Unlike the last few labs, our primary focus in the lab is not new language features. Instead, we will explore some related topics that we have bypassed in prior labs, namely parsing. And more importantly, since it is the last lab of semester, we want to play with the cool interpreters that we have built.

Concretely, we will consider regular expressions. We write construct a parser for a language of regular expressions and implement a regular expression matcher in Scala. We extend our Lab 5 interpreters with regular expression literals and regular expression matching (like JavaScript) using your parser and expression matcher.

The skills in this lab are as follows: constructing a recursive descent parser and programming with continuations.

**Instructions.** You will work on this assignment in pairs. However, note that each student needs to submit a write-up and are individually responsible for completing the assignment.

You are welcome to talk about these questions in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Recall the evaluation guideline from the course syllabus. Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expression computation currently unfamiliar to you. Finally, make sure that your file compiles and runs (using Scala 2.9.2). A program that does not compile will not be graded.

**Submission Instructions.** First, submit your .scala file to the auto-testing system to get instant feedback on your code. You may submit as many times as you want, but please submit at least once before the deadline. We will only look at your last submission. A link to the auto-testing system is available on the moodle.

Then, upload to the moodle exactly two files named as follows:

- `Lab6_YourIdentiKey.pdf` with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)
- `Lab6_YourIdentiKey.scala` with your answers to the coding exercises

Replace `YourIdentiKey` with your IdentiKey. To help with managing the submissions, we ask that you rename your uploaded files in this manner.
Getting Started. Download the code template Lab6_YourIdentiKey.scala from the assignment page. Be sure to also write your name, your partner, and collaborators there.

1. Feedback. Complete the survey on the linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.

2. Regular Expressions. Consider the following syntax for a language of regular expressions. We note the corresponding Scala case class or case object used to construct abstract syntax trees of type RegExpr (shown below the grammar in full).

```scala
sealed abstract class RegExpr
  case object RNoString extends RegExpr
  case object REmptyString extends RegExpr
  case class RSingle(c: Char) extends RegExpr
  case class RConcat(re1: RegExpr, re2: RegExpr) extends RegExpr
  case class RUnion(re1: RegExpr, re2: RegExpr) extends RegExpr
  case class RStar(re1: RegExpr) extends RegExpr
  case object RAnyChar extends RegExpr
  case class RPlus(re1: RegExpr) extends RegExpr
  case class ROption(re1: RegExpr) extends RegExpr
  case class RIntersect(re1: RegExpr, re2: RegExpr) extends RegExpr
  case class RNeg(re1: RegExpr) extends RegExpr
```

A regular expression defines a regular language (language = a set of strings). The first six constructors are the basic regular expression constants and operators. Let us write \( \mathcal{L}(re) \) for the language specified by the regular expression \( re \):

- \( \mathcal{L}(!) \) \( \overset{def}{=} \) \( \emptyset \), that is, the empty set.
- \( \mathcal{L}(\#) \) \( \overset{def}{=} \) \{""\}, that is, the set with the empty string.
- \( \mathcal{L}(c) \) \( \overset{def}{=} \) \{"c"\}, that is, the set with the string matching the single character \( c \).
- **Concatenation.** A string \( s = s_1s_2 \in \mathcal{L}(re_1re_2) \) iff \( s_1 \in \mathcal{L}(re_1) \) and \( s_2 \in \mathcal{L}(re_2) \).
- **Union.** A string \( s \in \mathcal{L}(re_1|re_2) \) iff \( s \in \mathcal{L}(re_1) \) or \( s \in \mathcal{L}(re_2) \) (i.e., \( s \in \mathcal{L}(re_1) \cup \mathcal{L}(re_2) \)).
- **Kleene Star.** A string \( s \in \mathcal{L}(re_1*) \) iff \( s \) is in zero-or-more concatenations of \( re \).
The basic mathematical definition of regular expressions given above are often extended with more operators in programming languages as a convenience for developers. We consider five extended operators:

- **Any Character.** A string \( s \in \mathcal{L}(.) \) iff \( s \) consists of any single character.
- **One-Or-More.** A string \( s \in \mathcal{L}(re_1^*) \) iff \( s \) is in one-or-more concatenations of \( re \) (i.e., \( s \in \mathcal{L}(re_1 re_1^*) \)).
- **Zero-Or-More.** A string \( s \in \mathcal{L}(re_1?) \) iff \( s \) is zero-or-one matches of \( re \) (i.e., \( s \in \mathcal{L}(#|re_1) \)).
- **Intersection.** A string \( s \in \mathcal{L}(re_1 & re_2) \) iff \( s \in \mathcal{L}(re_1) \) and \( s \in \mathcal{L}(re_2) \) (i.e., \( s \in \mathcal{L}(re_1) \cup \mathcal{L}(re_2) \)).
- **Complement.** A string \( s \in \mathcal{L}(\sim re_1) \) iff \( s \notin \mathcal{L}(re_1) \).

Note that \(!\), \#, \&, and \(\sim\) operators are not typically in regular expression constructs in programming languages (e.g., in JavaScript, Perl), though all others are almost always present.

You may complete the following parts in mostly any order. Matching is the most challenging part, but we suggest you get at least a few of simpler cases done first before working on parsing.

(a) **Regular Expression Matcher: Continuations.** For this exercise, we will implement a basic backtracking regular expression matcher.

We will write a regular expression matcher

```python
def retest(re: RegExpr, s: String): Boolean
```

that given a regular expression \( re \) and a string \( s \) returns **true** if the string \( s \) belongs to the language described by the regular expression \( re \) and otherwise returns **false**. The implementation of \( \text{retest} \) is provided for you, which calls a helper function \( \text{test} \) that you provide. This function corresponds to a restricted implementation of the \( test \) method on \( \text{RegExp} \) objects in JavaScript. For simplicity, we consider only whole string matches, which would be equivalent to

```
/\sim re$/ .test(s)
```

in JavaScript.

We will implement our regular expression matcher using continuations. In particular, we will implement a helper function:

```python
def test(re: RegExpr, chars: List[Char],
        sc: List[Char] => Boolean): Boolean
```

This helper function will see if a **prefix** of \( chars \) matches the regular expression \( re \). If there is a prefix match, then the success continuation is called with the remainder of \( chars \) that has yet to be matched. That is, the success continuation \( sc \) captures “what to do next if a prefix of \( chars \) successfully matches \( re \).” If \( \text{test} \) discovers a failure to match, then it can “return **false** early.”

A continuation a special kind of callback in that its a callback for the higher-order function itself. In this example, \( \text{test} \) is a higher-order function that takes in a continuation
that is callback for itself “in the future,” that is, in a recursive call. More precisely, a
continuation captures an action to do on return. It accumulates an action that should
be performed after the current downwards recursive sequence is complete.
The idea of continuations is an underlying concept in asynchronous programming,
such as in client-server systems. For example, Node.js forces continuation-style pro-
gramming because all I/O operations are asynchronous.

**Extra Credit.** The RIntersect and RNeg cases will be considered extra credit. All other
cases are part of this problem.

**Hints.**

- **Star** is the most difficult case (that is not extra credit). Consider completing the
  other cases first. Implementing the Concat case might help clarify how the suc-
cess continuation is used.
- From a general algorithm level, our regular expression matcher and our recursive
descent parser (in the next part). They are both backtracking search algorithms
on an input string. The input string is treated like a stream of characters where
we iteratively try to “consume” from the front of the stream according to some
constraints (i.e., “Does it match ‘this’ part of the regular expression?” or “Does it
match ‘this’ production?”). When the first try fails, we go on to try the next possi-
bility and so forth.

(b) **Regular Expression Parser: Recursive Descent Parsing**

To specify, how the concrete syntax of regular expressions is transformed into abstract
syntax. We resolve the ambiguity in the grammar given above by saying that all binary
operators should be left associative and the precedence of the operators are ordered as
follows (from highest precedence to lowest): {*,+,?,∼, (juxtaposition for concatena-
tion), &, and |. A set {...} means all of the operators are at the same precedence level.

In this part, we will implement one of the most basic parsing algorithms: recursive de-
scent parsing. Recursive descent parsing is generic pattern for manually constructing
parsers for simple languages. Parsing is an area rich with numerous more efficient al-
gorithms, useful tools, and interesting techniques, as well as elegant theory. For more
complex grammars, one often uses a type tool called a parser generator. Using such
tools is generally a topic in a compilers course.

A recursive descent parser works top-down in the grammar and left-to-right in the in-
put string. The basic pattern is to write a recursive function for each non-terminal in
the grammar. Each parsing function tries to parse (and consume) characters from the
input string by applying each production for that non-terminal in sequence. If apply-
ing production fails, then it backtracks and tries the next one. Applying a production
means either (1) consuming characters from the left of the string to match a termi-
nal or (2) calling the function corresponding to a non-terminal to try to match that
non-terminal. In the end, one ends up with a set of mutually recursive functions that
parallels the structure of the grammar.

The first task for constructing a recursive descent parser is to refactor the grammar
to eliminate ambiguity. But not any unambiguous grammar with do. One key re-
quirement for recursive descent parsing is that the grammars must not contain left
recursion. Otherwise, your parser will go into an infinite loop (why?).

Thus, obtaining left associativity requires a little bit more. To get there, we consider an extension of the meta-language for describing grammars with the braces \{a\} to mean a sequence of 0-or-more of a. This notation is part of what’s known as EBNF (Extended Backus-Naur Form). We give a refactored version of the previous grammar that eliminates ambiguity.

\[
\begin{align*}
  \text{re} &::= \text{union} \\
  \text{union} &::= \text{intersect} \{'|' \text{intersect}\} \\
  \text{intersect} &::= \text{concat} \{'&' \text{concat}\} \\
  \text{concat} &::= \text{not} \{ \text{not} \} \\
  \text{not} &::= \sim \text{star} | \text{star} \\
  \text{star} &::= \text{atom} \{'\ast' | 'c' | 'd' \} \\
  \text{atom} &::= '!' | '#' | c | 'l' | ('re')
\end{align*}
\]

The quotes ‘ ’ emphasize the symbols that are terminals in the object language (rather than meta-level symbols of the grammar notation). The 0-or-more sequence operator can be translated into BNF by using another non-terminal, which is what we need for our recursive descent parser. As an example, we can translate

\[
\text{union} ::= \text{intersect} \{'|' \text{intersect}\}
\]

in EBNF to

\[
\begin{align*}
  \text{union} &::= \text{intersect} \text{intersects} \\
  \text{intersects} &::= \epsilon | '|' \text{intersect}
\end{align*}
\]

in BNF. We can now use this grammar to structure our recursive descent parser. Notice that \text{intersects} looks a lot like a list. Now, we can enforce a left associativity of | by constructing RUnion abstract syntax nodes as we “fold-left” across the intersect “elements.”

Scala includes a powerful library for constructing parsers. We will use a small bit of that library to handle input and parsing results. We will implement a Scala object called RegExprParser that derives from Parsers:

```scala
object RegExprParser extends Parsers {
  type Elem = Char
  def re(next: Input): ParseResult[RegExpr] = ...
}
```

In this object, you should define your mutually recursive functions that define a recursive descent parser (one for each non-terminal). For example, the top-level function is re that takes a parameter of type Input and returns something of type ParseResult. These two types are inherited from Parsers. The relevant parts are as follows:

```scala
type Input = Reader[Elem]
sealed abstract class ParseResult[+T] {
  val next: Input
  def map[U](f: T => U): ParseResult[U]
}
```
The `ParseResult` type is somewhat similar to the `Option` type. A successful parse is indicated by returning a `Success` that bundles the result (in our case a `RegExpr`) and the remaining input. The `Failure` case class indicates a parse failure with an error message and the remaining input.

Scala's parsing library is a combinator parsing library. A combinator means essentially a higher-order function. So a combinator parsing library is a set of higher-order functions in library that one uses to construct parsers. The algorithm of a parser constructed using the parser combinators is very similar to the one that you implement in your recursive descent parser. What you might observe after you complete your recursive descent parser is that there is a lot of repeated boilerplate in your code (and then just imagine how much repetition there would be in a larger language like JavaScript!). What the parser combinators do is essentially factor out the boilerplate into a generic library that can be used, much like the higher-order methods in the collection classes.

After you have constructed your recursive descent parser in this exercise, you will be able to approach Scala's combinator parsing library and write your future parsers with it!

A reference implementation of the regular expression parser built using Scala's combinator parsing library is given in `RegExprParser.scala`. The code will probably not make much sense until you have written your recursive descent parser, but then, it will be interesting to compare. You can also use the reference parser implementation to work on the matcher before completing your parser.

(c) **Regular Expression Literals in JavaScript**. Let's extend our Lab 5 interpreter with regular expression literals and regular expression matching. We extend JavaScript as follows:

```plaintext
expressions  e ::= ··· | /* re$ */
values       v ::= ··· | /* re$ */
types        τ ::= ··· | RegExp
```

to specify regular expression literals. We will treat regular expression literals as values and introduce a type for regular expressions `RegExp`. To represent regular expression literals, we use the following case classes:

```scala
case class RE(re: RegExpr) extends Expr
case class TRegExp extends Typ
```

To match JavaScript, we write the regular expression test operator as follows:

```scala
e1.test(e2)
```

where `e1` should have type `RegExp` and `e2` should have type `string`. We do not introduce a new abstract syntax node, as the above will already parse as

```scala
Call(GetField(e1, "test"), List(e2))
```
and special case matching for this in `typeInfer` and `step` before doing the usual actions for a `Call` node. To “do” rule in `step` will call your run-time function `restep` that you wrote in Scala (instead for example as a library in `JAVASCRIPTY`). One note is that ideally, we still want to permit fields named `test` in objects that store functions, so the above expression is only a regular expression test if `e_1` is of type `RegExp`. As an aside, this observation suggests that a potential translation to a distinguished “regular expression test” AST node from the above would have to be done during type analysis. We will not do this, as such a translation seems overkill for our later phases but such translations are important software architectural decisions in structuring an interpreter/compiler.

i. Give typing and small-step operational semantic rules for regular expression literals and regular expression tests based on the informal specification given above. Clearly and concisely explain how your rules enforce the constraints given above and any additional decisions you made.

ii. Extend your Lab 5 interpreters. We will limit the evaluation to this extension as much as possible (e.g., we will try to avoid exercising the language extensions introduced in Lab 5). The interpreter part of the template is the same as was for Lab 5, except with the additions for regular expressions. In actuality, for this part, you only need to finish the template with the rules you gave in the previous part (which excludes Lab 5 features).

3. **JavaScripty Fun**

   (a) **Interpreter Debugging Competition.** In any reasonably rich programming language (like `JAVASCRIPTY`), the space of possible programs becomes huge, and it’s the job of an interpreter (or compiler) to handle all of them in a predictable way. As we have experienced, language implementations are challenging to get correct. For this question, you should think deviously and construct test cases that you think will break your classmates' interpreters.

   As a class, we will hold a competition to see who is best at breaking interpreters and who is best at constructing correct interpreters. The top 3 in each category will receive extra credit. You will receive “normal” credit for this credit if you participate actively (i.e., submit interesting test cases).

   The ground rules are as follows:

   - Your test cases must adhere to the `JAVASCRIPTY` language specification to be counted in the competition. The correctness of an interpreter is based on the operational semantics. Extra extra credit for any bugs found in the reference implementation.
   - In your breaking total, a test case will only count if your interpreter passes it.