CSCI 3155: Lab Assignment 6

Learning Goals. The primary learning goals of this lab are to understand the following:

- thinking inductively and continuations; and
- recursive descent parsing.

PL Ideas Recursive descent parsing. Regular languages.

FP Skills Continuations (in continuation-passing style). Thinking inductively.

Algorithmic Ideas Backtracking search.

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. We will also have a chance 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. For fun, we extend our Lab 5 interpreters with regular expression literals and regular expression matching (like JavaScript) using your parser and expression matcher.

General Guidelines. During recitation find a partner for this lab assignment (should be different for every lab assignment). You will work on this assignment closely with your partner. However, note that **each student is individually responsible** for completing the assignment so that you can do well in your interview.

You are welcome to talk about these questions beyond your teams. However, we ask that you code in pairs. See the collaboration policy for details, including the following:

> *Bottom line, feel free to use resources that are available to you as long as the use is **reasonable** and you **cite** them in your submission. However, copying answers directly or indirectly from solution manuals, web pages, or your peers is certainly unreasonable.*

Also, 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 via `sbt test`. A program that does not compile will *not* be graded—no interview will be conducted.

Submission Instructions. We are using the following tools:

- Github for assignment distribution and submission;
- INGInious for auto-testing submission;
- Moodle for archival and as a backup for assignment submission.

1. You will be editing and submitting the the following files to Github:

   - `src/main/scala/jsy/student/Lab6.scala` with your solution to the coding exercises;
   - `src/test/scala/jsy/student/Lab6Spec.scala` with your additional tests; and
   - `lab6-writeup.pdf` or `lab6-writeup.md` for a pdf or a Markdown document that should be pushed to the root directory of your repository with your response to the written questions (scanned, clearly legible handwritten write-ups are acceptable). **You will not get credit for write-ups in any other file format.**

   You are also likely to edit `src/main/scala/jsy/student/Lab6Worksheet.sc` for any scratch work.

   You need to have a Github identity and should have your full name in your Github profile so that we can associate you with your submissions if there are any confusions.

   Following good git practice, please make commits in small bits corresponding to completing small conceptual parts and push often so that your progress is evident. We expect that you have some familarity with git from prior courses or experience. If not, please discuss with your classmates and the course staff (e.g., via Piazza).

2. At any point, you may submit your `Lab6.scala` file to INGInious for auto-testing. You need to submit to INGInious for the auto-testing part of your score, as well as to continue to the interview.

3. Sign-up for an interview slot for an evaluator. 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 score. Please take advantage of your interview time to maximize the feedback that you are able receive. Arrive at your interview ready to show your team's implementation and your written responses. Implementations that do not compile and run will not be evaluated.

4. As a backup for any submission issues and archiving, please help us by also uploading to the moodle exactly the files named above, that is,

   - `Lab6.scala`
   - `Lab6Spec.scala`
   - `lab6-writeup.pdf` or `lab6-writeup.md`

Getting Started. First, form a team of two and pick a team name. For our bookkeeping, please prefix your team name with `lab6-` (e.g., `lab6-anatomists`).

You must work in teams of two, and you will form teams in lab section. If you miss lab section on the day teams are formed, you need to find a partner on your own. If you really, really cannot find a partner, then please contact the course staff (via Piazza).

Then, log into moodle and follow the Github Classroom link for setting up your Lab 6 repository with your team name. The first person will create the team, and the second person will select the team name from the existing team names.

If you would like to look at the code before getting your own copy for submission, you may go to `https://github.com/csci3155/pppl-lab6`.

Checkpoint. The checkpoint is to encourage you to start the coding portion of the assignment early and it requires you to submit your partial solution on INGInious a week before the assignment is due. You do not need to complete all coding a week early but we want you to start working on it. This means that submitting the empty template that fails all tests is **not sufficient**. Failing to submit to the checkpoint will prevent you from proceeding to the interview. However, as long as you pass the checkpoint, this early score from the checkpoint will not affect your grade for the assignment or your overall grade for the course.

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

2. **Warm-Up: Continuations**. To implement our regular expression matcher, we introduce the idea of a continuation that captures "what to do next." Consider again the binary search tree data structure from Labs 1 and 4:

   ```
   sealed abstract class Tree
   case object Empty extends Tree
   case class Node(l: Tree, d: Int, r: Tree) extends Tree
   ```

   (a) First, let us reimplement `foldLeft` to take an additional parameter `sc` called the *success continuation*.

```
def foldLeftAndThen[A,B](t: Tree)(z: A)(f: (A,Int) => A)(sc: A => B): B
```

This function performs an in-order traversal of the input tree `t` calling the callback `f` to accumulate a result that is passed to `sc`. Suppose the in-order traversal of the input tree yields the following sequence of data values: $d_1, d_2, \ldots, d_n$. Then, `foldLeftAndThen` yields

$$\texttt{sc}(\texttt{f}(\cdots(\texttt{f}(\texttt{f}(\texttt{z}, d_1), d_2))\cdots), d_n))\ .$$

The inner-helper function `loop` has been refactored with a similiar signature with an analogous `sc` parameter.

```
def loop(acc: A, t: Tree)(sc: A => B): B
```

The specification of `loop` is to yield

$$\texttt{sc}(\texttt{f}(\cdots(\texttt{f}(\texttt{f}(\texttt{acc}, d_i), d_{i+1}))\cdots), d_n))\ .$$

assuming the in-order traversal of the tree `t` yields the following sequence of data values: $d_i, d_{i+1}, \ldots, d_n$. The challenge in this question is to make the `loop` function tail recursive by a choice of the `sc` argument in the recursive call.

(b) Then, we implement `dfs` for a depth-first search of the tree:

```
def dfs[A](t: Tree)(f: Int => Boolean)(sc: List[Int] => A)(fc: () => A): A
```

The callback `f` is invoked on each element to decide whether the desired element has been found. If `f`(i) for an element i returns **true**, then the `dfs` function returns

$$\texttt{sc}(i :: \cdots :: r :: \texttt{Nil})$$

where $i :: \cdots :: r :: \texttt{Nil}$ is the list of elements from the found element to the root element r. Otherwise, `dfs` returns `fc`(). Like in the previous part, you should make the inner-helper function `loop` tail recursive.

3. **Regular Expressions.** Consider the syntax for a language of regular expressions shown in Figure 1. 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).

   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:

   - $\mathscr{L}(!) \stackrel{\text{def}}{=} \emptyset$, 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.

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

```
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
```

Figure 1: Syntax of regular expressions.

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.
- **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-One**. 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

/^*re*$/.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's 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.

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.

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): $\{*, +, ?\}$, ~, ␣(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 umambiguous grammar with do. One key requirement for recursive descent parsing is that the grammars must not contain left recursion. Otherwise, your parser will go into an infinite loop (why?).

i. **Exercise**. In your write-up, give a refactored version of the *re* grammar from Figure 1 that eliminates ambiguity in BNF (not EBNF). Use the following template for the new non-terminal names:

$$
\begin{aligned}
\textit{re} &::= \cdots \\
\textit{union} &::= \cdots \\
\textit{intersect} &::= \cdots \\
\textit{concat} &::= \cdots \\
\textit{not} &::= \cdots \\
\textit{star} &::= \cdots \\
\textit{atom} &::= \cdots
\end{aligned}
$$

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

We cannot use the above grammar as a specification to implement our recursive descent parser because of the left recursion, but we also want some left associative operators, so we are at a bit of impasse.

Obtaining left associativity requires a little bit more work. The basic idea is that we will turn the binary operators that we want to be left associative into n-ary operators in the parse tree. Then, in the parser, we will construct the *abstract syntax tree* with binary, left-associative operators from the parse tree with n-ary operators.

As an intermediate step, 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).

iii. **Exercise**. In your write-up, give a refactored version of the *re* grammar that replaces left-associative binary operators with n-ary versions using EBNF using the following template:

$$
\begin{aligned}
\textit{re} &::= \textit{union} \\
\textit{union} &::= \textit{intersect}\ \{\, \text{`|'}\ \textit{intersect}\,\} \\
\textit{intersect} &::= \cdots \\
\textit{concat} &::= \cdots \\
\textit{not} &::= \cdots \\
\textit{star} &::= \cdots \\
\textit{atom} &::= \cdots
\end{aligned}
$$

The quotes ‘ ’ in ‘émphasize that the symbol is a terminal 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

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

in EBNF to

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

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 constructing `RUnion` abstract syntax nodes as we "fold-left" across the *intersect* "elements."

iv. **Exercise**. 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`:

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

Your `REParser` object implements this interface where 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:

```
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 `jsy.lab6.RegExprParser`. 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:

expressions	$e ::= \cdots \mid /\hat{}\ re\$/$
values	$v ::= \cdots \mid /\hat{}\ re\$/$
types	$\tau ::= \cdots \mid \textbf{RegExp}$

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:

```
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 `typeof` and `step` before doing the usual actions for a `Call` node. The "do" rule in `step` will call your run-time function `retest` 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 distiguished "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. **Exercise**. 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!