Comp 311
Functional Programming

Nick Vrvilo, Robert “Corky” Cartwright

Instructors’ Background

- Corky Cartwright, PhD on semantics and verification of functional programs under David Luckham and John McCarthy at Stanford, 40 years of research in PL theory and PL systems (software engineering)
- Nick Vrvilo, fresh PhD (under Vivek Sarkar) and new 2σ employee focused on PL systems that support efficient parallel computation.

Course Overview

- An Introduction to Functional Programming
- Tuesdays and Thursdays 4PM-5:15AM
- Office hours:

  Corky: 2-4 Wed in DH 3110
  Nick: TBA

Course Mechanics

- Course website: https://wiki.rice.edu/confluence/display/FPSCALA/2017-Fall
  - Syllabus, lectures and homework assignments are posted there
  - Lecture topics are subject to change
- Course mailing list: comp311@rice.edu

Online Course Discussion

- Piazza
  https://piazza.com/rice/fall2017/comp311/home
  - We will make a best effort to answer questions posted on this page in a timely manner
  - *There is no SLA (?)*
  - *Bring your questions to class and office hours*

Course Overview

- No required textbook
    - We will draw from a variety of sources
- Coursework consists primarily of weekly homework assignments
    - Make sure you do these!
    - Missing even one assignment will significantly impact your grade

Homework Assignments

- Think of the assignments in this class as short essays
- Focus as much on style as you would for an essay
- 50% of a homework grade is based on clarity and style
- 50% on correctness

Homework Assignments

- There will be two weeks between assignment and due date.
- 7 slip days, no other extensions (not like the real world where 0 slip days often prevails). No more than 3 slip days per assignment.
- Aiming for roughly 10 hours of coursework per week.
- Block this time off now and make a priority of respecting it.

Homework Assignments

- Assignments are published on Thursdays
- Start on assignments early so that you have time to ask questions at class and at office hours

Homework Assignments

- Assignments will be programming exercises in Scala
- We will cover the parts of Scala needed for the assignments in class

Homework Assignments

- You have the option of DrScala and IntelliJ IDEA for assignments. DrScala is less professional but better supported.
  - Installed on all Rice systems and available for download from the course website
- We will use turnin on CLEAR for all assignments
  - Instructions on the course website

What is Functional Programming?

Early Models of Computation

- Turing Machines (Turing)
- Type-0 Grammars (Chomsky)
- The Lambda Calculus (Church)
- … *and many others*

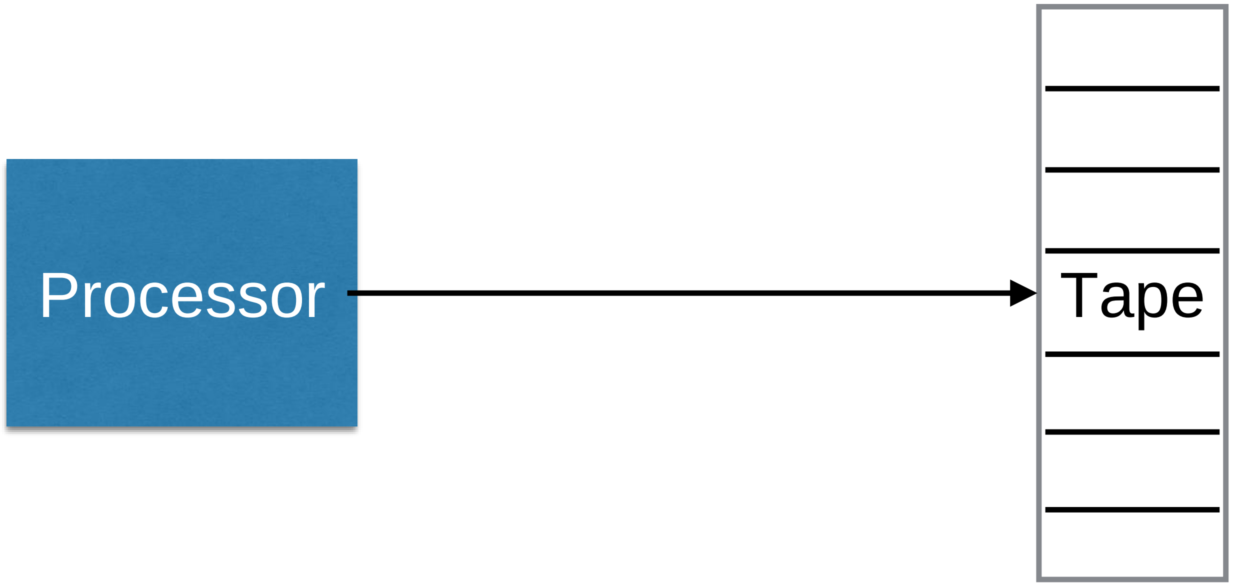
Early Models of Computation

- Turing Machines (Turing)
- Type-0 Grammars (Chomsky)
- The Lambda Calculus (Church)
- … *and many others*
- To the surprise of their inventors, all of these systems turned out to be equivalent in expressive power
  - Suggests there is a deeper structure to the nature of computation

Early Models of Computation

- Turing Machines (Turing)
- Type-0 Grammars (Chomsky)
- **The Lambda Calculus (Church)**
- … *and many others*
- To the surprise of their inventors, all of these systems turned out to be equivalent in expressive power
  - Suggests there is a deeper structure to the nature of computation

Turing Machines



- Processor is a finite state machine that loads and stores *memory cells*
- Turing coined the term “compute” and introduced the notion of storage
- Many programs, languages, and computer architectures are heavily influenced by this model (and its derivates: Von Neumann, etc.)

Early Models of Computation

- Turing Machines (Turing)
- Type-0 Grammars (Chomsky)
- **The Lambda Calculus (Church)**
- … *and many others*
- To the surprise of their inventors, all of these systems turned out to be equivalent in expressive power
    - Suggests there is a deeper structure to the nature of computation

The Lambda Calculus

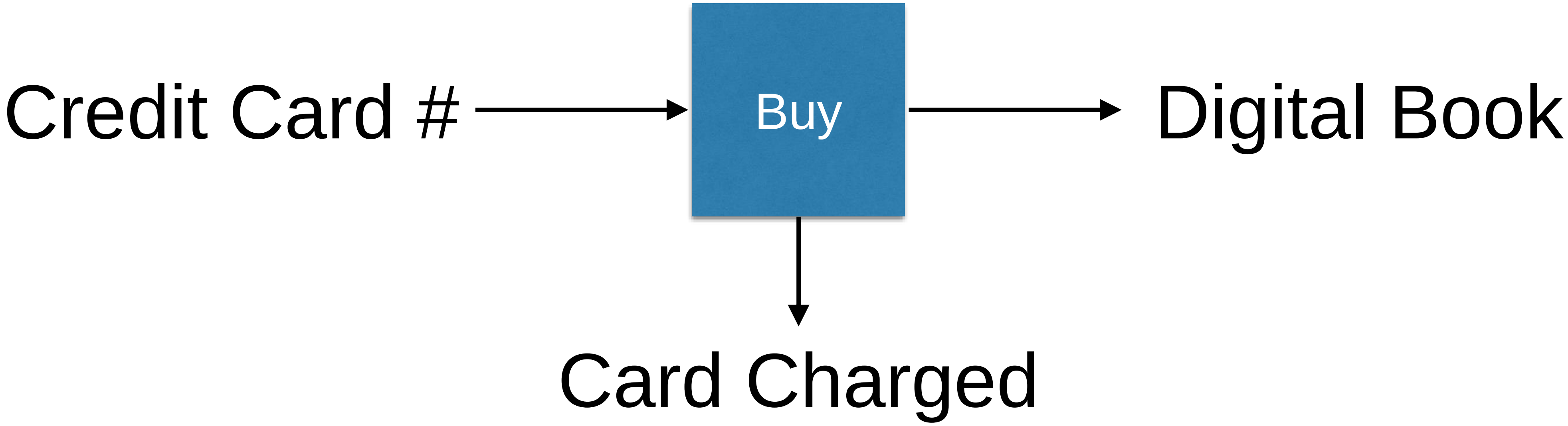
- A *calculus* consists of a set of rules for rewriting symbols
- An attempt to rebuild all of mathematics on the notion of *functions* and *applications*
- There is no mutation in the lambda calculus
- Every program consists solely of applications of functions to arguments (which are also functions)
- Applications of functions return values (which are also functions)

What is Functional Programming?

A style of programming inspired by the Lambda Calculus as a foundational model of computation.

What is Functional Programming?

- A style of programming that avoids side effects



Credit Card #

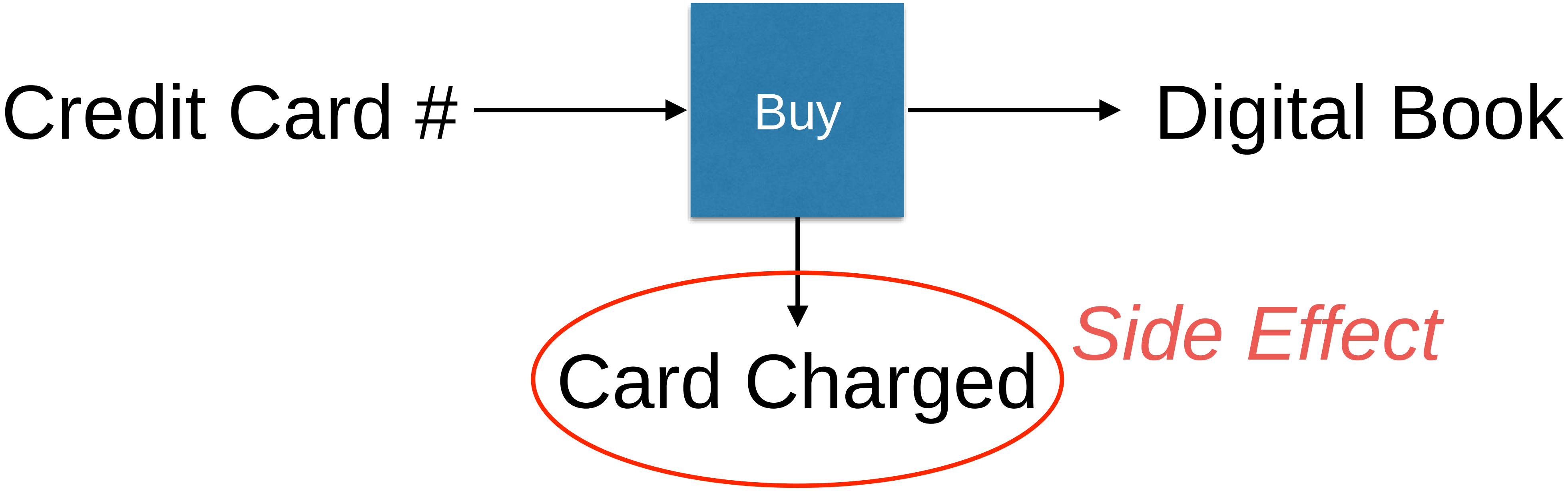
Buy

Digital Book

Card Charged

What is Functional Programming?

- A style of programming that avoids side effects



Credit Card #

Buy

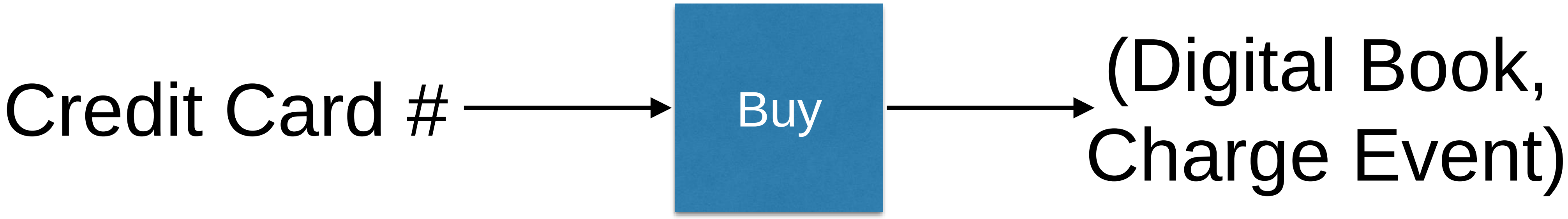
Digital Book

Card Charged

Side Effect

What is Functional Programming?

- A style of programming that avoids side effects



Credit Card # → Buy → (Digital Book, Charge Event)

- All results of a computation are sent as output

Why Avoid Side Effects?

- **Programs are easier to write:** There are fewer interactions between program components, enabling multiple programmers (or a single programmer on multiple days) to work together more easily

- **Programs are easier to read:** Pieces of a program can be read and understood in isolation

- **Programs are easier to test:** Less context needs to be built up before calling a function to test it

- **Programs are easier to debug:** Problems can be isolated more easily, and behavior is inherently deterministic

- **Programs are easier to reason about:** The model of computation needed to understand a program without mutation is much simpler

Why Avoid Side Effects?

- **Programs are easier to execute in parallel:** Because separate pieces of a computation do not interact, it is easy to compute them on separate processors

- This is an increasingly important consideration in the era of multicore chips, big data, and distributing computing

    - *This advantage undermines an often cited argument for mutation (efficiency)*

What is Functional Programming?

- A style of programming that emphasizes functions as the basis of computation
  - Functions are applied to arguments
  - Functions are passed as arguments to other functions
  - Functions are returned as values of applications

Why Emphasize Functions?

- Functions allow us to factor out common code
  - DRY: Don’t Repeat Yourself
    - Why is this important?
  - Passing functions as arguments is often the most straightforward way to abide by DRY
  - Returning functions as values is also important for DRY

Why Emphasize Functions?

- Functions allow us to concisely package computations and move them from one control point to another

  - Aids us with implementing and reasoning about parallel and distributed programming (yet again)

A Word on Object-Oriented Programming

- There is no tension between functional and object-oriented programming. In fact, OOP can be cast as an enrichment of FP. See https://www.cs.rice.edu/~javaplt/papers/OOPEnrichesFP.pdf
- In many ways, they complement one another
- Scala was designed to integrate both styles of programming

A New Paradigm

- Set aside what you’ve learned about programming
- The style we will practice might seem unfamiliar at first
- Initially, the material will seem quite basic
    - We will build a solid foundation that will enable us to explore advanced topics

A New Paradigm

- We will re-examine many things we’ve (partially) learned
  - Often in life, the way forward is to rethink our assumptions
  - Later, we can integrate what we’ve learned into our larger body of knowledge

Our First Exposure to Computation:

Arithmetic

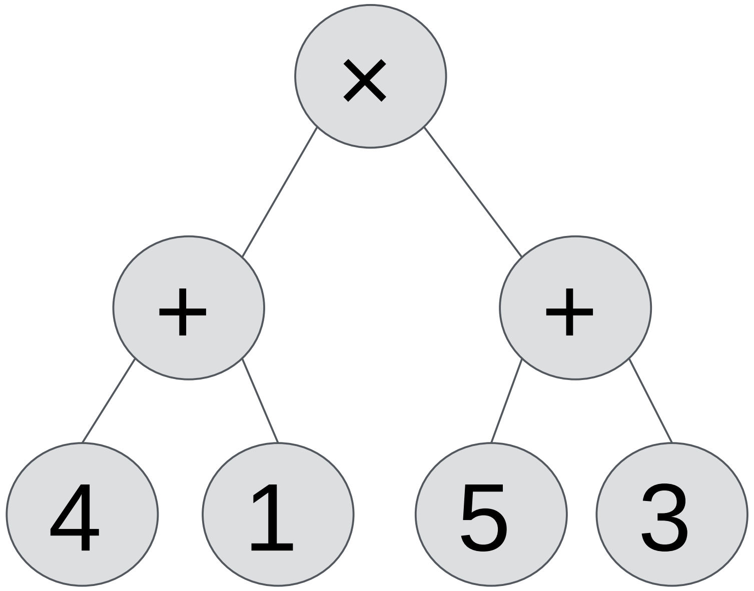
$4 + 5 = 9$

$$4 + 5 \mapsto 9$$

expressions are **reduced** to **values**

Critical Intuition*

- Reduction rules (although typically written using conventional [*concrete*] syntax) work on *abstract syntax trees* (ASTs).
- Every expression in conventional (concrete) syntax corresponds to an abstract syntax tree.
- Example: (4 + 1) × (5 + 3)



Critical Intuition II*

- Tree structure is typically encoded in concrete syntax using parentheses
- Example:
  - normal function application notation, *e.g.*, *prod(sum(3,1), sum(5,3))*
- Expressions with parentheses are hard for humans to read so common mathematical notation heavily relies on infix notation for binary operators and precedence conventions, *e.g.*, 2 + 3 × 6 *vs*. 2 × 3 + 6
- Thinking about syntax in terms of ASTs simplifies reduction rules

Expressions are Reduced to Values

- Rules for a fixed set of operators:
  - 4 + 5 ↦ 9
  - 4 - 5 ↦ -1
  - 4 × 5 ↦ 20
  - 9 / 3 ↦ 3
  - 4^2 ↦ 16
  - √4 ↦ 2

Expressions are Reduced to Values

To reduce an operator applied to expressions, first reduce the subexpressions, left to right:

$$(4 + 1) \times (5 + 3) \mapsto$$

$$5 \times (5 + 3) \mapsto$$

$$5 \times 8 \mapsto$$

$$40$$

Expressions are Reduced to Values

A precedence is defined on operators to help us decide what to reduce next:

$$4 + 1 \times 5 + 3 \mapsto$$

$$4 + 5 + 3 \mapsto$$

$$9 + 3 \mapsto$$

$$12$$

New Operations Often Introduce New Types of Values

- 4 + 5 ↦ 9
- 4 - 5 ↦ -1
- 4 × 5 ↦ 20
- 4 / 5 ↦ 0.8
- 4^2 ↦ 16
- √-1 ↦ i

Old Operations on New Types of Values Often Introduce Yet More New Types of Values

$1 + i$