Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

My Background

- Rice PhD, Computer Science
- Experience in distributed computing, language design and implementation, web services, natural language processing, machine learning
- Vice President, Engineering at Two Sigma Investments
    - Quantitative Software Engineering
    - Machine Learning
    - Distributed Computing

Course Overview

- An Introduction to Functional Programming
- Tuesdays and Thursdays 2:30 PM - 3:45 PM
- Office hours: Tuesdays 4 PM - 5 PM DH 2161

Course Mechanics

- Course website: https://wiki.rice.edu/confluence/display/PARPROG/COMP311
  - 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/class/ibslot8j6un5p6
    - 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 entirely 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 one week between assignment and due date (sometimes more)
- No slip days, no extensions (just like the real world)
- 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
- My office hours are on Tuesdays
- Start on assignments before the following Tuesday 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

- We will use DrScala for all assignments
    - Installed on all Rice systems and available for download from the course website
- We will use turnin 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*

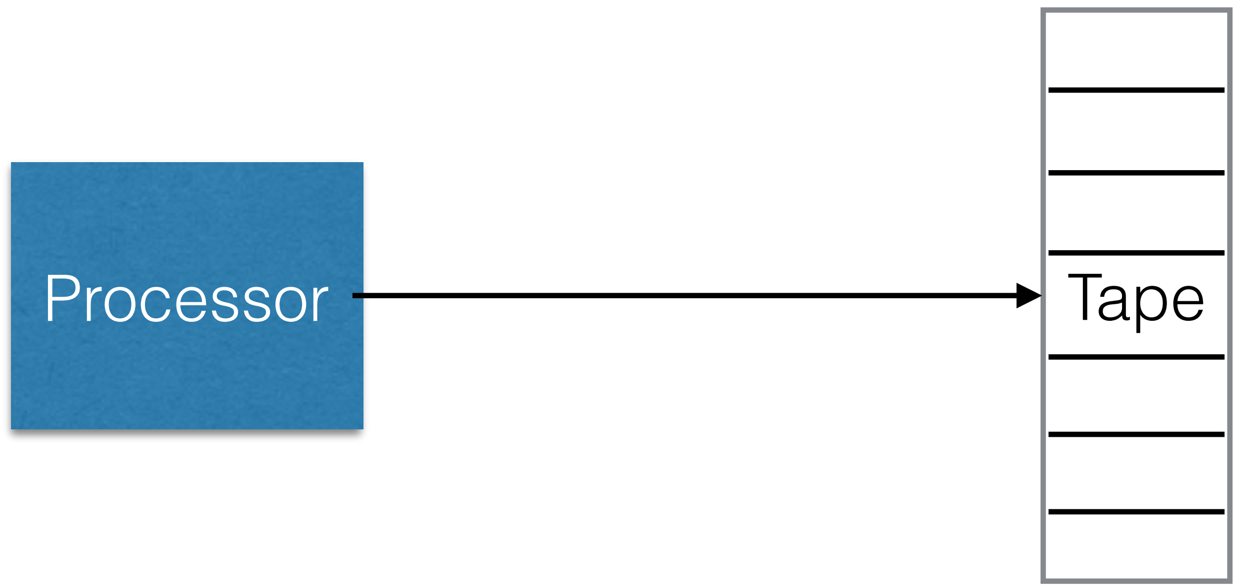
Early Models of Computation

- 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

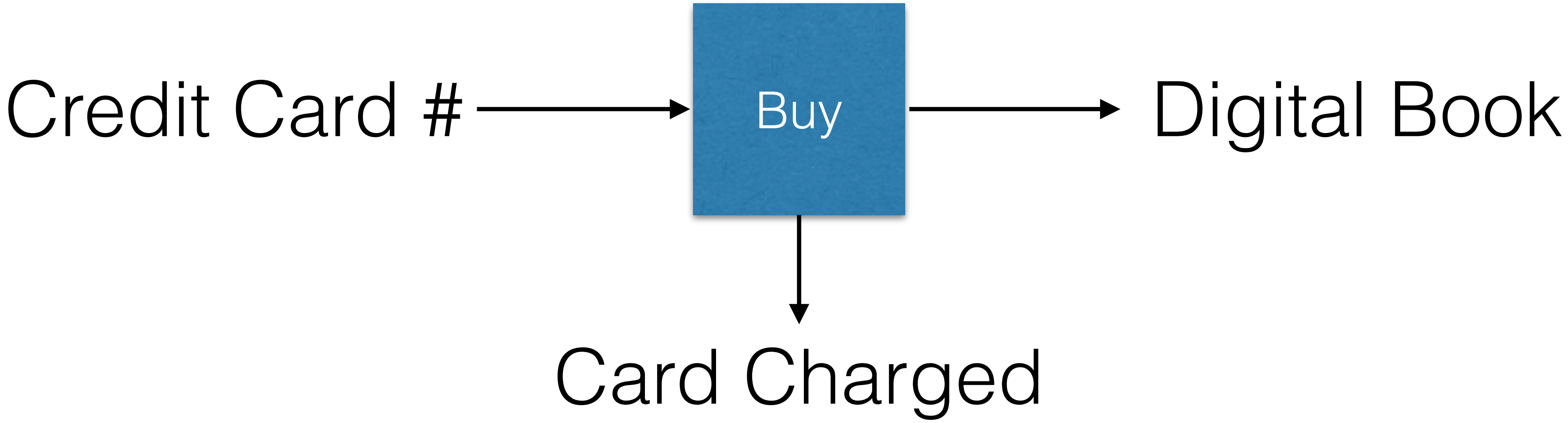
- A *calculus* consists of a set of rules for rewriting symbols
- An attempt to rebuild all of mathematics on the notion of *functions* and *applications*
- There is no mutation in the lambda calculus
- Every program consists solely of applications of functions to arguments (which are also functions)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

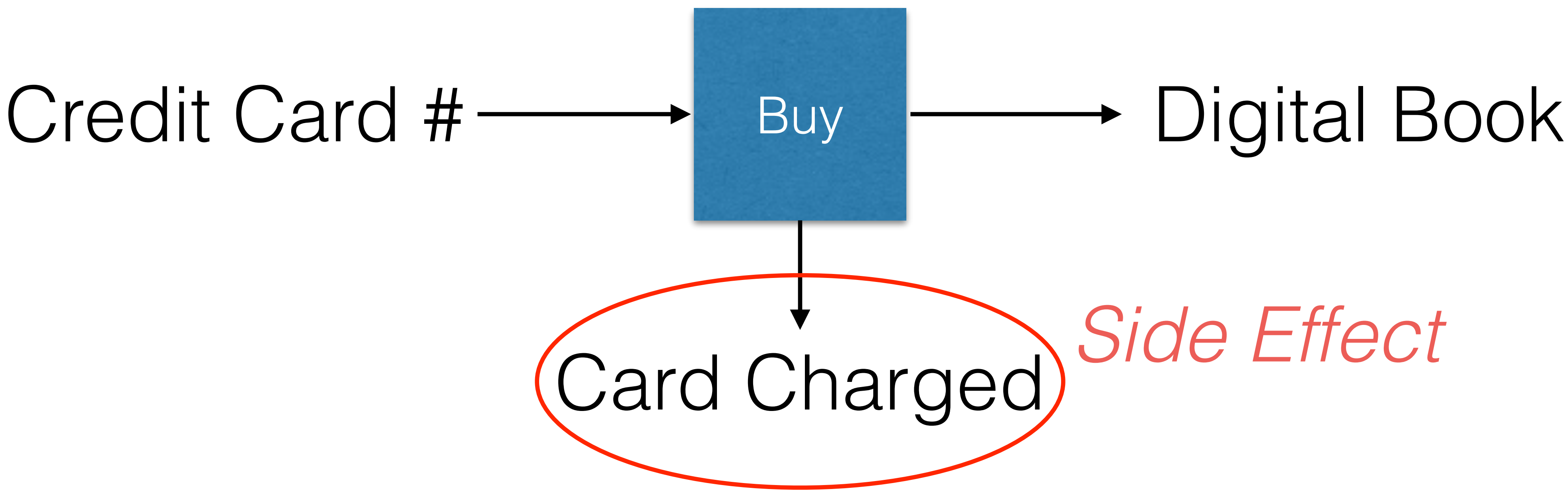
What is Functional Programming?

- A style of programming that avoids side effects



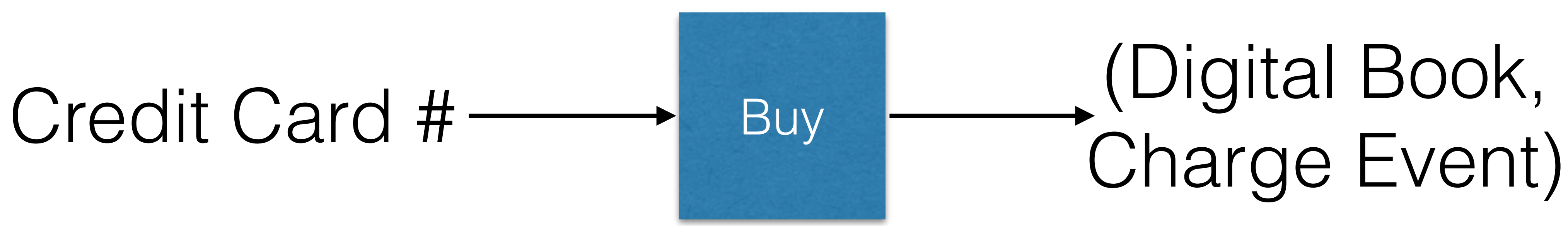
What is Functional Programming?

- A style of programming that avoids side effects



What is Functional Programming?

- A style of programming that avoids side effects



- 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 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**

Expressions are Reduced to Values

- Rules for a fixed set of operators:
  - $4 + 5 \mapsto 9$
  - $4 - 5 \mapsto -1$
  - $4 \times 5 \mapsto 20$
  - $9 / 3 \mapsto 3$
  - $4^2 \mapsto 16$
  - $\sqrt{4} \mapsto 2$

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

- $4 + 5 \mapsto 9$
- $4 - 5 \mapsto -1$
- $4 \times 5 \mapsto 20$
- $4 / 5 \mapsto 0.8$
- $4^2 \mapsto 16$
- $\sqrt{-1} \mapsto i$

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

$$1 + i$$

So, what are types?

Values Have
Value Types

Definition: A *value type* is a *name* for a collection of values with common properties.

Values Have
Value Types

- Examples of value types:
  - Natural numbers
  - Integers
  - Floating point numbers
  - *And many more*

Expressions Have **Static Types**

Definition (Attempt 1): A *static type* is an assertion that an expression reduces to a value with a particular *value type.*

Expressions Have **Static Types**

$$4 + 5 : \mathbf{N} \mapsto 9 : \mathbf{N}$$

Static Type — Dynamic Type

Rules for Static Types

- If an expression is a value, its static type is its value type

$$5 : \mathbf{N}$$

- With each operator, there are “if-then” rules stating the required static types of the operands, and the static type of the application:

Integer Addition: If the operands to + are of type N then the application is of type N

Expressions Have
Static Types

Definition (Attempt 1): A *static type* is an *assertion* that an expression reduces to a value with a particular *dynamic type.*

Not quite.

Expressions Have **Static Types**

$$16 / 20 : \mathbb{Q} \mapsto 0.8 : \mathbb{Q}$$

So far, so good…

Expressions Have **Static Types**

$16 / 0 : \mathbf{Q} \mapsto ?$

Expressions Have **Static Types**

Definition (Attempt 2): A *static type* is an *assertion* that either an expression reduces to a value with a particular *value type*, or one of a <u>well-defined</u> set of exceptional events occurs.

Why Static Types?

- Using our rules, we can determine whether an expression has a static type
  - If it does, we say the expression is *well-typed,* and we know that proceeding with our computation is *type safe:*
    - Either our computation will finish with a value of the determined value type, or one of a well-defined exceptional events will occur

What Constitutes the Set of Well-Defined Exceptional Events in Arithmetic?

- A “division by zero” error
- What else?

What are the Well-Defined Exceptional Events in Arithmetic?

- A “division by zero” error
- What if we run out of paper?
  - Or pencil lead? Or erasers?
- What if we run out of time?

What Constitutes the Set of Well-Defined Exceptional Events in Arithmetic?

- A “division by zero” error
- We run out of some finite resource

Our Second Exposure to Computation:

Algebra

Now, We Learn How to Define Our Own Operators (a.k.a. functions)

$$f(x) = 2x + 1$$

$$f(x, y) = x^2 + y^2$$

And We Learn How to Compute With Them

$$\frac{f(x) = 2x + 1}{\begin{array}{c} f(3 + 2) \mapsto \\ f(5) \mapsto \\ (2 \times 5) + 1 \mapsto \\ 10 + 1 \mapsto \\ 11 \end{array}}$$

The Substitution Rule of Computation

- To reduce an application of a function to a set of arguments:
  - Reduce the arguments, left to right
  - Reduce the body of the function, with each parameter replaced by the corresponding argument

Using the Substitution Rule

$$f(x, y) = x^2 + y^2$$

$$f(4 - 5, 3 + 1) \mapsto$$

$$f(-1, 3 + 1) \mapsto$$

$$f(-1, 4) \mapsto$$

$$-1^2 + 4^2 \mapsto$$

$$1 + 16 \mapsto$$

$$17$$

What About Types?

- Eventually, we learn that our functions need to include rules indicating the required types of their arguments, and the types of applications
- You might have seen notation like this in a math class:

$$f\colon \mathbf{Z} \rightarrow \mathbf{Z}$$

Typing Rules for Functions

$$f\colon \mathbf{Z} \to \mathbf{Z}$$

What does this rule mean?

Typing Rules for Functions

$$f: \mathbf{Z} \rightarrow \mathbf{Z}$$

- We can interpret the arrow as denoting data flow:

The function f consumes arguments with value type $\mathbf{Z}$ *and produces values with value type* $\mathbf{Z}$

(or one of a well-defined set of exceptional events occurs).

Typing Rules for Functions

$$f\!: \mathbf{Z} \rightarrow \mathbf{Z}$$

- We can also interpret the arrow as logical implication:

If f is applied to an argument expression with static type $\mathbf{Z}$ *then the application expression has static type* $\mathbf{Z}$.

What are The Exceptional Events in Algebra?

- A “division by zero” error
- We run out of some finite resource
- What else?

The Substitution Rule Allows for Computations that Never Finish

$f: \mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$

$$\frac{f(x, y) = f(x, y)}{f(4 - 5, 3 + 1) \mapsto}$$

$f(-1, 3 + 1) \mapsto$

$f(-1, 4) \mapsto$

$f(-1, 4) \mapsto$

…

The Substitution Rule Allows for Computations that Keep Getting Larger

$f\colon \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$

$$\frac{f(x, y) = f(f(x, y), f(x, y))}{}$$

$f(4 - 5, 3 + 1) \mapsto$

$f(-1, 3 + 1) \mapsto$

$f(-1, 4) \mapsto$

$f(f(-1, 4), f(-1, 4)) \mapsto$

$f(f(f(-1, 4), f(-1, 4)), f(f(-1, 4), f(-1, 4))) \mapsto$

…

But We Need at Least Limited Recursion to Define Common Algebraic Constructs

$$!: \mathbf{N} \to \mathbf{N}$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

What are The Exceptional Events in Algebra?

- A “division by zero” error
- We run out of some finite resource
- The computation never stops (unbounded time)
- The computation keeps getting larger (unbounded space)

Our Third Exposure to Computation:

Core Scala

Core Scala

- We will continue to use algebra as our model of computation
- We will switch to Scala syntax
- We will introduce new value types

Value Types in Core Scala

`Int: -3, -2, -1, 0, 1, 2, 3`

`Double: 1.414, 2.718, 3.14`

`Boolean: false, true`

`String: “Hello, world!”`

Primitive Operators on Ints and Doubles in Core Scala

Algebraic operators:

`e + e′` `e - e′` `e * e′` `e / e′`

- For each operator:
  - If both arguments to an application of an operator are of type Int then the application is of type Int
  - If both arguments to an application of an operator are of type Double then the application is of type Double

Primitive Operators on Ints and Doubles in Core Scala

Comparison operators:

```
e == e′        e <= e′        e >= e′

       e > e′        e < e′
```

- For each operator:
  - If both arguments to an application of an operator are of type Int then the application is of type Boolean
  - If both arguments to an application of an operator are of type Double then the application is of type Boolean

Some Primitive Operators on Booleans in Core Scala

Conjunction, Disjunction:

```
e & e'        e | e'
```

- In both cases:
  - If both arguments to an application are of type Boolean then the application is of type Boolean

More Primitive Operators on Booleans in Core Scala

Negation:

```
!e
```

- If the argument to an application is of type Boolean then the application is of type Boolean

Yet More Primitive Operators on Booleans in Core Scala

Conditional Expressions:

```
if (e) e′ else e′′
```

- If the first argument is of type Boolean and the second and third argument are of the same type T then the application is of type T

Primitive Operators on Strings in Core Scala

String Concatenation:

$$e \ + \ e'$$

- If both arguments are of type String then the application is of type String

An Example Function Definition in Core Scala

```
def square(x: Double) = x * x
```

Syntax for Defining Functions

```
def fnName(arg0: type0, …, argk: typek):returnType =

      expr
```

- If there is no recursion, we do not need to declare the return type:

```
def fnName(arg0: type0, …, argk: typek) =

      expr
```

The Substitution Rule Works as Before

```
def square(x: Double) = x * x

square(2.0 * 3.0) ↦
square(6.0) ↦
6.0 * 6.0 ↦
36.0
```

The Nature of Ints

Fixed Size Ints

- Unlike the integers we might write on a sheet of paper, the values of type Int are of a fixed size
- For every n: Int,

$$-2^{31} \leq n \leq 2^{31}-1$$

Fixing the Size of Numbers Has Many Benefits

- The time needed to compute the application of an operation on two numbers is bounded
- The space needed to store a number is bounded
- We can easily reuse the space used for one number to store another

But We Need to Concern Ourselves with Overflow

- If we compute a value larger than 2^{31}-1, our representation will “wrap around”

```
2147483647 + 1 ↦ -2147483648
```

The Moral of Computing with Ints

- If possible, determine the range of potential results of a computation
    - Ensure that this range is no larger than the range of representable values of type Int
- Otherwise, include in your computation a check for overflow

The Nature of Doubles

Scientific Notation

- Numeric values in scientific computations can span enormous ranges, from the very large to the very small
- At the same time, scientific measurements are of limited precision
- “Scientific notation” was devised in order to efficiently represent approximate values that span a large range

Scientific Notation

$$6.022 \times 10^{23}$$

mantissa exponent

Scientific Notation and Efficient Computation

- We normalize the mantissa so that its value is at least 1 but less than 10
- If we
  - Set the number of digits in the mantissa to a fixed precision, and
  - Set the number of digits in the exponent to a fixed precision
- Then all numbers in our notation are of a fixed size

Doubles

- Values of type Double are stored as with fixed sized numbers in scientific notation, but with a few differences:
    - Finite, nonzero numeric values can be expressed in the form:

$$\pm\, m\ 2^{e}$$

Doubles

$$\pm\, m\, 2^{e}$$

- $1 \leq m \leq 2^{53}-1$
- $-2^{10}-53+3 \leq e \leq 2^{10}-53$

Doubles

$$\pm\, m\, 2^{e}$$

- $1 \leq m \leq 2^{53}-1$
- $-2^{10}-53+3 \leq e \leq 2^{10}-53$
- $-1074 \leq e \leq 971$