CSCI 334:
Principles of Programming Languages

Lecture 1: Intro to F#

Instructor: Dan Barowy

Williams

Topics

What to **expect** in this course

What this course is **about**

Things to know for this week's **assignment**

Syllabus and Honor Code

Every week:

1. Readings to do **before doing the lab**.
   Read actively.
2. Labs **due every Wednesday at midnight**.
   Know how to get your lab.
3. In-class quiz **every Thursday**.
   If you did the reading and the lab yourself, this should not worry you much.

Your to-dos

1. Read the **syllabus**.
2. Lab 1, **due Wednesday 2/12**.
3. Be sure to do the assigned readings.
4. If you plan to use lab computers, check that you can login **today**.
   Only Lida/Kelsey can help you with account problems. They are well-adjusted people who work 9am-5pm during the work week.

If you don't remember your login…

Email csaccounts@williams.edu

If you don't know/remember the TCL 312/TBL 301 door code…

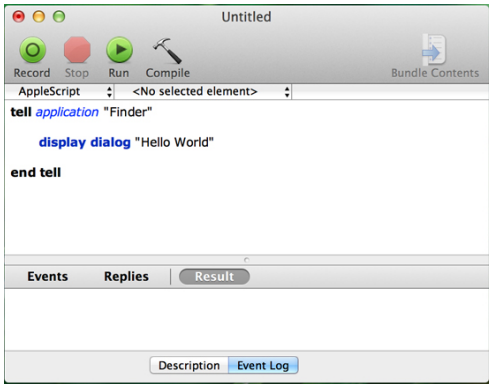
Why do I study programming languages?

A Bicycle for the Mind

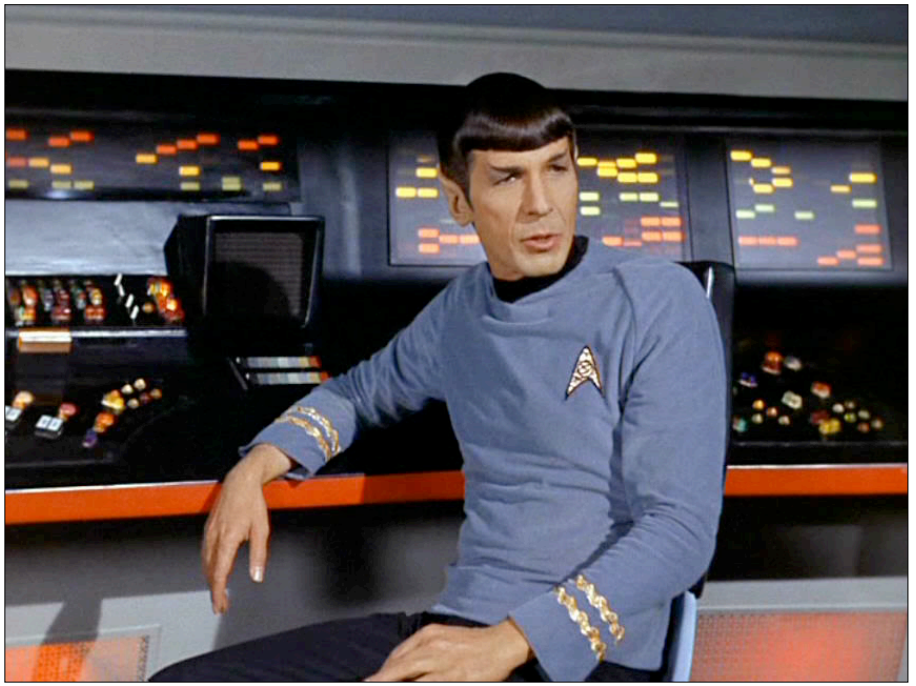
A Bicycle for the Mind

A computer is a key part of this “bicycle”—but not all of it.

A Bicycle for the Mind



The other key part is a programming language.



FlashFill

Conferences Sign In

USENIX ATC '22 ATTEND PROGRAM PARTICIPATE SPONSORS ABOUT

Riker: Always-Correct and Fast Incremental Builds from Simple Specifications

Authors:
Charlie Curtsinger, *Grinnell College*; Daniel W. Barowy, *Williams College*

Awarded Best Paper!

Abstract:
Build systems are responsible for building software correctly and quickly. Unfortunately, traditional build tools like make are correct and fast only when developers precisely enumerate dependencies for every incremental build step. Forward build systems improve correctness over traditional build tools by discovering dependencies automatically, but existing forward build tools have two fundamental flaws. First, they are incorrect; existing forward build tools miss dependencies because their models of system state are incomplete. Second, they rely on users to manually specify incremental build steps, increasing the programmer burden for fast builds.

This paper introduces Riker, a forward build system that guarantees fast, correct builds. Riker builds are easy to specify; in many cases a single command such as gcc *.c suffices. From these simple specifications, Riker automatically discovers fast incremental rebuild opportunities. Riker models the entire POSIX filesystem—not just files, but directories, pipes, and so on. This model guarantees that every dependency is checked on every build so every output is correct.

We use Riker to build 14 open source packages including LLVM and memcached. Riker incurs a median overhead of 8.8% on the initial full build. On average, Riker's incremental builds realize 94% of make's incremental speedup with no manual effort and no risk of errors.

Open Access Media

USENIX is committed to Open Access to the research presented at our events. Papers and proceedings are freely available to everyone once the event begins. Any video, audio, and/or slides that are posted after the event are also free and open to everyone. Support USENIX and our commitment to Open Access.

BibTeX

Curtsinger PDF

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

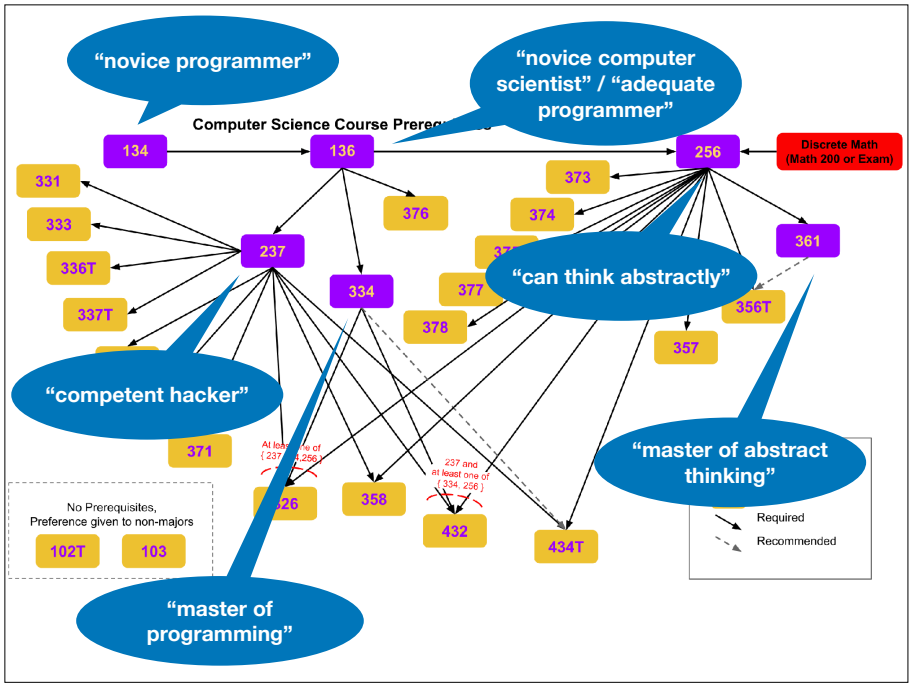
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Award: Best Paper

Class outcomes

1. Speak the **language of languages**
   a. understand the role of a **language model**
   b. evaluate **fitness** of language for **purpose**
   c. **rapidly learn** new languages
2. Add tools to your **mental toolbox**
   a. techniques for **clear thinking**
   b. become a **much better programmer**



Class outcomes

1. Speak the **language of languages**
   a. understand the role of a **language model**
   b. evaluate **fitness** of language for **purpose**
   c. **rapidly learn** new languages
2. Add tools to your **mental toolbox**
   a. techniques for **clear thinking**
   b. become a **much better programmer**
3. **Be your favorite class!**

I always care about what you think

1. Optional feedback on assignments (for bonus credit!)
2. Optional, anonymous feedback form on course website

push-checker

- To trigger:

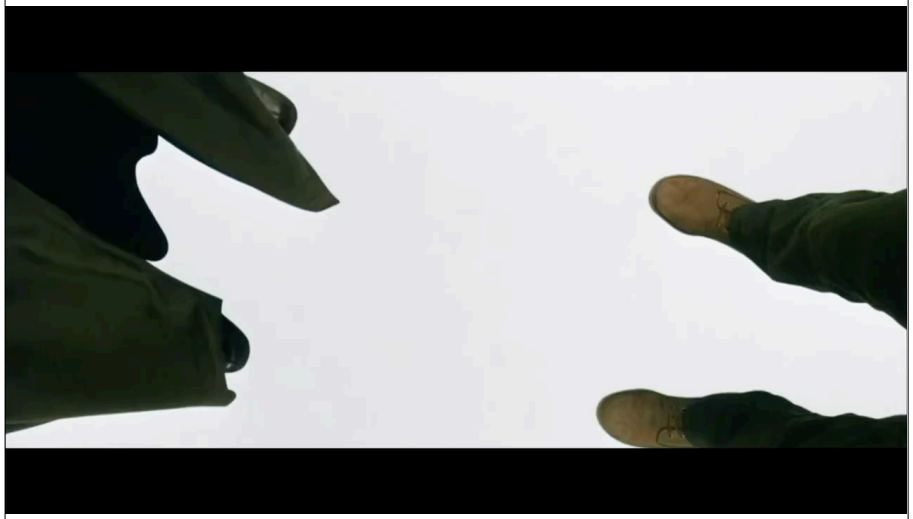
```
$ git commit -m "check"
$ git push
```

- A checklist will appear in your repo's **issues** tab.
- You get **1 attempt per day**.

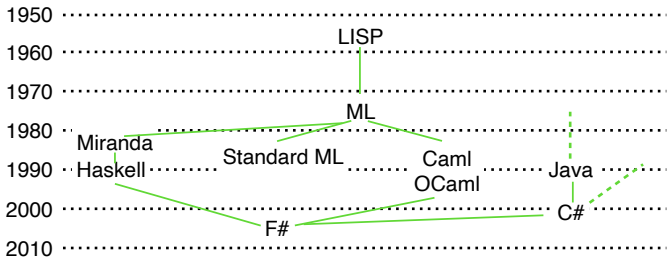
  (lab 1 gets 10 attempts per day)

F# as ML

"Free your mind"



ML



Two F# environments

F# REPL

```
$ dotnet fsi

Microsoft (R) F# Interactive version 12.8.0.0 for F# 8.0
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

>
```

Type `#quit;`; to quit.

Use the REPL as a code playground.

Making a project

```
$ dotnet new console -lang f#
```

Use a project for homework.

Homework workflow

1. **Clone** your repository.
2. `cd` to the question you want to work on.
3. **Create a project** in that directory.
4. **Write**, **build**, and **test** your code.
5. `git commit` when you've reached a checkpoint.
6. `git push` when you want to upload your code.
7. Go to step 2 **unless done**.

Expressions vs **statements**

Arithmetic operators

operation	syntax
add	`+`
subtract	`-`
multiply	`*`
divide	`/`
modulus	`%`
exponent	`**`

Logical operators

operation	syntax
and	`&&`
or	`||`
not	`not`
equals	`=`
not equals	`<>`
inequalities	`<, >, <=, >=`

Immutability vs **mutability**

Pure vs **side-effecting** functions

Strong, static vs **weak, dynamic** types

Other things

`unit` datatype

```
public static void main(String[] args) { … }
```

```
let main args = …
```

`unit` datatype

```
public static void main(String[] args) { … }
```

```
let main(args: string[]) = …
```

Remember: every expression must **return a value**.

A function **can't** return nothing.

`unit` datatype

```
public static void main(String[] args) { … }
```

```
let main(args: string[]) : unit = …
```

Therefore, "nothing" is a thing… called `unit`.

`unit` datatype

```
$ dotnet fsi

Microsoft (R) F# Interactive version 10.2.3 for F# 4.5
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> unit;;

  unit;;
  ^^^^

stdin(1,1): error FS0039: The value or constructor 'unit' is
not defined.

> ();;
val it : unit = ()

>
```

How does one obtain a value of `unit`? `()`

By the way…

```
let main(args: string[]) : unit = …
```

By the way…

```
let main(args: string[]) : int = …
```

Go practice

Recap & Next Class

Today:

- Course goals
- Course structure
- Basic F#

Next class:

- Recursive functions