Comp 311
Functional Programming

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

Announcements

- It is strongly recommended that you follow the 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

Announcements

Two Sigma internships and full-time positions available (Houston and New York Offices)

Type Checking

- So far, we have been rigorous about computation of programs, but we have relied on intuition for static type checking
- We can provide a *static semantics* for Core Scala along with our *dynamic semantics*

The Substitution Model of Type Checking

- To type check a value **v**, replace **v** with its value type

```
1.003 ⇒ Double
```

- To type check a constant **c**, reduce the defining expression of **c** to a static type **T**, then replace all occurrences of **c** with **T**

```
pi = 3.14 ⇒

pi : Double

pi * radius * radius ⇒

Double * radius * radius
```

The Substitution Model of Type Checking

- To type check a function definition:
  - Type check the body of the definition, replacing all occurrences of each parameter with the corresponding parameter type
- To type check the occurrence of a function name:
  - Replace the name with an *arrow type*, where the parameter types of the function are to the left of the arrow and the return type is to the right

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

square(3.14) ⇒

(Double → Double)(3.14)
```

The Substitution Model of Type Checking

- To type check the application of a function to arguments:
  - Reduce the function to an arrow type
  - Reduce the arguments, left to right, to static types
  - If the parameter types match the corresponding argument types, reduce the application to the return type

The Substitution Model of Type Checking

```
square(3.14) ⇒

(Double → Double)(3.14) ⇒

(Double → Double)(Double) ⇒

Double
```

Conditional Functions On Point Values

Conditional Functions On Point Values

- Often the cases on a conditional function must test for equality rather than whether values fall in a range
  - This is especially common with String values
  - What about Boolean values?
  - Double values should not be tested this way (why?)

Example: Days in a Month

- Given the name of a month, we want to return the number of days

Data Analysis and Definition

- We use Strings to denote months and Ints for the number of days

Contract

- We state the preconditions in documentation:

```
/**
 * Given a string identifying a month,
 * with the first (and only the first) letter capitalized,
 * returns the number of days in that month
 * for an ordinary year (non-leap) year.
 */
def days(month: String): Int = {
  …
} ensuring (_ <= 31)
```

- How can we improve the precondition? What data types would we want?

A Function Template for Conditional Functions on Point Values

```scala
/**
 * Given a string identifying a month,
 * with the first (and only the first) letter capitalized,
 * returns the number of days in that month
 * for an ordinary year (non-leap) year.
 */
def days(month: String): Int = {
  month match {
    case … => …
    …
  }
} ensuring (_ <= 31)
```

Syntax for Match

```
expr0 match {
  case Pattern => expr1
    …
  case Pattern => exprN
}
```

Primitive Value Patterns

- A primitive value pattern is either:
  - A primitive value
  - A free parameter
  - The special pattern _

Matching a Primitive Value With a Pattern

- A primitive value **v** matches:
  - Itself
  - A free parameter
  - The special pattern _
    - Should only be used as the final clause of a match (why?)

Meaning of a Match Expression

- To reduce a match expression:

```
expr0 match {
  case Pattern => expr1
    …
  case Pattern => exprN
}
```

  - Reduce **expr0** to a value **v**

  - Find the first pattern **k** matching **v** (if it exists) and reduce to **exprK** (replacing all occurrences of **k** with **v** if **k** is a free parameter)

- Failure to match a pattern results in a new form of exceptional condition

Using Match for Point Value Matching

```scala
/**
 * Given a string identifying a month,
 * with the first (and only the first) letter capitalized,
 * returns the number of days in that month
 * for an ordinary year (non-leap) year.
 */
def days(month: String): Int = {
  month match {
    case "January" => 31
    case "February" => 28
    case "March" => 31
    case "April" => 30
    case "May" => 31
    case "June" => 30
    case "July" => 31
    case "August" => 31
    case "September" => 30
    case "October" => 31
    case "November" => 30
    case "December" => 31
  }
} ensuring (_ <= 31)
```

Reducing Match

```
days(“September”)
↦

“September” match {
    case "January" => 31
    case "February" => 28
    case "March" => 31
    case "April" => 30
    case "May" => 31
    case "June" => 30
    case "July" => 31
    case "August" => 31
    case "September" => 30
    case "October" => 31
    case "November" => 30
    case "December" => 31
  }
} ensuring (_ <= 31)
↦

30
```

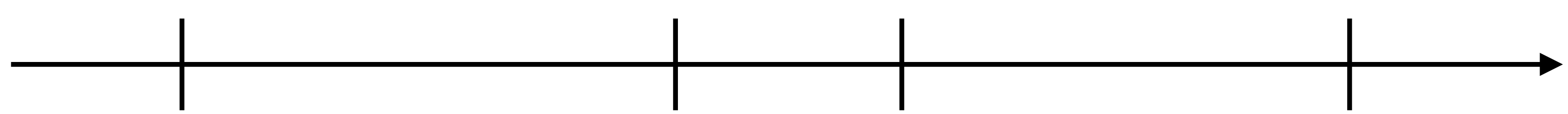
A Match With a Free Parameter

```scala
def plural(word: String): String = {
  word match {
    case "deer" => "deer"
    case "fish" => "fish"
    case "mouse" => "mice"
    case x => x + "s"
  }
```

Conditional Functions On Intervals

Conditional Functions On Intervals

- Often a computation falls into distinct cases depending on which of a finite set of intervals a value falls into
  - In such cases, it can help to break the number line into distinct regions that we must handle separately:



Designing Conditional Functions

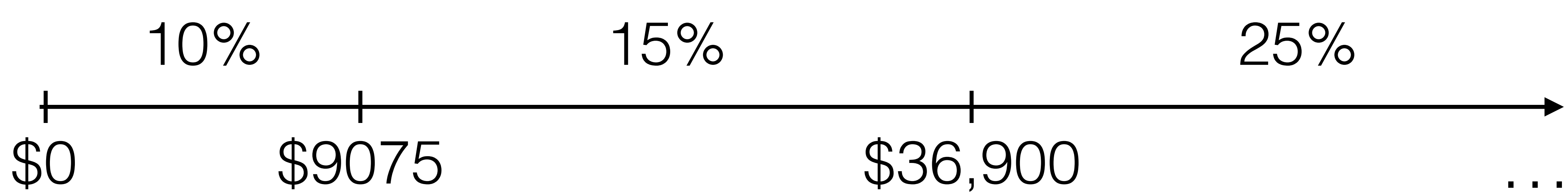
- Example: Graduated Income Tax (Single Filer):
  - Up to $9,075: 10%
  - $9,075 to $36,900: 15%
  - $36,901 to $89,350: 25%
  - $89,351 to 186,350: 28%
  - $186,351 to $405,100: 33%
  - $405,101 to $406,750: 35%
  - $405,751 or more: 39.6%
- We follow the Design Recipe

Graduated Income Tax: Data Analysis and Definition

- We use Ints to denote U.S. Dollar values and tax percentages (using integer division by 100 as a last step)

- Both income and tax should be non-negative

- We break the number line into the relevant intervals



Contract

```
/**
 * Given an income in U.S. Dollars,
 * returns the dollar value of tax
 * owed for a single tax payer, using
 * 2014-2015 IRS tax brackets.
 */
def incomeTax(income: Int) = {
  require(income >= 0)
  …
} ensuring (_ >= 0)
```

Function Application Examples

- We should develop at least one example per case, as well as borderline cases

```
100 = incomeTax(1000)

907 = incomeTax(9075)

907 + 138 = incomeTax(10000)

…
```

Our Function Template for Conditional Functions

```
/**
 * Given an income in U.S. Dollars,
 * returns the dollar value of tax
 * owed for a single tax payer, using
 * 2014-2015 IRS tax brackets.
 */
def incomeTax(income: Int): Int = {
  require(income >= 0)

  if (income <= cutoff0) {
    …
  } else if (income <= cutoff1) {
    …
  } else if (income <= cutoff2) {
    …
  } else if (income <= cutoff3) {
    …
  } else if (income <= cutoff4) {
    …
  } else if (income <= cutoff5) {
    …
  } else if (income <= cutoff6) {
    …
  } else { // income > cutoff6
    …
  }
} ensuring (_ >= 0)
```

Defining Our Constant Values in One Place

```
val bracket0 = 0
val cutoff0 = 0

val bracket1 = 100
val cutoff1 = 9075

val bracket2 = 150
val cutoff2 = 36900

val bracket3 = 250
val cutoff3 = 89350

val bracket4 = 280
val cutoff4 = 186350

val bracket5 = 330
val cutoff5 = 405100

val bracket6 = 350
val cutoff6 = 406750

val bracket7 = 396
val cutoff7 = Int.MaxValue
```

As We Fill In Cases, We Find a Common Pattern

```
/**
 * Given:
 *   an income in U.S. Dollars
 *   the next lowest cutoff in U.S. Dollars
 *   a tax percentage for the bracket above the cutoff
 * Returns the income tax due for the given income
 */
def incomeTaxForBracket(income: Int, cutoff: Int, bracket: Int) = {
  require(income >= 0)
  (income - cutoff) * bracket / divisor + incomeTax(cutoff)
} ensuring (_ >= 0)
```

And Now We Call This New Function to Fill in the The Income Tax Function Template

```scala
/**
 * Given an income in U.S. Dollars, returns the dollar value of tax
 * owed for a single tax payer, using 2014-2015 IRS tax brackets.
 */
def incomeTax(income: Int): Int = {
  require(income >= 0)

  if (income <= cutoff0) {
    bracket0
  } else if (income <= cutoff1) {
    incomeTaxForBracket(income, cutoff0, bracket1)
  } else if (income <= cutoff2) {
    incomeTaxForBracket(income, cutoff1, bracket2)
  } else if (income <= cutoff3) {
    incomeTaxForBracket(income, cutoff2, bracket3)
  } else if (income <= cutoff4) {
    incomeTaxForBracket(income, cutoff3, bracket4)
  } else if (income <= cutoff5) {
    incomeTaxForBracket(income, cutoff4, bracket5)
  } else if (income <= cutoff6) {
    incomeTaxForBracket(income, cutoff5, bracket6)
  } else { // income > cutoff6
    incomeTaxForBracket(income, cutoff6, bracket7)
  }
} ensuring (_ >= 0)
```

Remarks On Conditional Functions

- The clauses in a conditional function need not all have the same form
- Avoid factoring out code into a helper function until there is more than one place to call the helper
- There is more we can factor out in this example, but first we will need more powerful language features (stay tuned)

Compound Datatypes

Compound Datatypes

- Although many computations can be performed on primitive data types, it is often useful to combine data into larger structures
- We call all data of this form *compound data*
- The two simplest compound datatypes in Core Scala are tuples and arrays

Tuple Values

- A tuple value contains a sequence of values

```
(v1, …, vN)
```

- There is one empty tuple `()`
- Tuples of length one do not exist (why?)
- The value type of a tuple is simply the tuple of the corresponding value types

```
(T1, …, TN)
```

Tuple Types

- The empty tuple has the special type `Unit`
- The static type of a tuple expression:

`(e1, … eN)`

is

`(T1, …, TN)`

where

`e1: T1, … eN: TN`

Tuple Types

- Tuple types allow us to combine data of distinct types. For example:

```
(Int, Boolean, String)
```

- However, tuple types restrict the length of any corresponding tuple value

Accessing Tuple Elements

- We can access the **kth** element of an expression **e** with static type **(T1, …, TN)** using the syntax:

```
e._k
```

- The static type of this expression is Tk

- Note that tuples are 1-indexed

- Example:

```
(1,2,3)._2 ↦ 2
```

Accessing Tuple Elements

- We can access the elements of a tuple using match expressions
  - We add the following syntactic form to our definition of patterns

```
(Pattern1, … , PatternN)
```

- We call this new syntactic form a *tuple pattern*

Accessing Tuple Elements

- A tuple matches a tuple pattern iff each element of the tuple matches a corresponding element of the tuple pattern

Income Tax Revisited

```
def incomeTaxForBracketCutoff(income: Int, bracketCutoff: (Int, Int)) = {
  require(income >= 0)

  bracketCutoff match {
    case (bracket, cutoff) => {
      (income - cutoff) * bracket /
        divisor + incomeTax(cutoff)
    }
  }
} ensuring (_ >= 0)
```

Tuple Types and Arrow Types

- We can now view every arrow type as taking exactly one parameter:
- Example:

```
(Int, String, Boolean) → Int
```

Tuple Types and Arrow Types

- We can also use tuple types to denote that a function returns “multiple values”:

- Example:

```
(Int, String, Boolean) → (Int, Double)
```

Array Values

- An array is a sequence of values all of the same value type

```
Array(1,2,3)
```

Array Types

- If the elements of an array value are of type T then the array is of type Array[T]

- If the expressions `e1, … , eN` are of static type `T` then the expression

```
Array(e1, …, eN)
```

- has static type

```
Array[T]
```

Array Types

- Array types require that all elements of an array share a common type
- However, array types match array values of any length
- Contrast with tuple types

Accessing Array Values

- We can access the `kth` element of an expression of type `Array[T]` with the syntax:

```
expr(k)
```

- The static type of this expression is `T`
- Note that arrays are zero-indexed
- Example:

```
Array(1,2,3)(2) ↦ 3
```

Accessing Array Elements

- We can access the elements of an array using match expressions
  - We add the following syntactic form to our definition of patterns:

```
Array(Pattern1, … , PatternN)
```

- We call this new syntactic form an *array pattern*

Accessing Array Elements

- An array matches an array pattern iff each element of the array matches a corresponding element of the array pattern

Accessing Array Elements

```
def sumOfSquares(coordinates: Array[Int]) = {
  coordinates match {
    case Array(x,y,z) => x*x + y*y + z*z
  }
}
```

Structural Data

Structural Data

- Tuples and arrays allow us to combine multiple primitive values into a single data value
- However,
  - They do not allow us to attach names to the constituent elements
  - They do not allow us to distinguish elements of conceptually distinct datatypes

Case Classes

- We can think of a case class as a tuple with its own *type* and *accessors* for its elements

Case Classes

```
case class Coordinate(x: Int, y: Int)
```

Simple Syntax for Case Classes

```
case class Name(field1: Type1, …, fieldN: TypeN)
```

Creating Instances of a Case Class

- We construct new instances of a case class

  `case class C(field1: Type1, …, fieldN: TypeN)`

- with the syntax

  `C(expr1, …, exprN)`

- To reduce this expression, reduce each argument `exprK` to a value `vK`, forming the value `C(v1, …, vN)`

- If the types of `expr1,…,exprN` match the types of the corresponding fields, then this expression has type `C`

Accessing Fields of a Case Class

- Given a case class:

```
case class C(field1: Type1, …, fieldN: TypeN)
```

- We can access field with name `fieldK` of `C` with the expression syntax:

```
C.fieldK
```

- The static type of this expression is `TypeK`

Accessing Fields of a Case Class

```
def magnitude(coordinate: Coordinate) = {
  coordinate.x * coordinate.x +
  coordinate.y * coordinate.y
}
```

Accessing Class Elements

- We can access the elements of a case class instance using match expressions
  - For each case class, we add the following syntactic form to our definition of patterns

```
C(Pattern1, … , PatternN)
```

- We call this new syntactic form a *class pattern*

Accessing Case Class Elements

- An instance of a case class `C(v1, …, vN)` matches a class pattern `C(P1, …, PN)` iff
  - The class name is identical to the class pattern name
  - Each element of the instance matches a corresponding element of the class pattern

Accessing Case Class Elements

```scala
def magnitude(coordinate: Coordinate) = {
  coordinate match {
    case Coordinate(x,y) => x*x + y*y
  }
}
```

Class Methods

- Methods are functions defined in the body of a class definition. They have direct access to the members of a class instance

- Syntactically, they are placed between braces, after the class parameters

Class Methods

```
case class C(field1: Type1, …, fieldN: TypeN) {
  def m1(x11: TypeP11, … xK1: TypePk1): TypeR11 =
      expr
  …
  def mJ(x1J: TypeP1J, … xKJ: TypePkJ): TypeR1J =
      expr
  }
```

Method Definitions

```
case class Coordinate(x: Int, y: Int) {
  def magnitude() = x*x + y*y
}
```

Applying a Class Method

- Given a class definition:

```
class C(p1:T1, …, pk:Tk) { …
    def m(param1:T11, paramN:T1N):T = e
    …
}
```

- To reduce the application of a method:

```
C(v1, …, vk).m(arg1, …, argN)
```

  - Reduce the receiver and arguments, left to right
  - Reduce the body of `m`, replacing parameters `p1, …, pk` with v`1,…,vk` and `param1, …, paramN` with `arg1, …, argN`

Applying a Class Method

```
Coordinate(5,3).magnitude() ↦
5*5 + 3*3 ↦
25 + 9 ↦
34
```

Nested Pattern Matching

```
def dotProduct(c1: Coordinate, c2: Coordinate) = {
  (c1, c2) match {
    case (Coordinate(x1,y1), Coordinate(x2,y2)) =>
      x1*x2 + y1*y2
  }
}
```