Lecture: Introduction to Regular Expressions

Required Reading

- Java Regular Expressions
- Scala Regular Expressions

Regular Expressions

Many programs need to do some form of text processing and *regular expressions* are the most fundamental and pervasive text processing tools. Average web pages use regular expressions to validate emails, phone numbers, URLs, and so on; IDEs and programmers' text editors support regular-expression based search and replace; regular expressions are often used to "clean" data; and regular expressions are also used in virus scanners, network intrusion detection systems, and other computer security products.

You can think of a regular expression as a pattern that describes a set of strings. For example, the regular expression `a.*z` describes all strings that start with `'a'` and end with `'z'`. In a regular expression, the character `.` means "any character" and the symbol `*` means "zero or more repetitions". Therefore, we can read `a.*z` as "a, followed by zero or more repetitions of any character, followed by 'z"'.

In Scala, you can construct a regular expression by written it as a string and invoking the `.r` method. For example:

```
val regex = "a.*z".r
```

We can use the `regex.findFirstIn` method to match a string:

```
regex.findFirstIn(str)
```

As the name suggests, the method finds the first substring of `str` that matches the regular expression. For example, if `str` is `"hello abcdz bye"`, the expression produces `Some("abcdz")`. However, if no substring matches the regular expression, it produces `None`.

Notation Regular expressions often use special characters that need to be quoted within strings. For example, the regular expression `\d` matches a single decimal digit. However, to write this as a string, we'd have to quote the backslash (`"\d"`). This becomes cumbersome very quickly. However, Scala has a special syntax for defining strings that is very handy in this situation. In a string that is enclosed with triple-quotes, a backslash is interpreted as a literal character and not as the start of an escape sequence. Therefore, we can just write `"""\d"""`, which is equivalent to `"\d"`.

A regular expression can be built in several ways:

1. *Literal characters* that match characters exactly, for example `a` is a regular expression that only matches the string `"a"`.

2. A *metacharacters*, such as `.`, matches any character. For example, the regular expression `...` matches all strings of length three. If we want to match only the string `"..."`, we need to escape the metacharacters and write the regular expression `\.\.\.`. There are a few other metacharacters and they can all be matched by escaping with with a backslash.

3. A *character range*, such as `[0-9]` matches any digit. Several ranges can be composed together. For example, `[A-Za-z]` matches uppercase and lowercase letters.

4. A *range complement* matches all characters not in a range. For example, `[^ 0-9]` matches all non-numeric characters.

5. *Alternation* can be used to match one of several regular expressions. For example, `(abc)|(xyz)` matches either `"abc"` or `"xyz"`. Similarly, `a(b|c)d` matches either `"abd"` or `"acd"`.

6. *Iteration* can be used to repeat a pattern several times. For example, you can read`a*` as the pattern `a` repeated zero or more times. Therefore, this regular expression matches `""`, `"a"`, `"aa"`, and so on. Similarly, `(hi)*` matches `""`, `"hi"`, `"hihi"`, `"hihihi"`, and so on.

   There are several other operators that apply a pattern several times:

   - *regex*`+` repeats *regex* one or more times.
   - *regex*`?` matches either the empty string or *regex*.
   - *regex*`{m,n}` repeats *regex* between m and n times.

7. A *character class*, such as `\d` matches a digit.

Examples

Undergraduate Computer Science Classes Computer science classes begin with the prefix `"COMPSCI"`. However, undergraduate classes are numbered 499 or lower. The following regular expression matches undergraduate computer science classes:

```
COMPSCI[1-4].*
```

Credit Card Numbers A Visa or MasterCard has 16 digits, so the simplest regular expression to match then would be `16`. However, people tend to format credit card numbers by grouping them into blocks of four digits and separating the blocks with spaces:

```
\d{4} \d{4} \d{4} \d{4}
```

—or with hyphens—

```
\d{4}-\d{4}-\d{4}-\d{4}
```

We can make both regular expressions more compact by rewriting them as `\d{4}( \d{4}){3}` and `\d{4}(-\d{4}){3}` respectively.

We can also make accept both formats by writing:

```
(\d{4} \d{4} \d{4} \d{4})|(\d{4}-\d{4}-\d{4}-\d{4})
```

—or, using the more compact representation, we could write:

```
(\d{4}( \d{4}){3})|(\d{4}(-\d{4}){3})
```

> **Think!** What does the following regular expression match?
> `\d{4}([ -]\d{4}){3}`

Even Numbers The following regular expression matches all even numbers:

```
\d*(0|2|4|6|8)
```