



Assignment 4

Hand in this assignment until **Tuesday, June 04, 12pm** at the latest.

🤔 Running out of ideas?

Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the [#forum](#) channel on our Discord server—maybe you'll find just the help you need.

⚠️ The lecture evaluation is coming up!

We rely on your feedback to steer Functional Programming in the right direction. We look forward to both your criticism and your praise! So please take part in the lecture evaluation **by June 10**. Thank you very much!

📖 Exam-style Exercises

Exercises marked with 📖 are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Task 1: Regular Expressions (Algebraic Data Types)

Finite state machines aren't the only method to implement regular expression matching. Here, we will build a regular expression matcher using the *derivatives of a regular expression*.

To implement this approach, we first need a representation for regular expressions on a given alphabet of symbols.

A 📖 Complete the definition of an algebraic data type (ADT) for regular expressions:

```
1 data RegExp a = RxNone    --  $\emptyset$ 
2   | RxEpsilon    --  $\epsilon$ 
3   | ...
```

The ADT needs constructors for the following cases:

- the empty sequence of symbols ϵ ,
- a symbol of the alphabet (symbols have type a – typically $a \equiv \text{Char}$),
- a concatenation $r_1 r_2$ of two regular expressions r_1 and r_2 (r_1 followed by r_2),
- the *Kleene star* r^* of a regular expression r (r repeated zero or more times),
- an alternation $r_1 | r_2$ of two regular expressions r_1 and r_2 (r_1 or r_2),
- a special regular expression \emptyset which accepts no input at all.

B 📖 Write an instance of type class `Show` for `RegExp a`. Use parentheses `()` to group parts of the regular expression, if necessary.


Example: The output of `show` for regular expression $(x|(yz))^*$ looks like this: `"(x|(yz))^*"`.

The *derivative of a regular expression* r with respect to symbol a is another regular expression r' . If input s is accepted by r , then r' accepts s with the starting symbol a removed. For example, consider the regular expression $r = ab^*$. The derivative of r with respect to a is b^* and the derivative of b^* with respect to b is again b^* . However, the derivative of r with respect to b is the regular expression \emptyset .

These derivatives can be used to implement regular expression matching.

First we need a function $\nu(r)$ to test whether a regular expression r is nullable. We say that r is nullable, if r accepts the empty sequence ϵ :

$$\begin{aligned}
\nu(\varepsilon) &= \text{True} \\
\nu(a) &= \text{False} \\
\nu(r^*) &= \text{True} \\
\nu(r_1 r_2) &= \nu(r_1) \wedge \nu(r_2) \\
\nu(r_1 | r_2) &= \nu(r_1) \vee \nu(r_2) \\
\nu(\emptyset) &= \text{False}
\end{aligned}$$

- C**  Write a function `nullable :: RegExp a -> Bool` implementing ν . Now we can define a function $\partial_{c(r)}$ to compute the derivative of a regular expression r with respect to a symbol c :

$$\begin{aligned}
\partial_c(\varepsilon) &= \emptyset \\
\partial_c(a) &= \begin{cases} \varepsilon & , \text{ if } a = c \\ \emptyset & , \text{ if } a \neq c \end{cases} \\
\partial_c(r^*) &= \partial_c(r)r^* \\
\partial_c(r_1 r_2) &= \begin{cases} \partial_c(r_1)r_2 \mid \partial_c(r_2) & , \text{ if } \nu(r_1) \\ \partial_c(r_1)r_2 & , \text{ if } \neg\nu(r_1) \end{cases} \\
\partial_c(r_1 | r_2) &= \partial_c(r_1) \mid \partial_c(r_2) \\
\partial_c(\emptyset) &= \emptyset
\end{aligned}$$

- D** Write a function `derive :: Eq a => RegExp a -> a -> RegExp a` implementing ∂ .

Suppose we have a regular expression r and a string of symbols $s = c_1 \dots c_n$. To test whether r accepts s , we can make use of a successive application of ∂ to r with respect to the symbols of s . If and only if the final derivative with respect to c_n is nullable, *i.e.*, matches the empty string ε , the regular expression r matches the whole string s :

$$r \text{ matches } s \Leftrightarrow \nu(\partial_{c_n}(\dots \partial_{c_1}(r)))$$

- E** Write a function `match :: Eq a => RegExp a -> [a] -> Bool` implementing the regular expression matcher. Remember to provide some tests, *i.e.* sample regular expressions and sample symbol sequences.
- F** Extend your definitions of `RegExp a`, `nullable` and `derive` to also support the *Kleene plus*: A regular expression can also be r^+ (the regular expression r repeated one or more times).