Scala Lab Suggestions

Functions

Temperature Converter

The goal is to prompt the user to enter a number, then print the equivalent celsius temperature.

Write a pure function to convert a floating point number that represents fahrenheit into celsius. Implement the formula below to perform the conversion:
c = 5 x (f – 32) / 9.

Create a `main` method to prompt the user for input, convert the text to a floating point value, invoke the conversion, and present the result. Print out the result neatly formatted using the `f"..."` String formatting mechanism.

Modify the conversion method so it has a default value for its argument of 98.4, arrange that the `main` method calls the function with an empty argument list and prints the result. Place this before the existing behavior.

Date manipulation

Write a method to determine if a particular year is a leap year. Leap years are those that are exactly divisible by 4, but not by 100, or are exactly divisible by 400. You can determine if a date is exactly divisible by a number using the modulo operator "%".

Write a method to determine the number of days in a month. Remember that February has 29 days in a leap year.

Write a method that determines if a proposed date (that is, three arguments representing day, month, and year) present a valid date. Give the method a default value for the year parameter.

Test the method, printing the results, for several values, and use named-parameter and default parameter mechanisms to verify that these behave as expected.

Write a method that converts a month number into the textual name.

Write a method that converts a day, month, year proposal into either a nicely formed presentation with the name of the month as text, or "Invalid date" if appropriate.

Zeller’s Congruence

"Zeller's Congruence" is a means of computing the day of the week of a given date (day of month, month of year, and year).

Define a pure function, without mutating any variables, that performs the following computation given the day-of-month, month, and (full, four digit) year in the traditional Gregorian calendar form. The computation is described below:

First, calculate two values `m`, and `y`. If the month is January (1) or February (2) then `m` is the month number plus 12, and `y` is the year minus one. Otherwise, `m` is simply the month number and `y` is simply the year number.

Using these numbers m and y, the following formula uses *integer arithmetic* to calculate the day of the week represented as 0 = Saturday through 6 = Friday. Note that the formula is presented using “math-notation”, not valid Scala, and you will need to fix that!

(day + (13(m + 1) / 5) + y + y/4 - y/100 + y/400) modulo 7

When you have created the function that performs this calculation create a `main` method that prompts the user for day, month, and year and calculate and print out the number representing the day of the week.

Test the program with at least these dates:

Saturday January 1st 2000
Tuesday February 29th 2000
Wednesday March 1st 2000
Thursday March 1st 1900
Monday January 1st 2001

Next, duplicate the invocation of the function and convert the second copy to use the named-parameter form. Change the order of the arguments, and verify that the function produces the same result when called in this way.

Using `match/case`, Enumeration, and Case Classes

Days of the Week

Create two pure functions that convert the day of week values from the previous day of week code into text. Each function should perform the conversion using a different approach:

1. Using a match / case construction
2. Using an `Enumeration` class

Transport Logistics

Define two case classes, one representing a car and the number of passengers that car can carry, the other representing a truck and the payload in pounds that it can carry.

Create a method `allocateFreight`, that takes an argument of type `AnyRef` and a weight of freight, and returns a

`Boolean`. If the input item is neither a car nor a truck, print a message indicating “unsuitable for transport” and return false.

If the input is a truck, and the weight is less than the payload of that truck then return true.

If the input is a car, calculate the payload by taking the passenger count, subtracting 1 (for the driver) and multiplying by 170. If the result is large enough to support the freight, return true.

Exercise the method with cars and trucks of varying capacities, printing the returned value each time.

What happens if you attempt to allocate freight to a text string, instead of either a car or a truck?

Define a trait `Transporter`, arrange that both `Car` and `Truck` extend this, and change the type of the allocateFreight method argument from `AnyRef` to `Transporter`. What happens to the attempt to allocate freight to a text string?

List Processing, Pattern Matching, and Recursion

Write a function that uses recursion and pattern matching to find the length of a list (don’t use the built in methods of `List` for this!)

Modify / ensure the function is tail-recursive

Write a function to find and return the longest string in a list, ensure that it is tail-recursive

Palindrome Checker

Write a program that tests user input to see if it is a palindrome (text that has the same letter sequence read backwards as forwards)

Your program should ignore whitespace, punctuation, and capitalization.

Some palindromes you can use for testing are:

- A Santa dog lived as a devil God at NASA
- Able was I, ere I saw Elba!
- Go deliver a dare, vile dog!
- Racecar
- Some men interpret nine memos

Hints:

A String can be accessed at an explicit (zero-based) index using the charAt method.

A String is a palindrome if the character at position n (zero based) is the same as the character at position length-n-1.

Calendar Printing

Create a program that uses the Zeller's calculation to present a calendar for a given month and year. The format should look like this:

```
Sa Su Mo Tu We Th Fr
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Note that you will need to create functions for `isLeapYear`, and `daysInMonth` (and probably others) in support of this behavior.

By Name Parameters, and Curried Functions

Repetitions

Write a function called `repeat`. This takes two arguments using the curried form. The first is a count value, the second is a `Unit` expression that is *passed by name*.

Use recursion to execute the expression as many times as the count argument specifies.

The following invocation:

```
repeat(2){ println("Hello world") }
```

Should result in the message `Hello world` being printed twice. (Note the use of curly braces instead of parentheses.)

Determine the following:

- What happens if the body of the operation to be repeated refers to a value in the enclosing scope in the caller?
- What happens if the operation argument is surrounded by parentheses instead of curly braces?
    - What if the parentheses wrap behavior that spans multiple lines?
- What happens if repeat is declared to take two arguments in one argument list, rather than in the curried form with two single-argument lists?
    - Can you use curly braces to surround the arguments?
- In the curried form, can you surround the count argument with curly braces?

Give the count argument a default value. How can you invoke the function so it uses the default value?

Logging

Create a function called log that takes three arguments, one is a message (String), passed by name, the second is the priority level of the message, and the third is the priority level of the logging system.

If the message priority is greater or equal to the logging system priority, the print the message.

Invoke the log function with a code block that has String type and that prints a message when it is evaluated. Demonstrate that your log function does not evaluate the code block unless the logging system will use the message it creates.

Do you think this function would benefit from curried arguments? What order makes sense for such arguments?

Objects and Classes

Define an object called `MyDate`, and arrange for it to incorporate the Zeller's congruence, `isLeapYear`, `daysInMonth`, and `calendar` functions that you already created.

(Note, there are 30 days in month numbers 4, 6, 9, and 11. A year is a leap year if it is divisible exactly by 4 but not by 100, or if it's divisible by 400)

Create a function `isValidDate` which accepts day, month, and year, and returns a boolean. Place this in your `MyDate` object.

Next, define a class called `MyDate` that represents day, month, and year of a normal calendar date. Instances of this should be immutable, and provide direct access to the day, month, and year.

Make the constructor `protected`. In the companion object provide a factory behavior that allows a `MyDate` to be prepared using the form `MyDate(<mm>, <dd>, <yyyy>)` -- specifically, avoid the user having to invoke `new`.

> *Note, given the potential for confusion between various orders for the elements of a date (U.S. mm/dd/yyyy, U.K. dd/mm/yyyy and elsewhere commonly yyyy/mm/dd, it's probably a good idea to use the explicitly-named form for passing arguments when invoking methods that take day, month and year, in dates)*

In the class define the methods `toString`, and `addDays`. The addDays method should create a *new* `MyDate` object representing the original date moved forward in time by a number of days (which might wrap to a new month, or year)

Create two dates representing dates in January, such that one is in a leap year, and the other is not. To each, add 31, 59, and 365 days, and print the dates that result from each addition.

Create a subclass called `Holiday`, which also has a companion object with a factory. The `Holiday` class should inherit day, month, and year, but override the parent's `toString` method.

Equality Comparisons

Declare and initialize three values `hello`, `world`, and `helloWorld`. Assign to them the literal values “Hello”, “ world”, and “Hello world” respectively (note the leading space on the second of those strings.)

Print out the values of `helloWorld` and the expression `hello + world` (both should appear identical).

Print out the value of the comparisons:

- `helloWorld == hello + world`
- `helloWorld equals (hello + world)`
- `helloWorld eq (hello + world)`
- `helloWorld eq (hello + world).intern()`

In the MyDate class, define a `final equals` method, and verify that a `MyDate` and a `Holiday` representing the same day, month, and year compare as equal using the == comparison.

Optionally, print a message in the `equals` method so you can see that it's invoked in response to the == comparison.

Higher Order, and Generic, Functions

Writing Higher Order Functions

Write a function that accepts a `List[String]` and returns a list that contains only the strings that are longer than a particular length threshold. Parameterize the length threshold.

Modify the function so it returns a list containing only strings that begin with a particular character, parameterize the character. Pay attention to which parts of the function changed, and which remained the same.

Modify the function further so that it accepts as its second parameter a `String => Boolean` function, and returns a list that contains only the strings that pass the test implied by the function argument.

Generalize the function further so that it accepts a `List[T]` and appropriate test function, and returns an appropriate return type.

> *Note: This is likely to cause some problems with type inferencing. Two solutions are possible, either apply a type to the invocation of the filtering method, or curry the arguments.*

Concordance

The goal of this exercise is the create a table of the most frequently occuring words in a text document, so that the table has the word, followed by the number of times that the word occurs in the document. The table should list the 200 most frequently occurring words, in descending order of frequency. A sample of the first few lines of output might look like this:

```
the :  4507
 to :  4242
 of :  3729
and :  3658
her :  2203
```

Go to the website `gutenberg.org` and search for books by the author Jane Austen. Select the book Pride and Prejudice, and from the list of formats, select Plain Text UTF-8. When the text shows, use your browser to do a “Save As” operation, and place the file in the root directory of your lab exercises project. This directory will be the “current working directory” when your Scala project starts up, so you should be able to open the file using only the filename, and without any path specification.

Hints:

- You will need to read the file into a monad structure, The class `scala.io.Source` can do this. The method of interest treats one line of text as a single element.

- You will need to turn one line into many words. This can be done with a regular expression via the class `java.util.regex.Pattern`. A workable regular expression is `\W+` which represents “a sequence of one or more non-word characters”.
- Scala allows a regular expression to be created directly by appending “r” to the string literal, like this:
  `val regex = “\\W+”r`
- The regular expression given will leave some empty strings in its output, these should not be part of the count.
- Ensure that you ignore capitalization, i.e. count “The” and “the” as the same word.
- The lines of the file are presented from the `Source` in an `Iterable` object, but building the map is most easily achieved using the method `groupBy`, which exists in the `Seq` trait but not in an `Iterable`. A `Seq` can be extracted directly from the `Iterable` with the method `toSeq`.

Folding

Use a fold operation to concatenate a list of Strings. Repeat this with fold left and fold right. Is there any difference?

Use a fold operation to find the longest String in a list. Which fold operations can work? Modify your goal to find the longest string, and its position in the list. Does this change which operations can be used?

Reimplement the calendar printing exercise using fold operations and avoiding explicit recursion.

Imagine a series of cars on a single lane road (so no passing is possible). There’s a tendency for these cars to bunch up into groups, where a slower vehicle leads a group of other cars that would like to go faster than the lead, but cannot pass. Write a function that takes a list of car speeds and returns a list of groups that form. For example, given this input:

```
List(67, 72, 68, 61, 98, 66, 67, 55, 62)
     ==========  ==============  ======
```

Three groups will form (traveling from right to left) and the output should be:

```
List(
  List(67, 72, 68),
  List(61, 98, 66, 67),
  List(55, 62))
```

Exception Handling

Rejecting bad arguments

Ensure that your `MyDate` object throws an `IllegalArgumentException` if provided with day, month, and year combinations that do not form a valid date.

Write a new object with a main method that creates four `MyDate` objects, two with bad values, and two with good. Immediately print the value of the newly created object, e.g.:

```
println(MyDate(1, 1, 2000))
```

Try to run the program, and note how it fails. What is printed, and what is, and is not, executed?

Wrap each call to create MyDate objects in a try/catch construction. Arrange that a message is printed for each failure. Run the code again and verify that the code attempts to create all four MyDate objects, and failures no longer crash the program.

Create a new object and `main` method. Attempt to create the four `MyDate` objects, but this time, wrap the attempt in a `Try` object. Write a utility method called `show`. The method takes a `Try[MyDate]` as argument and prints a message. If the argument is a `Success` object, print the date, if it's a `Failure`, print a message indicating the nature of the problem. Use this to show the results of all four attempts to create `MyDate` objects.

Create a wrapper function for the factory for `MyDate` so that instead of returning a `MyDate` directly, it represents its outcome using an `Either[Throwable, MyDate]`. Set the left value to an `IllegalArgumentException` if the values given for day, month, and year cause the main factory to throw an exception. Set the right value if a MyDate is returned.

Create a list of tuples containing day, month, and year values. Process the list using a `map` operation to convert the tuples using the `MyDate` factory. Print messages for the bad dates, and produce a list of only valid `MyDate` objects that have been extracted from their `Either` wrappers.

Try / Catch Handling

Create a program that reads a filename from StdIn and attempts to open a file of that name. Provide a try/catch block and if the named file cannot be opened print a message re-prompt the user and retry. If the file is opened, print the contents and quit.

Higher Order Function Adapters

Converting Exceptions to Eithers

Modify the concordance example so that it starts with a list of filenames and attempts to open all of them in turn, adding all the words to the frequency table.

Demonstrate that if a named file is not found, the program crashes.

Create a wrapper function that converts a function that returns X and throws an exception in the event of failure into a function that returns an `Either[Throwable, X]`

Use the wrapper in the program so that failure to open a file causes a message to be printed, and the file to be skipped, but the processing to continue with the next file.