

# Comp 311

# Functional Programming

Eric Allen, Two Sigma Investments  
Robert “Corky” Cartwright, Rice University  
Sagnak Tasirlar, Two Sigma Investments

# Announcements

- Homework 1 will be assigned next Thursday
- Watch “Working Hard to Keep it Simple” Available on the course website

# Including Constant Definitions

- We can include constant definitions in functions using `val`
- We refer to expressions prefixed with a sequence of constant definitions as compound expressions

Place After The Requires Clause  
and Before the “Result” Expression

```
def cost(ticketPrice: Int) = {  
  require (ticketPrice >= 0 & ticketPrice <= 1000)  
  
  val fixedCost = 18000  
  val perAttendeeCost = 4  
  
  fixedCost + perAttendeeCost * attendance(ticketPrice)  
} ensuring (_ >= 0)
```

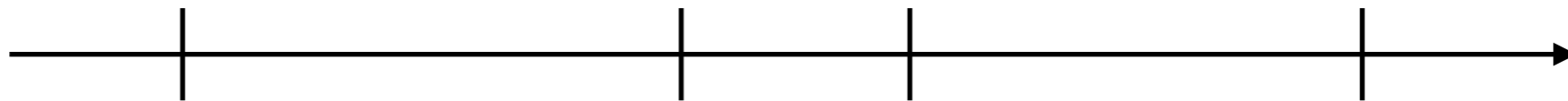
# To Reduce A Compound Expression

- First compute the value of each constant definition, top to bottom
- Then reduce the result expression, replacing each occurrence of a constant name with its computed value

# Conditional Functions On Ranges

# Conditional Functions On Ranges

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



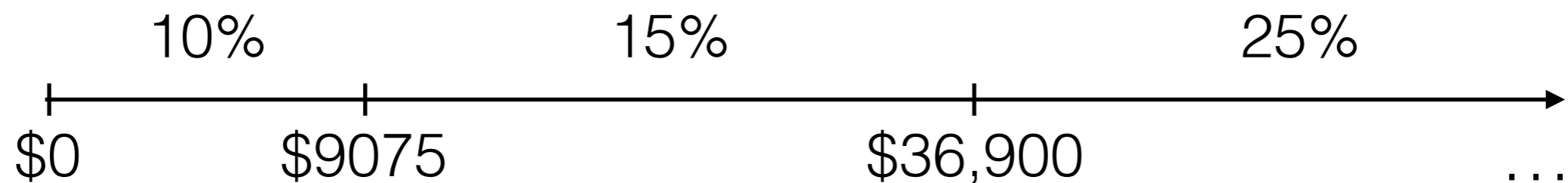
# Designing Conditional Functions

- 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

$$100 = \text{incomeTax}(1000)$$

$$907 = \text{incomeTax}(9075)$$

$$907 + 138 = \text{incomeTax}(10000)$$

...

# 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

```
/**
 * 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)



# 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

```
/**  
 * 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

```
/**
 * 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

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

# 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

$(v_1, \dots, v_N)$

- 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

$(T_1, \dots, T_N)$



# Tuple Types

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

$(e_1, \dots, e_N)$

is

$(T_1, \dots, T_N)$

where

$e_1: T_1, \dots, e_N: T_N$

# 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  $T_k$
- Note that tuples are 1-indexed
- Example:

$(1, 2, 3)._2 \mapsto 2$

# 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  $\text{Array}[T]$
- If the expressions  $e_1, \dots, e_N$  are of static type  $T$  then the expression

$\text{Array}(e_1, \dots, e_N)$

- has static type

$\text{Array}[T]$

# 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 **expr<sub>k</sub>** to a value **v<sub>k</sub>**, forming the value **C(v<sub>1</sub>, ..., v<sub>N</sub>)**
- If the types of **expr<sub>1</sub>, ..., expr<sub>N</sub>** 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 an instance `C(v1, ..., vN)` with the expression syntax:

```
C(v1,...,vN).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

$C(\text{Pattern1}, \dots, \text{PatternN})$

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



# Accessing Case Class Elements

- An instance of a case class  $C(v_1, \dots, v_N)$  matches a class pattern  $C(P_1, \dots, P_N)$  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

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