# Comp 311
# Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

# Homework 0

- Please follow these instructions for checking out your **turnin** repository as soon as possible:
  - Follow the instructions under Homework Submission Guide at the Course Website
  - Submit a **hw_0** folder with a single file **HelloWorld.txt** and a single line of text, **Hello, world!**
  - This submission is not for credit
  - We will let you know if we have not received your submission
  - You will be responsible for successfully submitting your **hw_1** assignment using **turnin**
  - Please bring problems to our attention as soon as possible

# *What is functional programming? (continued)*

# 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.

  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*

# Our First Exposure to Computation: Arithmetic

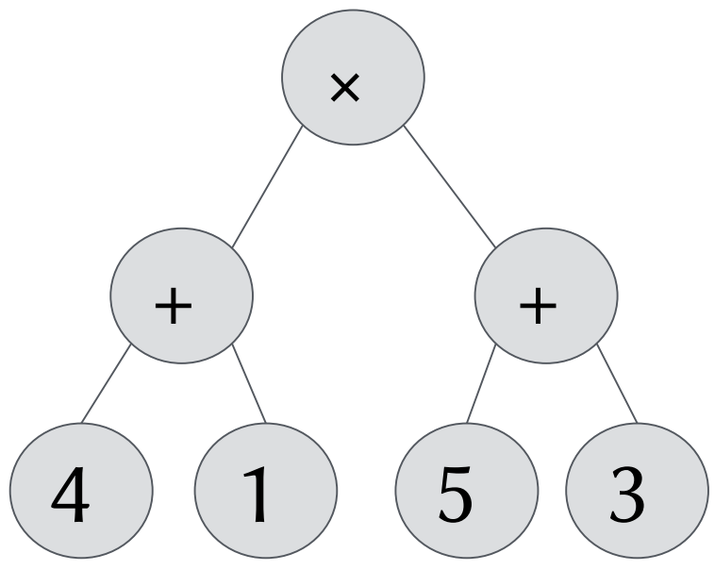
$$4 + 5 = 9$$

$$4 + 5 \longmapsto 9$$

**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: $(4 + 1) \times (5 + 3)$



# 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.*,
  $2 + 3 \times 6$ *vs.* $2 \times 3 + 6$
- Thinking about syntax in terms of ASTs simplifies reduction rules

# Expressions are Reduced to Values

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

# Expressions are Reduced to Values

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

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

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

$$5 \times 8 \longmapsto$$

$$40$$

# Expressions are Reduced to Values

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

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

$$4 + 5 + 3 \longmapsto$$

$$9 + 3 \longmapsto$$

$$12$$

# 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**

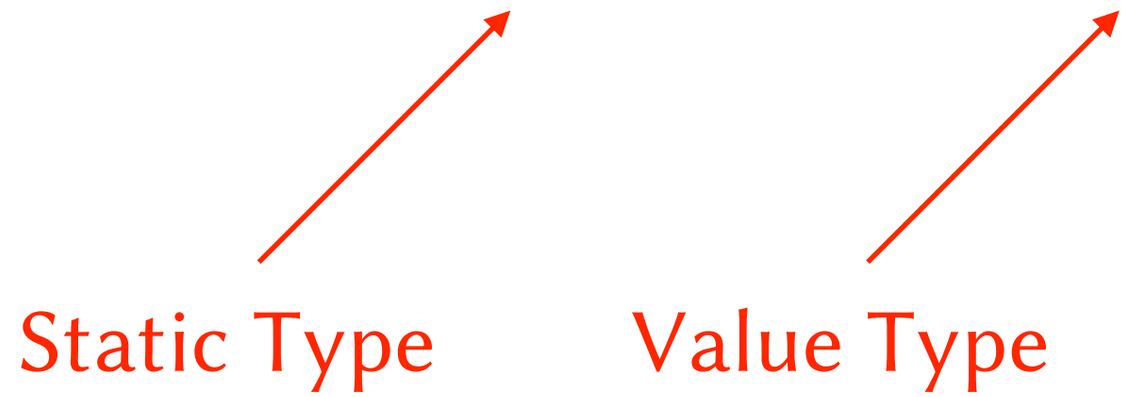
- Examples of value types:
  - Natural numbers
  - Integers
  - Floating point numbers
  - *And many more*

# 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**

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



# 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 *value type.*

Not quite.

# Expressions Have **Static Types**

$$16 / 20 : \mathbf{Q} \longmapsto 0.8 : \mathbf{Q}$$

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 well-defined 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

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

---

$$f(3 + 2) \longmapsto$$

$$f(5) \longmapsto$$

$$(2 \times 5) + 1 \longmapsto$$

$$10 + 1 \longmapsto$$

$$11$$

# 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:

$$f: \mathbb{Z} \longrightarrow \mathbb{Z}$$

# Typing Rules for Functions

$$f: \mathbb{Z} \longrightarrow \mathbb{Z}$$

What does this rule mean?

# Typing Rules for Functions

$$f: \boldsymbol{Z} \longrightarrow \boldsymbol{Z}$$

- We can interpret the arrow as denoting data flow:

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

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

# Typing Rules for Functions

$$f: \mathbb{Z} \longrightarrow \mathbb{Z}$$

- We can also interpret the arrow as logical implication:

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

# 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

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

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

$$\mathrm{f}(-1, 3 + 1) \longmapsto$$

$$\mathrm{f}(-1, 4) \longmapsto$$

$$\mathrm{f}(-1, 4) \longmapsto$$

$$\ldots$$

# The Substitution Rule Allows for Computations that Keep Getting Larger

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

$$f(x, y) = f(f(x, y), f(x, y))$$

---

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

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

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

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

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

...

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

$$\begin{array}{c} \text{!:} \\ \mathbf{N} \longrightarrow \mathbf{N} \end{array}$$

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

# 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′ 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 may elide 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" (i.e., overflow):

$$2147483647 + 1 \longmapsto -2147483648$$

# 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

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

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

# Doubles

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

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

# 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

$$\underbrace{6.022}_{\text{mantissa}} \times \underbrace{10}_{\text{base}}\overbrace{^{23}}^{\text{exponent}}$$

# 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:

$$\pm m \times 2^e$$

# Doubles

$$\pm m \times 2^e$$

- $1 \leq m \leq 2^{53} - 1$
- $-1022 \leq e \leq 971$

For more details, you can read about double-precision binary representation: https://en.wikipedia.org/wiki/Double-precision_floating-point_format

# Doubles

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

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

# Representations of Doubles

- Many quantities have more than one representation in this format:

$$1024 \times 2^{500}$$

$$512 \times 2^{501}$$

# Distances Between Doubles

- The distance between adjacent values of type Double is not constant
- The values are most dense near zero
- They grow sparser exponentially as one moves away from zero

# Operations and Rounding

- Arithmetic operations round to the closest representable value

- Ties are broken by choosing the value with the smaller absolute value

# Overflow with Doubles

- Computations on Doubles that result in values larger than the largest finite Double are represented with special values:

Double.PositiveInfinity

Double.NegativeInfinity

# Underflow with Doubles

- Computations on Doubles that result in values with magnitudes smaller than the smallest non-zero Double are represented with special values:

0.0 -0.0

# Division By Zero

- Division of a non-zero finite value by a zero value results in an infinite value:

$$1.0 \,/\, 0.0 \longmapsto \text{Double.PositiveInfinity}$$

$$1.0 \,/\, -0.0 \longmapsto \text{Double.NegativeInfinity}$$

# Division By Zero

- As does division of an infinite value by a zero value:

$$\text{Double.PositiveInfinity} / 0.0 \mapsto \text{Double.PositiveInfinity}$$

# Division By Zero

- Division of a zero value by a zero value results in another special value NaN (for “Not a Number”):

$$0.0 \,/\, 0.0 \longmapsto \text{Double.NaN}$$

$$\text{-}0.0 \,/\, 0.0 \longmapsto \text{Double.NaN}$$