Final project - Calculator

Presentation

The goal of the final project is to apply the knowledge acquired during this introductory Scala course, as well as to do some additional individual research on some Scala libraries.

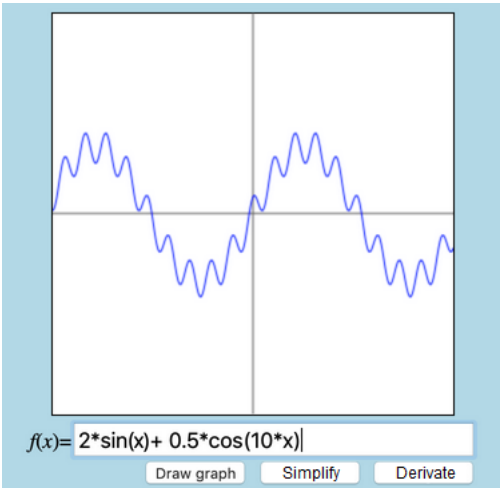
The project aims to create a program that would be able to read Strings representing mathematical functions of a single variable **x** and convert them to an internal tree-like data representation based on a set of rules. The representation will be used to create a Scala function that applies to some input value for the variable **x**. As it will be able compute a value based on different values of **x**, it will be used by a Graphical User Interface (GUI) that will display a set of points **(x, f(x))** that will be connected with lines to make the whole picture look like a plot that will represent the mathematical function.

NB ! Using something similar to Python's `eval` (some existing function that evaluates Scala code) is prohibited ! The whole point of the project is for you to make your own evaluator.

The project is inspired by a similar assignment from a Haskell functional programming course given at Chalmers University of Technology :

http://www.cse.chalmers.se/edu/year/2018/course/TDA452_Functional_Programming/labs/4/graph.shtml [1]

Here is what the **structure** of the GUI will look like :



Adapted from [1]

You don't need to waste time replicating it perfectly, the only important things are the graphic, the text field for the formula and the buttons. You can modify the layout if you wish so.

The user will :

1. Write the expression of the function f in a text field
2. Click on "Draw graph"

After the click, a graph of the function will be displayed if its expression is valid (based on the rules defined by the user).

You can add an additional text field (or label) that would display "OK" if the String's conversion to a data representation succeeded, and "NOK" (or some error message) if it failed.

Organization

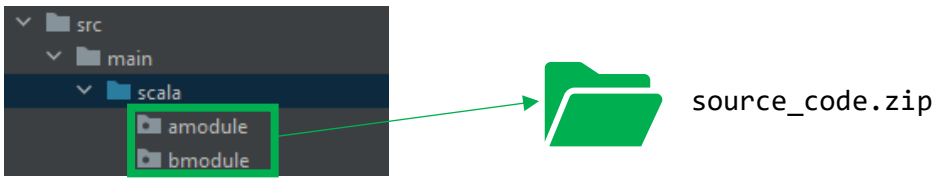
The project will be done in pairs, and only one group of 3 will be accepted per class.

Submission

The project will be submitted via Teams by one of the pair's members.

The submission will consist of :

- The source code in a zip file, i.e., the contents of the project's `src/main/scala` folder (note that dummy names were used for the modules here) :



- The JAR file packaged via **SBT** ;
- A short video (3 min. max, you can record your screen via some software such as OBS Studio) showing how you use the GUI with different formulas.

The submitted source code should be clean and its readability will impact your grade.

Here are some guidelines :

- Do not write long lines of code (a max. length of ~80 should be enough)
- Keep a consistent layout for your code
- Comment your code with meaningful explanations
- Clean out the junk code (unused, obsolete comments and/or commented code)
- Avoid code repetition as much as possible (copy-pasting code)

Specification

Part I

Part I will make heavy use of case classes/objects, sealed traits and pattern matching. You will also need to use parser combinators that you will have to research within you project group, but some links and guidance will be provided.

In this part, the goal is to design a data type for modelling univariate functions, i.e., functions that take a single input variable **x** and allow you to compute **f(x)**.

Examples of valid functions are :

- 3*(7+1)
- 3*x+17.3
- sin(x)+cos(x)
- sin(2*x+3.2)+3*x+5.7

For this project, an expression will consist of :

- **Numbers** : positive integers or floating-point numbers (you can represent all numbers as doubles)
- **Variables** : only one variable is possible, x
- **Operators** : + and * will be enough
- **Functions** : sin and cos will be enough

Aside from the data type, you will also define some useful functions over it.

1. Design a recursive data type **`Expr`** that represents expressions of the above kind.
   Hint : use a sealed trait and case classes/object to design each data constructor. Numbers and variables will be trivial, while operators and functions may be a little bit more complex. *You can get inspired by these slides from the previously mentioned Chalmers course (note that the code is in Haskell and everything does not apply, but the ideas are similar) :*

   http://www.cse.chalmers.se/edu/year/2014/course/TDA452_Functional_Programming/FPLectures/04B.pdf

2. Define a function of the following signature :

   ```
   showExpr(e: Expr): String
   ```

   that converts any **`Expr`** to **`String`**. The strings that are produced should look something like the example expressions (« valid functions ») shown earlier. Use as little parentheses as possible but consider that we want to be able to read an expression back without loss of information. Hint : you may eventually need to define helper methods for **`showExpr`** to avoid showing unnecessary parentheses.

   It is not required to show floating point numbers that represent integer numbers without the decimal part. For example, you may choose to always show 2.0 as "2.0" and not as "2". (But you are allowed to do this.)

3. Define a function of the following signature :

```
eval(e: Expr)(x: Double): Double
```

that, given an expression, and the value for the variable x, calculates the value of the expression.

4. Using the Scala Standard Parser Combinator Library (https://github.com/scala/scala-parser-combinators), define a parser that would be able to transform a **`String`** of a mathematical function to an **`Expr`** representing the said function.

To get some examples for the parser combinator library, use the following links :

https://github.com/scala/scala-parser-combinators/blob/main/docs/Getting_Started.md

https://enear.github.io/2016/03/31/parser-combinators/

Hint 0 : you will need to define a set of rules which will allow you to parse a String to get its corresponding **`Expr`** representation.

Hint 1 : using the **`RegexParsers`** class should be enough to define the rules that allow interpreting the mathematical expressions allowed in this project.

Hint 2 : you can then use the previously implemented **`showExpr`** method to transform the **`Expr`** back to **`String`** and see if there was no information loss. You can also just print the obtained **`Expr`** using the case classes' **`toString`** method.

Hint 4 : this is probably the trickiest part of this project. If you encounter difficulties, do not hesitate to ask the instructor for help.

5. Define a function of the following signature :

```
simplify(e: Expr): Expr
```

which simplifies expressions so that subexpressions not involving variables are always simplified to their smallest representation, and that (sub)expressions representing **x + 0** , **0 * x** and **1 * x** and similar terms are always simplified. Simplify as much as you expect (or as much as possible).

6. Bonus : define a function

```
differentiate(e: Expr): Expr
```

which differentiates the expression (with respect to x). You should use the simplify function to simplify the result. *Again, you can get inspired by these slides from the previously mentioned Chalmers course:*

http://www.cse.chalmers.se/edu/year/2014/course/TDA452_Functional_Programming/FPLectures/04B.pdf

Part II

The goal of **part II** is to create a graphical user interface (GUI) that would display a graph of the parsed function.

You can check the following guide to get a feeling of the possibilities offered by the Swing library in Scala : just create a new project and paste the code you find on this page. Do not forget to add the Swing to the **build.sbt** file.

https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20Scala%20GUI.html

Based on the elements present in this guide's example, you will need to add/remove (un)necessary elements to your window.

Hint : For the drawing part, you should read about how coordinates are represented in the canvas, and how you can make a correspondence between your (**x, f(x)**) coordinates and pixel coordinates in the canvas.

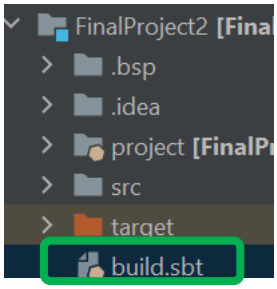
The GUI should work as follows :

- To start the GUI, run the project's JAR using the following command in a Terminal/cmd.exe :

```
scala path/to/the/projects/jar
```

- At the beginning, the graph's canvas ("rectangle" it is drawn in), as well as the text field where the expression should be written to, is empty.
- Clicking "Draw graph" should parse the expression. If successful, it should display the new graph, otherwise it should just clean the canvas.
- Clicking "Simplify" should simplify the expression and change the text's value with the new expression in the text field. It can also update the graph : normally, there should be no change.
- Bonus : clicking "Derivate" should derivate the expression and update the graph.

Help with project configuration



Adding 3rd party libraries as dependencies to your SBT projects

To add external libraries to your SBT project, you need to edit the **`build.sbt`** file found at your project's root. You will need to add the following dependencies for your project :

```
libraryDependencies += "org.scala-lang.modules" %% "scala-parser-combinators" %
"2.1.1"
libraryDependencies += "org.scala-lang.modules" %% "scala-swing" % "3.0.0"
```

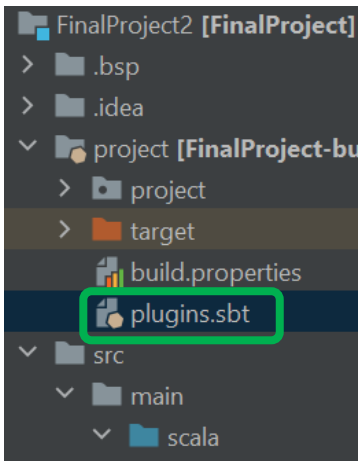
Creating a Fat JAR using SBT

Once you have your code base, instead of using the usual `run` command in your SBT shell, you will need to package your code in a JAR file to execute it from a command line (a Terminal or cmd.exe) with the command **`scala path/to/your/JAR`**.

The packaging is already done by the `run` command, which itself executes other steps, one of which is `package`. However, the JAR generated this way does not include the 3rd party libraries that you have added as dependencies to your project. Consequently, when running your JAR you may encounter errors due to the missing dependencies during runtime.

One solution to this problem is to also include the dependencies inside the generated JAR : such a JAR file will be called a "fat JAR" or an "uber-JAR". There are different ways to create a fat JAR, e.g., via SBT or Maven plugins.

During this course, we used SBT as our build tool for Scala. To create a fat JAR with SBT, you will need to add the `sbt-assembly` plugin to your SBT project. You can do that by creating a **`plugins.sbt`** file inside the project folder of your sbt project :



then adding the following line to the **file** :

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "1.2.0")
```

You can now use the command **`assembly`** instead of **`run`** in IntelliJ's **sbt shell**. Using the assembly command will generate a Fat JAR inside the **`target/scala-2.12`** folder of your project.