

# Comp 311

# Functional Programming

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

September 4, 2018

# Announcements

- Homework 0 is “due” on Thursday
- Homework 1 will also be assigned on Thursday

*Review:*  
*Computation by Reduction*

# Arithmetic Operations

Operation	Static Type	Examples
$v_1 + v_2$ $v_1 - v_2$ $v_1 * v_2$ $v_1 / v_2$	$\text{Int} \times \text{Int} \rightarrow \text{Int}$ $\text{Double} \times \text{Double} \rightarrow \text{Double}$	$5 - 1 \mapsto 4$ $9 / 0 \mapsto \perp$ $9.0 / 0.0 \mapsto$ $\text{Double.PositiveInfinity}$
$+ v$ $- v$	$\text{Int} \rightarrow \text{Int}$ $\text{Double} \rightarrow \text{Double}$	$-(0) \mapsto 0$ $+(-7) \mapsto -7$ $-(-7) \mapsto 7$ $-(0.0) \mapsto -0.0$
$v.\text{toDouble}$	$\text{Int} \rightarrow \text{Double}$	$3.\text{toDouble} \mapsto 3.0$

# Comparison Operations

Operation	Static Type	Examples
$v_1 == v_2$ $v_1 != v_2$	$\tau \times \tau \rightarrow \text{Boolean}$ $v_1: \tau \quad v_2: \tau$	"x" == "x" $\mapsto$ true false != false $\mapsto$ false (-0.0) == 0.0 $\mapsto$ true
$v_1 < v_2$ $v_1 <= v_2$ $v_1 > v_2$ $v_1 >= v_2$	$\text{Int} \times \text{Int} \rightarrow \text{Boolean}$ $\text{Double} \times \text{Double} \rightarrow \text{Boolean}$	1 < 1 $\mapsto$ false 5 > 4 $\mapsto$ true Double.NegativeInfinity <= Double.NaN $\mapsto$ false

# Logical Operations

Operation	Static Type	Examples
$v_1 \& v_2$ $v_1 \mid v_2$	$\text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean}$	$\text{true} \& \text{true} \mapsto \text{true}$ $\text{true} \& \text{false} \mapsto \text{false}$ $\text{false} \mid \text{true} \mapsto \text{true}$
$!v$	$\text{Boolean} \rightarrow \text{Boolean}$	$! \text{true} \mapsto \text{false}$ $! \text{false} \mapsto \text{true}$

# Function Applications

Given a function definition

```
def fn( $x_0: T_0, x_1: T_1, \dots, x_N: T_N$ ):  $T_R = \{$   
   $\text{expr}_{\text{body}}$   
}
```

we get a corresponding reduction rule:

$$\text{fn}(v_0, v_1, \dots, v_N) \mapsto \{ \text{expr}_{\text{body}}[x_0 \mapsto v_0, x_1 \mapsto v_1, \dots, x_N \mapsto v_N] \}$$

i.e., the function application reduces to the function body expression, but with a new rule for each formal parameter's symbol, reducing the symbol to the corresponding argument value from the application.

# Function Application Example

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

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



# *Conditional Expressions*

# Computing Conditional Expressions

- We used a slight of hand when presenting `if` expressions

`if (e1) e2 else e3`

- According to the substitution model of computation, how do we compute the value of this expression?

# Computing Conditional Expressions

`if (e1) e2 else e3`

- First we compute  $e1 \mapsto v1$ , then  $e2 \mapsto v2$ , then  $e3 \mapsto v3$
- If  $v1$  is true then reduce to  $v2$
- Otherwise reduce to  $v3$

# But Consider the Following Expression

```
if (false) 1/0 else 3
```

This expression should reduce to 3

# New Rule for Conditional Expressions

- To reduce an if expression:
  - Reduce the `test` clause
    - If the test clause reduces to `true`, reduce the `then` clause
    - Otherwise, reduce the `else` clause

# Short-Circuiting Logical Operations

Rule	Static Type
$\text{if (true) } \text{expr}_1 \text{ else } \text{expr}_2 \mapsto \text{expr}_1$ $\text{if (false) } \text{expr}_1 \text{ else } \text{expr}_2 \mapsto \text{expr}_2$	$\text{Boolean} \times \tau \times \tau \rightarrow \tau$ $\text{expr}_1: \tau \quad \text{expr}_2: \tau$
$\text{true} \ \&\& \ \text{true} \mapsto \text{true}$ $\text{true} \ \&\& \ \text{false} \mapsto \text{false}$ $\text{false} \ \&\& \ \text{expr}_2 \mapsto \text{false}$ $\text{true} \ \ \ \text{expr}_2 \mapsto \text{true}$ $\text{false} \ \ \ \text{true} \mapsto \text{true}$ $\text{false} \ \ \ \text{false} \mapsto \text{false}$	$\text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean}$

# *Defining Constants*

# Including Constant Definitions

- We can include constant definitions within functions by using `val`

```
    val x = expr  
val y: Type = expr
```

- 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 (0 <= ticketPrice & ticketPrice <= 1000)  
  
  val fixedCost: Int = 18000  
  val perAttendeeCost: Int = 4  
  
  fixedCost + perAttendeeCost * attendance(ticketPrice)  
} ensuring (result => 0 <= result)
```

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

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

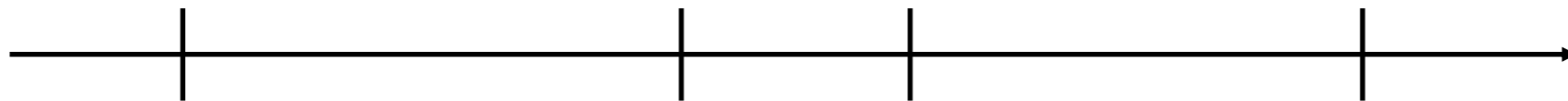
# Reducing Compound Expressions

- First, compute the value of each constant definition, reducing top to bottom, left to right
- Then reduce the result expression, substituting 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

- First 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%

# Designing Conditional Functions

- Second Example – ASCII Character Classes

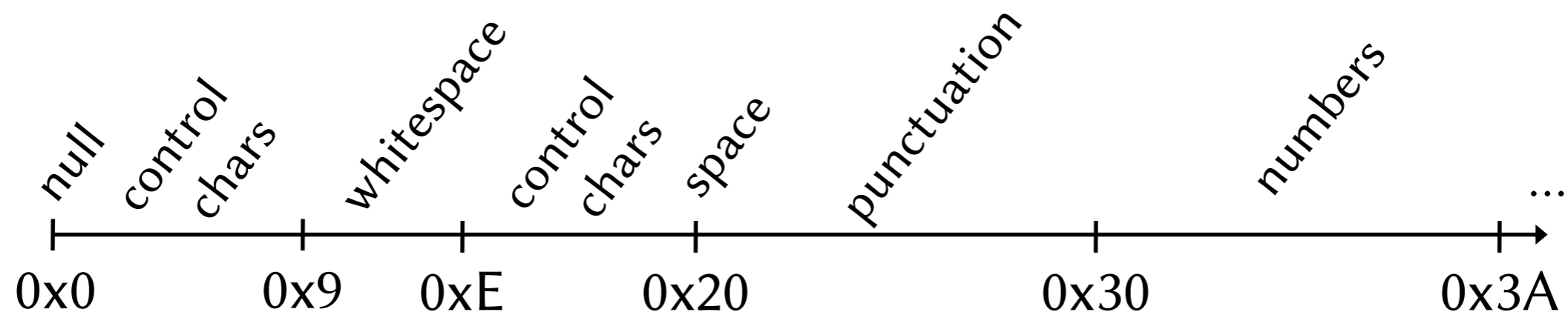
N U L S O H S T X E T X E O T E N Q A C K B E L B S H T L F V T F C R S S I D L E C 1 C 2 C 3 C 4 N A S Y N E T B C A N E M S U B E S C F S G S R S U S

**!"#\$%&'()\*+,-./0123456789:;<=>?**  
**@ABCDEFGHIJKLMNO PQRSTUVWXYZ[\]^\_**  
**`abcdefghijklmnopqrstuvwxyz{|}~<sup>DEL</sup>**

- We follow the Design Recipe

# ASCII Codepoint Descriptions: Data Analysis and Definition

- We use `Ints` to ASCII character values.
- We use `Strings` to describe the character class of a given ASCII codepoint.
- ASCII characters are defined in the range `[0, 127]`.
- We break the number line into the relevant intervals:





# Contract

```
/**  
 * Given an ASCII character codepoint,  
 * return a String describing the type of  
 * character represented by that codepoint.  
 */  
def describeAsciiChar(char: Int): String = {  
  require(0 <= char & char <= 127)  
  ...  
} // no ensuring clause
```

# Function Application Examples

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

```
"Null" == describeChar(0)
```

```
"Control Character" == describeChar(10)
```

```
"Lowercase Letter" == describeChar(97)
```

```
...
```

# Our Function Template for Conditional Functions

```
/**
 * Given an ASCII character codepoint,
 * return a String describing the type of
 * character represented by that codepoint.
 */
def describeAsciiChar(char: Int): String = {
  require(0 <= char & char <= 127)

  if (char == 0x0) { "Null" }
  else if (char <= 0x08) { "Control Character" }
  else if (char <= 0x0D) { "Whitespace" }
  else if (char <= 0x19) { "Control Character" }
  else if (char == 0x20) { "Whitespace" }
  else if (char <= 0x29) { "Punctuation" }
  else if (char <= 0x39) { "Number" }
  else if (char <= 0x40) { "Punctuation" }
  else if (char <= 0x5A) { "Uppercase Letter" }
  else if (char <= 0x60) { "Punctuation" }
  else if (char <= 0x7A) { "Lowercase Letter" }
  else if (char <= 0x7E) { "Punctuation" }
  else { "Control Character" } // 0x7F
}
```

# Our Function Template for Conditional Functions

```
/**
 * Given an ASCII character codepoint,
 * return a String describing the type of
 * character represented by that codepoint.
 */
def describeAsciiChar(char: Int): String = {
  require(0 <= char & char <= 127)

  if (char == 0x0) "Null"
  else if (char <= 0x08) "Control Character"
  else if (char <= 0x0D) "Whitespace"
  else if (char <= 0x19) "Control Character"
  else if (char == 0x20) "Whitespace"
  else if (char <= 0x29) "Punctuation"
  else if (char <= 0x39) "Number"
  else if (char <= 0x40) "Punctuation"
  else if (char <= 0x5A) "Uppercase Letter"
  else if (char <= 0x60) "Punctuation"
  else if (char <= 0x7A) "Lowercase Letter"
  else if (char <= 0x7E) "Punctuation"
  else "Control Character" // 0x7F
}
```

# 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 could do to improve this example, but we need to learn more of Core Scala first.

*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
- We use `Ints` for the number of days
- Months have between 1 and 31 days (inclusive)

# 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 (result => 0 < result & result <= 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 (result => 0 < result & result <= 31)
```

# Syntax for Match

```
expr0 match {  
  case Pattern1 => expr1  
  ...  
  case PatternN => exprN  
}
```

# Primitive Value Patterns

A primitive value pattern is either:

- A primitive value
- A free parameter
- The special “don’t care” pattern:

—

*That’s an underscore*

# Matching a Primitive Value With a Pattern

A primitive value  $v$  matches:

- Itself (e.g., 5 matches 5)
- A free parameter (e.g.,  $x$  matches 5)
- The special “don’t care” 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 Pattern1 => expr1  
  ...  
  case PatternN => exprN  
}
```

- Reduce **expr<sub>0</sub>** to a value **v**
- Find the first pattern **k** matching **v** (if it exists) and reduce to **expr<sub>k</sub>** (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 (a `MatchError`)

# 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 (result => 0 < result & result <= 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 (result => 0 < result & result <= 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 ***k***th element of an expression ***e*** with static type  $(\mathbf{T}_1, \dots, \mathbf{T}_N)$  using the syntax:

$e._k$

- The static type of this expression is  $\mathbf{T}_k$
- 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

$(\text{Pattern}_1, \dots, \text{Pattern}_N)$

- 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, and vice versa (bijection)
- Does  $(x, y, z)$  match  $(1, 2)$ ?

# 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 (result => result >= 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  $k$ th 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:

$\text{Array}(\text{Pattern}_1, \dots, \text{Pattern}_N)$

- 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, and vice versa

# 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  $\text{expr}_k$  to a value  $v_k$ , forming the value  $C(v_1, \dots, v_N)$
- If the types of  $\text{expr}_1, \dots, \text{expr}_N$  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): Int = {  
    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

$$C(\text{Pattern}_1, \dots, \text{Pattern}_N)$$

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

# Key Takeaways

- Simple rules to reduce operators with value arguments
- Conditionals can reduce unevaluated expressions
- Template for conditional function on ranges (if/else)
- Template for conditional function on point values (match)
- Compound data: Tuples and Arrays
- Structural data: Case Classes



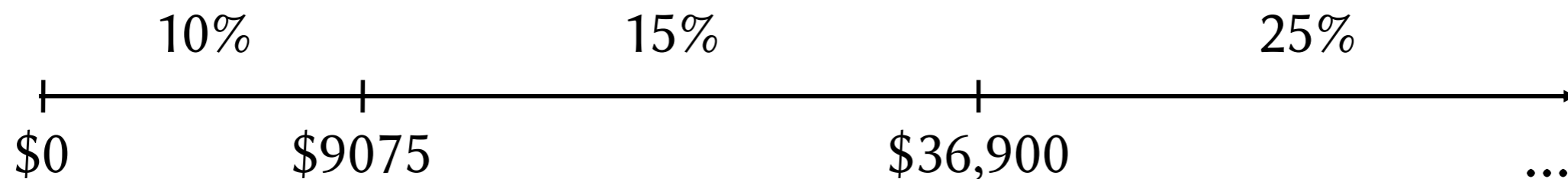
# *Supplemental Sides*

# 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

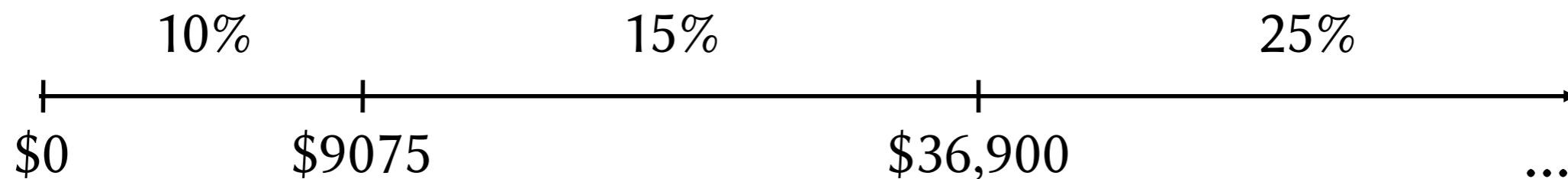
# Graduated Income Tax: Data Analysis and Definition

- We use `Ints` to denote US\$ values and tax percentages
- Both income and tax should be non-negative
- We break the number line into the relevant intervals



# Graduated Income Tax: Data Analysis and Definition

- We use  $\text{Ints}$  to denote US\$ values and tax percentages
- 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): 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)
```