyone here does **not** like SBT, you should take some time to try out the Mill build tool. We just released version 0.11.0, with a ton of improvements by both myself and other contributors. People like saying that SBT is a problem, confusing newbies and veterans alike, and holding Scala back from widespread adoption. But there is an alternative to SBT - It's Mill - and many of us have been using Scala without SBT for years now. You all should try it out and see how great the post-SBT world is!