CSCC24 2025 Summer – Assignment 1
Due: June 10, 2025, 11:59 PM
This assignment is worth 10% of the course grade.

In this assignment, you will work in Haskell with algebraic data types and implement interesting recursive algorithms.

As usual, you should also aim for reasonably efficient algorithms and reasonably organized, comprehensible code.

Correctness (mostly auto-testing) is worth 90% of the marks; code quality is worth 10%.

Regular Expressions

Like in CSCB36, regular expressions in this assignment are inductively defined by:

- ϵ (Greek letter epsilon) is a regular expression.
- A single character is a regular expression.
- (Concatenation) If r_1 and r_2 are regular expressions, then $r_1 r_2$ is a regular expression.
- ("Or") If r_1 and r_2 are regular expressions, then $r_1 \mid r_2$ is a regular expression.
- (Kleene star) If r is a regular expression, then r^* is a regular expression.

This is represented by the following algebraic data type:

```
data R = Eps | Single Char | Cat R R | Or R R | Star R
    deriving (Eq, Show)
```

Question 1 (10 marks)

This question is about rendering the `R` representation as a string in the usual mathematical notation. Example:

```
Or (Cat (Single '0') (Star (Cat (Single '1') (Single '0'))))
   (Cat (Single '1') Eps)
```

is rendered as `"0(10)*|1e"`. Parenthesizing is according to this operator precedence rule: `*` is at precedence level 3 (the highest), concatenation is at level 2, `|` is at level 1 (the lowest).

The function I will be testing is simply

```
render :: R -> String
```

but that alone won't help you. Read on:

Adding parentheses where they're needed (and equally importantly, not adding where not needed) will be the most tricky part of this question. Here is a very evidently tedious way:

```
render (Star r) = further split r into 5 cases so you can:
    if r is a Cat, add parentheses around render r
    if r is a Single, don't add parentheses around render r
    3 other cases
render (Cat r1 r2) = further split r1 into 5 cases, r2 into 5 cases:
    if r1 is an Or, add parentheses around render r1
    if r1 is a Star, don't add parentheses around render r1
    ad nauseum
render (Or r1 r2) = you get the point
```

I want you to think of a much less repetitive solution. The hint is this:

In the repetitive approach, the parent has the burden of putting parentheses around the child. This causes part of the tedium—a parent can have multiple children.

So how about the opposite: Each case has the burden of putting parentheses around itself. The parent just has to pass some information about itself to the child, then the child can decide for itself.

This manifests as setting up your own helper function and doing the real recursion there, e.g.,

```
renderHelper :: InfoFromParent -> R -> String

renderHelper ifp (Cat r1 r2) =
   Based on ifp and the fact that I am a Cat,
     I know whether myself need parentheses.
   My recursive calls to r1 and r2 are
       renderHelper ??? r1
       renderHelper ??? r2
   ??? is my turn giving info to children
```

It is also your job to choose a good type and good values for `InfoFromParent`. Here is a bad choice: `R` again (too much irrelevant information).

Question 2 (10 marks)

This question is about implementing an algorithm for checking whether a string is in the language given by a regular expression. It will not be the most efficient algorithm, but not the worst either, and it is a fairly straightforward structural recursion.

The function you will implement has this surprising type:

```
consume :: R -> String -> [String]
```

The answer is not a simple boolean, but a list of strings. Here is the explanation. Given a regular expression r and a string s, there may be multiple ways of splitting s such that the prefixes match r; then `consume r s` should give the list of the suffixes. If no prefix matches, then the answer is the empty list. Examples:

- `consume Eps "0101" = ["0101"]`

  Reason: The empty prefix matches ϵ, so the suffix is the whole string. This is the only way.

- `consume (Single '0') "0011" = ["011"]`
- `consume (Single '1') "0011" = []`
- `consume (Star (Single '0')) "0011" = ["0011", "011", "11"]`

  Reason: 0011 has 3 prefixes that match 0^*: the empty prefix, the prefix 0, and the prefix 00. The answer list has the respective suffixes.
- `consume (Or (Star (Single '0')) (Single '0')) "0011" = ["0011", "011", "11", "011"]`

  Reason: Both branches of `Or` contribute to the answer.

  It is OK for `"011"` to occur twice. In general, it is up to you whether to weed out this kind of duplicates. For simplicity, perhaps don't bother.
- `consume (Cat (Star (Single '0')) (Single '0')) "0011" = ["011", "11"]`

  Strategy: Use a recursive call to get

  ```
  consume (Star (Single '0')) "0011" = ["0011", "011", "11"]
  ```

  Each leftover is a good candidate for `consume (Single '0')`:

  ```
  consume (Single '0') "0011" = ["011"]
  consume (Single '0') "011" = ["11"]
  consume (Single '0') "11" = []
  ```

  Merge the three outputs for the overall answer. So this is a benefit of computing the list of all possible leftovers—it helps with the `Cat` case.

  In general, this will be your strategy for handling the `Cat` case. It is a little bit better than brute-forcing: There are 5 ways to split 0011, and this strategy tries only 3 of them.

The examples have shown one particular order used in the answer list. This order is not required; it is OK if your code outputs some other order.

It follows that s is in the language of r iff the empty string is an element of `consume r s`:

```
isInRL :: R -> String -> Bool
isInRL r s = "" `elem` consume r xs
```

Question 3 (5 marks)

OOP advocates talk excitedly about how OOP is extensible: Adding a feature is non-intrusive, i.e., requiring no change to existing declarations and definitions, just adding new ones. Let's calmly examine that.

Please hand in your answers to the following parts in comparison.txt.

a. In OOP, what is the typical approach to solving the two questions above? You just need to describe the organization: List the superclasses/interfaces, subclasses, and methods. You don't need to code up the methods.

b. If we extend by adding one more case to regular expressions (say a wildcard that matches any single character): For each of the Haskell version and the OOP version, name the existing declarations and definitions (types, functions; classes, methods) that need change, and name what to add. Which version makes it less intrusive?

c. If we extend by adding one more operation (say, `match` such that `match r s` gives all prefixes of s that match r): Again, for each version, name which existing declarations and definitions need change, and what to add. Which version makes it less intrusive this time?

(The tension is known as the "expression problem".)

End of questions.