CSCI 3155: Lab Assignment 6

Checkpoint: Friday, April 8, 2016 at 6pm
Due Date: Friday, April 15, 2016 at 6pm
Lightning talk presentations due separately the week of April 18,
with additional intermediate dates

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, specifically parsing. We will also have a chance to play with the cool interpreters that we have built.

Concretely, we will consider regular expressions. We construct a parser for a language of regular expressions and implement a regular expression matcher in Scala. For fun, we extend our Lab 5 interpreters with regular expression literals and regular expression matching (like JavaScript) using your parser and expression matcher. There's also a chance to get some extra credit.

PL Ideas Recursive descent parsing. Regular languages.

FP Skills Backtracking search with continuations.

Instructions. As usual we ask you to find a new partner for this lab assignment (different from your partners in prior labs). You will work on this assignment closely with your partner. However, note that **each student needs to submit** 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 questions with which you discussed, including your partner and those outside of your pair.

Recall the evaluation guideline from the course syllabus.

> *Both your ideas and also the clarity with which they are expressed matter—both in your English prose and your code!*
>
> *We will consider the following criteria in our grading:*
>
> - How well does your submission answer the questions? *For example, a common mistake is to give an example when a question asks for an explanation. An example may be useful in your explanation, but it should not take the place of the explanation.*

- How clear is your submission? *If we cannot understand what you are trying to say, then we cannot give you points for it. Try reading your answer aloud to yourself or a friend; this technique is often a great way to identify holes in your reasoning. For code, not every program that "works" deserves full credit. We must be able to read and understand your intent. Make sure you state any pre-conditions or invariants for your functions (either in comments, as assertions, or as* `require` *clauses as appropriate).*

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 on COG. A program that does not compile will *not* be graded.

Submission Instructions.   Upload to Moodle exactly four files named as follows:

- `Lab6-`*YourIdentiKey*`.scala` with your answers to the coding exercises

- `Lab6Spec-`*YourIdentiKey*`.scala` with any updates to your unit tests. Note that there are no closed autograder tests for this assignment.

- `Lab6-`*YourIdentiKey*`.jsy` with a challenging test case for your JAVASCRIPTY interpreter. Do not use the extra credit regular expressions.

- `Lab6-`*YourIdentiKey*`.pdf` with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)

Replace *YourIdentiKey* with your IdentiKey (e.g., I would submit `Lab6-bec.pdf` and so forth). Don't use your student identification number. To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Submit your `Lab6.scala` file to COG for auto-testing. We ask that you submit both to COG and to Moodle in case of any issues.

Sign-up for an interview slot for an evaluation. To fairly accommodate everyone, the interview times are strict and **will not be rescheduled**. Missing an interview slot means missing the interview evaluation component of your lab grade. Please take advantage of your interview time to maximize the feedback that you are able receive. Arrive at your interview ready to show your implementation and your written responses. Implementations that do not compile and run will not be evaluated.

Getting Started.   Clone the code from the Github repository with the following command:

```
git clone -b lab6 https://github.com/bechang/pppl-labs.git   lab6
```

1. **Feedback**. Complete the survey linked on 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).

$$
\begin{array}{llll}
\text{regular expressions} & re ::= & ! & \text{no string (RNoString)}\\
& | \, \# & & \text{empty string (REmptyString)}\\
& | \, . & & \text{any character (RAnyChar)}\\
& | \, c & & \text{the character } c \text{ (RSingle}(c)\text{)}\\
& | \, re_1 \, re_2 & & \text{concatenation (RConcat}(re_1, re_2)\text{)}\\
& | \, re_1 \text{`|'} re_2 & & \text{union, "or" (RUnion}(re_1, re_2)\text{)}\\
& | \, re_1 * & & \text{Kleene star, "0-or-more" (RStar}(re_1)\text{)}\\
& | \, re_1 + & & \text{"1-or-more" (RPlus}(re_1)\text{)}\\
& | \, re_1 ? & & \text{"0-or-1" (ROption}(re_1)\text{)}\\
& | \, re_1 \& re_2 & & \text{intersection, "and" (RIntersect}(re_1, re_2)\text{)}\\
& | \, {\sim} re_1 & & \text{complement, "not" (RNeg}(re_1)\text{)}
\end{array}
$$

```
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 $\mathscr{L}(re)$ for the language specified by the regular expression re:

- $\mathscr{L}(!) \stackrel{\text{def}}{=} \varnothing$, that is, the empty set.

- $\mathscr{L}(\#) \stackrel{\text{def}}{=} \{""\}$, that is, the set with the empty string.

- $\mathscr{L}(c) \stackrel{\text{def}}{=} \{"c"\}$, that is, the set with the string matching the single character c.

- **Concatenation**. A string $s = s_1 s_2 \in \mathscr{L}(re_1 \, re_2)$ iff $s_1 \in \mathscr{L}(re_1)$ and $s_2 \in \mathscr{L}(re_2)$.

- **Union**. A string $s \in \mathscr{L}(re_1|re_2)$ iff $s \in \mathscr{L}(re_1)$ or $s \in \mathscr{L}(re_2)$ (i.e., $s \in \mathscr{L}(re_1) \cup \mathscr{L}(re_2)$).

- **Kleene Star**. A string $s \in \mathscr{L}(re_1 *)$ iff s is in zero-or-more concatenations of re_1.

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 $s \in \mathscr{L}(.)$ iff s consists of any single character.

3

- **One-Or-More**. A string $s \in \mathscr{L}(re_1+)$ iff s is in one-or-more concatenations of re_1 (i.e., $s \in \mathscr{L}(re_1\,re_1*)$).

- **Zero-Or-More**. A string $s \in \mathscr{L}(re_1?)$ iff s is in zero-or-one matches of re_1 (i.e., $s \in \mathscr{L}(\#|re_1)$).

- **Intersection**. A string $s \in \mathscr{L}(re_1\&re_2)$ iff $s \in \mathscr{L}(re_1)$ and $s \in \mathscr{L}(re_2)$ (i.e., $s \in \mathscr{L}(re_1) \cap \mathscr{L}(re_2)$).

- **Complement**. A string $s \in \mathscr{L}(\sim re_1)$ iff $s \notin \mathscr{L}(re_1)$.

Note that !, #, &, and ~ 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

  ```
  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 `retest` is provided for you, which calls a helper function `test` that you provide. This function corresponds to a restricted implementation of the `test` method on `RegExp` objects in JavaScript. For simplicity, we consider only whole string matches, which would be equivalent to

  $$/\hat{}\,re\$/.\text{test}(s)$$

  in JavaScript.

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

  ```
  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 `sc` 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 `test` discovers a failure to match, then it can "return **false** early."

  A continuation a special kind of callback in that it is a callback for the higher-order function itself. In this example, `test` is a higher-order function that takes in a continuation `sc` 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.

4

The idea of continuations is an underlying concept in asynchronous programming, such as in client-server systems and interactive systems (e.g., mobile apps or any GUI-based app). For example, Node.js forces continuation-style programming 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.

Additional extra credit is also available for implementing other parts of the JavaScript specification of regular expressions, such as group captures. You will be able to further extend the regex AST; and determine the corresponding semantics. Any such extra credit submission must demonstrate strong unit testing.

Hints.

- `RStar` is the most difficult case (that is not extra credit). Consider completing the other cases first. Implementing the `RConcat` case might help clarify how the success continuation is used.

- From an 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 possibility 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): $\{*, +, ?\}$, $\sim$, ␣ (juxtaposition for concatenation), &, and |. A set $\{\ldots\}$ 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 descent parsing. Recursive descent parsing is generic pattern for manually constructing parsers for simple languages. Parsing is an area rich with numerous more efficient algorithms, 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 input 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 applying a 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 terminal 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 require-
ment for recursive descent parsing is that the grammars must not contain left recur-
sion. Otherwise, your parser will go into an infinite loop (why?).

  i. In your write-up, give a refactored version of the *re* grammar that eliminates am-
     biguity in BNF (not EBNF) using left recursion. Use the new non-terminal names
     from the EBNF grammar below (*union*, *intersect*, etc.).

 ii. Explain briefly why a recursive descent parser following your grammar with left
     recursion would go into an infinite loop.

Without left recursion, obtaining left associativity requires a little bit more work. To get
there, we consider an extension of the meta-language for describing grammars with
the braces $\{\alpha\}$ to mean a sequence of 0-or-more of α. 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.

$$
\begin{aligned}
re &::= union \\
union &::= intersect \, \{ \, `|\text{'} \, intersect \, \} \\
intersect &::= concat \, \{ \, `\&\text{'} \, concat \, \} \\
concat &::= not \, \{ \, not \, \} \\
not &::= `\sim\text{'} \, not \mid star \\
star &::= atom \, \{ \, `*\text{'} \mid `+\text{'} \mid `?\text{'} \, \} \\
atom &::= `!\text{'} \mid `\#\text{'} \mid c \mid `.\text{'} \mid `(\text{'} \, re \, `)\text{'}
\end{aligned}
$$

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

$$
union ::= intersect \, \{ \, `|\text{'} \, intersect \, \}
$$

in EBNF to

$$
\begin{aligned}
union &::= intersect \; unions \\
unions &::= \varepsilon \mid `|\text{'} \, intersect \; unions
\end{aligned}
$$

in BNF. We can now use this grammar to structure our recursive descent parser. Notice
that *unions* looks a lot like a list. Now, we can enforce left associativity of | by construct-
ing RUnion abstract syntax nodes as we "fold-left" across the *intersect* "elements."

iii. In your write-up, give the full refactored grammar in BNF without left recursion
     and new non-terminals like *unions* for lists of symbols. You will need to introduce
     new terminals for *intersects* and so forth.

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:

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

6

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]
}
case class Success[T](result: T, next: Input) extends ParseResult[T]
case class Failure(msg: String, next: Input) extends ParseResult[Nothing]
```

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. To determine what `RegExpr` abstract syntax tree to construct on a successful parse, you should consult the specification in the original, unrefactored grammar for *re*. 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 a 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 JAVASCRIPTY!). 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 your implementation to it. You can also use the reference parser implementation to work on the matcher before completing your parser.

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

$$
\begin{array}{lll}
\text{expressions} & e ::= \cdots \mid /\hat{}\,re\$/ \\
\text{values} & v ::= \cdots \mid /\hat{}\,re\$/ \\
\text{types} & \tau ::= \cdots \mid \textbf{RegExp}
\end{array}
$$

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:

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

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

e_1`.test(`e_2`)`

where e_1 should have type **RegExp** and e_2 should have type **string**. We do not introduce a new abstract syntax node, as the above will already parse as

`Call(GetField(`e_1`, "test"), List(`e_2`))`

and special case matching for this in `typeInfer` and `step` before doing the usual actions for a `Call` node. The "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. In your write-up, 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. **Optional:** Extend your Lab 5 interpreter to include regular expressions.
The discussion above enables you to extend your Lab 5 interpreter so that you can have your own full JAVASCRIPTY interpreter!

3. **Lightning talk mini-project**

In your lightning talk you will present on how a specific concrete language feature — syntax and corresponding semantics — is added to a given language.

Consider the context. Successful programming languages tend to evolve over time, especially as part of their overall developer community. Some languages use standards committees to represent this community; others use more informal processes, such as mailing lists and issue trackers. Either way, any potential evolution is frequently encapsulated into a change proposal, which the community can evaluate, discuss, and further enhance. Frequently the change proposal is rejected outright, but sometimes it may live on in some other way, or even influence another language.

For the lightning talk pick a language from the set of Java, JavaScript, Python, or Scala. Restrict the change proposals to those that are proposed for one of the following versions:

   1. Java 8 or Java 9

   2. JavaScript, as defined by EcmaScript 6 or EcmaScript 7

3. Python 3.3, 3.4, or 3.5
4. Scala 2.10, 2.11, 2.12

This change must be substantive; and it must reflect a change in both syntax and semantics of the language. For Scala, such syntax changes can of course be implemented by Scala's support of internal DSLs. What is substantive can perhaps be best illustrated by a counter example: "Floating point and octal literal syntax deprecation in Scala 2.10" is not a sufficiently substantive topic.

This change proposal may be currently in a proposed state, or it may now be rejected, accepted, or implemented. You may find it most straightforward to write about accepted proposals that are now implemented. However, rejected proposals may be of especial interest if the language later returned to that proposed idea, but in some other way.

Change proposals may be in the form of a PEP, SIP, JEP, JSR, or similar document. You will want to also discuss primary sources like development mailing lists to find out about what was said about pros and cons. You may want consider subsequent proposals that cite the proposal as part of the development of your theme.

It's up to you what aspects of the syntax and semantics to present, but regardless describe using sample code and/or test cases. It's best to keep with the formalism of the change proposal itself, but if you want to elaborate using judgments and inference rules as motivated by the class, that can be a great aspect of your talk.

Your work is public and will be presented in class. All presentation material, including especially titles and other wording (and this includes all GitHub identifiers), should be acceptable for presentation at a professional conference.

For the lightning talk mini-project, you will create the following aspects of your presentation:

- A 5 minute, 20 slide talk on your topic to be given in class/recitation; then posted publicly on YouTube, using a steady camera and clear sound. In either case, slides will be advanced every 15 seconds automatically. This means your lightning talk follows an Ignite style presentation format.
- Generate your slides using the Markdown markup language. A template will be provided.
- Merging your work, via a pull request, in the corresponding GitHub repo for the course. This merged work will contain the source for the slides in your talk, including Markdown text, code samples, and any images.

We will post on Piazza the GitHub repo for you to fork, then submit a pull request against. This repo will also include a template to build your presentation.

Topics should be picked after consultation with your TA by Friday April 8 at 6p; a GitHub pull request submitted by Monday April 11 at 6p with work-in-progress items; merged into the class repo by Monday April 18 at 6p; and presented in class or during recitations the week of April 18.

Be creative and have fun!