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

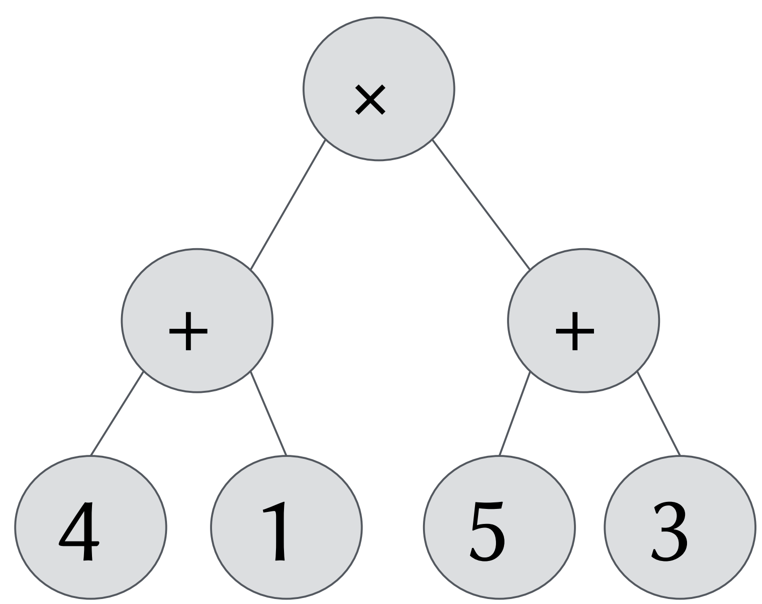
$$4 + 5 = 9$$

$$4 + 5 \mapsto 9$$

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



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:

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

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

$$5 \times 8 \mapsto$$

$$40$$

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**

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

Static Type　　　Value 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 *value type.*

Not quite.

Expressions Have **Static Types**

$$16\ /\ 20\colon \mathbf{Q} \mapsto 0.8\colon \mathbf{Q}$$

So far, so good...

Expressions Have **Static Types**

$16 \;/\; 0 \colon \mathbf{Q} \mapsto\ ?$

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

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

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

$$\frac{f(x, y) = x^2 + y^2}{f(4 - 5, 3 + 1) \mapsto}$$

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

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

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

$$1 + 16 \mapsto$$

$$17$$

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:

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

Typing Rules for Functions

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

What does this rule mean?

Typing Rules for Functions

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

- 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

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

- We can also interpret the arrow as logical implication:

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