This Scala notebook uses *BeakerX*, a Two Sigma Open Source project that enhances Jupyter.

http://beakerx.com/ (http://beakerx.com/)

In [1]:
```
scala.util.Properties.versionMsg
```

Out[1]:
```
Scala library version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL
```

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.

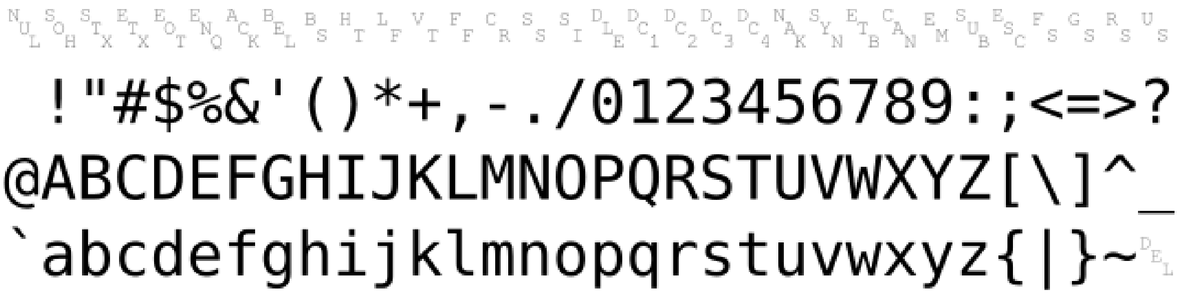
Example 1: Graduated Income Tax (Single Filer)

Bracket (Dollars)	Percentage	Bracket (Dollars)	Percentage
0 to 9,075	10%	186,351 to 405,100	33%
9,075 to 36,900	15%	405,101 to 406,750	35%
36,901 to 89,350	25%	405,751 and up	39.6%
89,351 to 186,350	28%		

We leave this as an exercise for the reader.

Example 2: ASCII Character Classes

We'll use the design recipe to implement a function to describe the different classes of characters included in the 7-bit US-ASCII character set (e.g., control characters, numbers, punctuation).

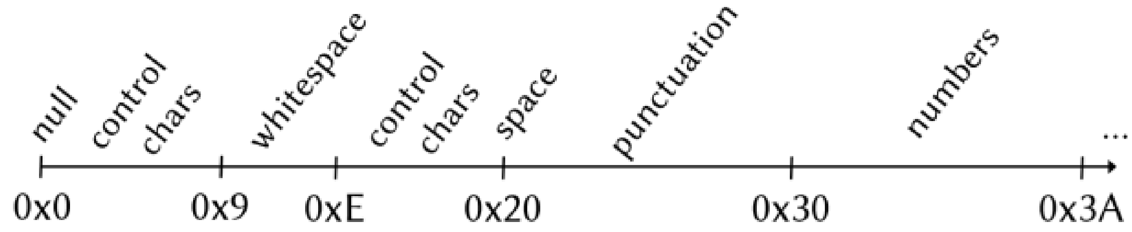

NUL SOH STX ETX EOT ENQ ACK BEL BS HT LF VT FF CR SO SI DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM SUB ESC FS GS RS US
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~DEL

(https://commons.wikimedia.org/wiki/File:ASCII-infobox.svg)

Analysis

- We use `Int` s to ASCII character values.
- We use `String` s 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

```
In [2]: /**
         * 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
```

```
Out[2]: describeAsciiChar: (char: Int)String
```

Tests

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

```
"Null" == describeAsciiChar(0)
"Whitespace" == describeAsciiChar(10)
"Lowercase Letter" == describeAsciiChar(97)
...
```

Definition

The definition of `describeAsciiChar` will be our template for defining conditional functions on ranges.

```
In [3]: /**
         * 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 { ??? }
        }
```

```
Out[3]: describeAsciiChar: (char: Int)String
```

Iterate

In [4]:

```
/**
 * 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
}
```

Out[4]:

```
describeAsciiChar: (char: Int)String
```

Remember, the braces are optional! Leaving them out looks much less cluttered:

In [5]:

```
/**
 * 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
}
```

Out[5]:

```
describeAsciiChar: (char: Int)String
```

In [6]:

```
"Null" == describeAsciiChar(0)
```

Out[6]:

```
true
```

In [7]:

```
"Whitespace" == describeAsciiChar(10)
```

Out[7]:

```
true
```

In [8]:

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

Out[8]:

```
true
```

Notes 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 to do improve these examples, but we need to learn more of Core Scala first.

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?
- Why is it a bad idea to test `Double` s values this way?

Example: Days in a Month

Given the name of a month, we want to return the number of days in that month.

We'll apply the design recipe to implement this function.

Analysis

- We use `String` s to denote months
- We use `Int` s for the number of days
- Months have between 1 and 31 days (inclusive).

Contract

In [9]:

```
/**
 * 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 daysInMonth(monthName: String): Int = {
  ??? : Int
} ensuring (result => 0 < result & result <= 31)
```

Out[9]:

```
daysInMonth: (monthName: String)Int
```

We stated the preconditions in the documentation comment for our function.

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

We'll be able to improve this precondition after learning more Core Scala.

Tests

In [10]:

```
31 == daysInMonth("January")
```

```
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  at daysInMonth(<console>:95)
  ... 46 elided
```

Definition

The definition of `daysInMonth` will be our template for defining conditional functions on ranges.

In [11]:

```
/**
 * 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 daysInMonth(monthName: String): Int = {
  monthName match {
    case "January" => 31
  }
} ensuring (result => 0 < result & result <= 31)
```

Out[11]:

```
daysInMonth: (monthName: String)Int
```

In [12]:

```
31 == daysInMonth("January")
```

Out[12]:

```
true
```

Syntax for match

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

Primitive Value Patterns

A primitive value pattern is one of the following:

- A literal value (e.g., `5` or `"Foo"`)
- A free parameter (e.g., `x` or `apples`)
- The "don't care" pattern, represented with an underscore: `_`

A primitive value `v` matches:

- Itself (e.g., `5` matches `5`)
- A free parameter (e.g., `x` matches `5`)
- The "don't care" pattern (e.g., `_` matches `5`)

The `_` pattern should only be used as the final clause of a match. Why?

Reducing match Expressions

To reduce a match expression:

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

1. Reduce `expr0` to a value `v`.
2. Find the first pattern `K` matching `v`, if it exists.
3. Reduce to `exprK`, replacing all occurrences of `K` with `v` if `K` is a free parameter.
4. If no match exists, the result is ⊥.

In [13]:

```
5 match { case 4 => true }
```

```
scala.MatchError: 5 (of class java.lang.Integer)
  ... 46 elided
```

Static Typing of match

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

expr0: τ
Pattern1: τ
PatternN: τ
expr1: φ
exprN: φ
```

If the pattern is a literal value, it must have type `τ` (i.e., it must match the type of `expr0`). If the pattern is a free parameter, then that free parameter has type `τ` in that `case` 's result expression. The whole `match` expression has result type `φ` , which is also the type of all the `case` s' result expressions.

Finishing the Days in Month Example

In [14]:

```
/**
 * 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 daysInMonth(monthName: String): Int = {
  monthName 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)
```

Out[14]:

```
daysInMonth: (monthName: String)Int
```

In [15]:

```
31 == daysInMonth("January")
```

Out[15]:

```
true
```

```
In [16]: 28 == daysInMonth("February")
```

```
Out[16]: true
```

```
In [17]: 30 == daysInMonth("April")
```

```
Out[17]: true
```

Example of match with Free Parameter

```
In [18]: def plural(word: String): String =
         word match {
           case "deer" => "deer"
           case "fish" => "fish"
           case "mouse" => "mice"
           case x => x + "s"
         }
```

```
Out[18]: plural: (word: String)String
```

```
In [19]: plural("fish")
```

```
Out[19]: fish
```

```
In [20]: plural("cat")
```

```
Out[20]: cats
```

We could also implement this function using `_` :

```
In [21]: def plural(word: String): String =
         word match {
           case "deer" => "deer"
           case "fish" => "fish"
           case "mouse" => "mice"
           case _ => word + "s"
         }
```

```
Out[21]: plural: (word: String)String
```

```
In [22]: plural("dog")
```

```
Out[22]: dogs
```

Compound Datatypes

Please refer to the lecture slides.