Calculator: Instructions

Help Center

Attention: You are allowed to submit **a maximum of 5 times**! for grade purposes. Once you have submitted your solution, you should see your grade and a feedback about your code on the Coursera website within 20 minutes. If you want to improve your grade, just submit an improved solution. The best of all your first 5 submissions will count as the final grade. You can still submit after the 5th time to get feedback on your improved solutions, however, these are for research purposes only, and will not be counted towards your final grade.

To start, first download the assignment: calculator.zip.

Note to IntelliJ users: IntelliJ seems to have trouble importing this assignment. Please use the workaround in this comment.

In this assignment, you will use Function Reactive Programming (FRP), with the `Signal[A]` class that you have seen in the lectures, to implement a spreadsheet-like calculator. In this calculator, cells can depend on the value of other cells, and are recomputed automatically when the latter change.

The User Interface (UI) for the programs in this assignment are Web-based, i.e., they run in an HTML page in your Web browser. To compile your code to runnable JavaScript, we use Scala.js, the Scala to JavaScript compiler. We have all set it up for you in the assignment template, but this will change the way you run the program.

Tweet length monitoring

As you probably know, Tweets (messages on Twitter) are limited to 140 characters. When typing a Tweet, it is very helpful to receive instantaneous visual feedback on the number of characters you have left.

The traditional way of implementing this visual feedback is to set up an `onchange` event handler (a callback) on the text area. In the event handler, we imperatively ask for the text, compute the length, then write back the computed remaining characters.

In FRP, we use `Signal`s to abstract away the mutable nature of the UI (both inputs and outputs) while working on the logic that binds output to input, which is functional.

From Tweet text to remaining char count

Your first task is to implement the function `tweetRemainingCharsCount` in `TweetLength.scala`. This function takes a `Signal[String]` of the text of a Tweet being typed, and returns a `Signal[Int]` with the corresponding number of characters that are left.

Note that the Tweet length is *not* `text.length`, as this would not properly handle supplementary characters. Use the provided function `tweetLength` to correctly compute the length of a given text.

(Actually, even this is a simplification from reality. The complete specification, which we do not implement in this assignment, can be found here.)

Note that the remaining number of characters could very well be negative, if the Tweet text currently contains more than `MaxTweetLength` characters.

Running the application so far

Now that you have implemented the first function, you can start running the Web UI. To do so, you need to compile the UI to JavaScript from sbt (you cannot do this from your IDE).

In sbt, run:

```
> webUI/fastOptJS
```

If your code compiles, this will produce the JavaScript necessary to run the HTML page in the browser. You can now open the file `web-ui/index.html` in your favorite browser, and enter text in the first text area.

If you implemented `tweetRemainingCharsCount` correctly, the remaining number of characters should automatically update.

From remaining char count to "warning" color

For better visual feedback, we also want to display the remaining character count in colors indicating how "safe" we are:

- If there are 15 or more characters left, the color `"green"`
- If there are between 0 and 14 characters left, included, the color `"orange"`
- Otherwise (if the remaining count is negative), the color `"red"`

(these are HTML colors).

Implement the function `colorForRemainingCharsCount`, which uses the signal of remaining char count to compute the signal of color.

You can now re-run `webUI/fastOptJS` and refresh your browser. You should see the number of remaining characters changing color accordingly.

Root solver for 2nd degree polynomial

The second part of the assignment is similar, for a different application: find the real root(s) of a 2nd degree polynomial of the form $ax^2 + bx + c$, where a is a non-zero value.

We explicitly ask for the intermediary *discriminant*, which we call Δ:

$$\Delta = b^2 - 4ac$$

which you should compute in the function `computeDelta`, from the signals for the coefficients a, b and c. Note that in this case, your output signal depends on several input signals.

Then, use Δ to compute the set of roots of the polynomial in `computeSolutions`. Recall that there can be 0 (when Δ is negative), 1 or 2 roots to such a polynomial, and that can be computed with the formula:

$$(-b \pm \sqrt{\Delta}) / (2a)$$

After compiling with `webUI/fastOptJS` and refreshing your brower, you can play with the root solver.

Spreadsheet-like calculator

Now that you are warmed up manipulating `Signal`s, it is time to proceed with the original goal of this assignment: the spreadsheet-like calculator.

To simplify things a bit, we use a *list* of named variables, rather than a 2-dimensional table of cells. In the Web UI, there is a fixed list of 10 variables, but your code should be able to handle an arbitrary number of variables.

The main function is the following:

```
def computeValues(namedExpressions: Map[String, Signal[Expr]])
    : Map[String, Signal[Double]]
```

This function takes as input a map from variable name to expressions of their values. Since the expression is derived from the text entered by the user, it is a `Signal`. The `Expr` abstract data type is defined as follows:

```
sealed abstract class Expr
final case class Literal(v: Double) extends Expr
final case class Ref(name: String) extends Expr
final case class Plus(a: Expr, b: Expr) extends Expr
final case class Minus(a: Expr, b: Expr) extends Expr
final case class Times(a: Expr, b: Expr) extends Expr
final case class Divide(a: Expr, b: Expr) extends Expr
```

The `Ref(name)` case class represents a reference to another variable in the map `namedExpressions`. The other kinds of expressions have the obvious meaning.

The function should return another map from the same set of variable names to their actual values, computed from their expressions.

Implement the function `computeValues`, and its helper `eval`, while keeping the following things in mind:

- `Ref`s to other variables could cause cyclic dependencies (e.g., `a = b + 1` and `b = 2 * a`. Such cyclic dependencies are considered as errors (failing to detect this will cause infinite loops).
- `Ref`erencing a variable that is not in the map is an error.

Such errors should be handled by returning `Double.NaN`, the Not-a-Number value.

It is not asked that, when given an expression for `a = b + 1`, you compute the resulting value signal for `a` in terms of the value signal for `b`. It is OK to compute it from the *expression* signal for `b`.

Once all of this is done, you can, once again, compile to JavaScript with `webUI/fastOptJS` and refresh your browser. You can now play with the simplified spreadsheet. Observe that, when you type in some expression, the values recomputed as a result are highlighted for 1.5s. If all ten values are highlighted every time you modify something, then something is wrong with the way you construct your signals.

Notice that, as you modify dependencies between variables in the expressions, the dependencies between signals adapt dynamically.