Advanced RegEx

Week 14 – Presentation 3

The effect of eager matching on regular expression alternation

This regular expression engine behavior may return unexpected matches in alternation if alternations are not ordered carefully in the regex pattern.

Take an example of this regex pattern, which matches the strings `white` or `whitewash`:

```
white|whitewash
```

While applying this regex against an input of *whitewash*, the regex engine finds that the first alternative `white` matches the *white* substring of the input string *whitewash*, hence, the regex engine stops proceeding further and returns the match as `white`.

Note that our regex pattern has a better second alternative as `whitewash`, but due to the regex engine's eagerness to complete and return the match, the first alternative is returned as a match and the second alternative is ignored.

However, consider swapping the positions of the third and fourth alternatives in our regex pattern to make it as follows:

```
whitewash|white
```

If we apply this against the same input, *whitewash*, then the regex engine correctly returns the match as `whitewash`.

Basic quantifiers

The following table lists all the quantifiers available in Java regular expressions:

Quantifier	Meaning
m*	Match **m** zero or more times
m+	Match **m** one or more times
m?	Match **m** one or zero times (also called an optional match)
m{X}	Match **m** exactly *X* times
m{X,}	Match **m** *X* or more times
m{X,Y}	Match **m** at least *X* and at most *Y* times

In all the aforementioned cases, ***m*** *can be a single character or a group of characters. We will discuss grouping in more detail later.*

Examples using quantifiers

Let's look at few examples to understand these basic quantifiers better.

Which regex pattern should be used to match a two-digit year or a four-digit year?

```
\d{2}|\d{4}
```

Which regex pattern should be used to match a signed decimal number? The pattern should also match a signed integer number:

```
^[+-]?\d*\.?\d+$
```

Here is the breakup of the preceding regex pattern:

- The `^` and `$` symbols are the start/end anchors
- The `[+-]?` pattern makes either the `+` sign or the `-` sign (optional because of `?`) at the start
- The `\d*` pattern matches zero or more digits
- The `\.?` pattern matches an optional dot (.) literally
- The `\d+` pattern matches one or more digits

The preceding regex will match all of these inputs:

- `.45`
- `123789`
- `5`
- `123.45`
- `+67.66`
- `-987.34`

What would be the regex to match a number that is at least `10` but not more than `9999`?

```
^\d{2,4}$
```

Since we have a minimum of two digits, `10` is the smallest match, whereas the maximum number of digits allowed is four, and hence, `9999` is the highest match.

What is the regex for an input that has seven digits and that can have `+` or `-` at the start?

```
^[+-]?\d{7}$
```

The `[+-]?` pattern makes it an optional match at the start before we match the seven digits using `\d{7}`.

> *The preceding regex can also be written as `^[+-]?[0-9]{7}$`, as `\d` is a shorthand property to match `[0-9]`*

Possessive quantifiers

Possessive quantifiers are quantifiers that are greedy when matching text like greedy quantifiers do. Both greedy and possessive quantifiers try to match as many characters as possible. The important difference, however, is that the possessive quantifiers do not backtrack (go back) unlike greedy quantifiers; therefore, it is possible that the regex match fails if the possessive quantifiers go too far.

This table shows all the three types of quantifiers, side by side:

Greedy Quantifier	Lazy Quantifier	Possessive Quantifier
m*	**m*?**	**m*+**
m+	**m+?**	**m++**
m?	**m??**	**m?+**
m{X}	**m{X}?**	**m{X}+**
m{X,}	**m{X,}?**	**m{X,}+**
m{X,Y}	**m{X,Y}?**	**m{X,Y}+**

Let's take an example input string, `a1b5`, and see the behavior of the greedy, lazy, and possessive quantifiers.

If we apply a regex using the greedy quantifier, `\w+\d`, then it will match `a1b` (the longest match before backtracking starts) using `\w+`, and `5` will be matched using `\d`; thus, the full match will be `a1b5`.

Now, if we apply a regex using the non-greedy quantifier, `\w+?\d`, then it will match `a` (the shortest match before expanding starts) using `\w+?`, and then the adjacent digit `1` will be matched using `\d`. Thus, the first full match will be `a1`. If we let the regex execute again, then it will find another match, `b5`.

Finally, if we apply a regex using the possessive quantifier, `\w++\d`, then it will match all the characters `a1b5` (the longest possible match without giving back) using `\w++` . Due to this, `\d` remains unmatched, and hence the regex fails to find any match.

Let's take another example. The requirement is to match a string that starts with lowercase English alphabets or hyphen. The string can have any character after the alphabets/hyphens, except a colon. There can be any number of any characters of any length after the colon until the end.

An example of a valid input is `as-df999` and that of an invalid input is `asdf-:123`.

Now, let's try solving this regex problem using a greedy quantifier regex:

```
^[a-z-]+[^:].*$
```

Unfortunately, this is not the right regex pattern because this regex will match both the aforementioned valid and invalid inputs. This is because of the backtracking behavior of the regex engine in greedy quantifiers. The `[a-z-]+` pattern will find the longest possible match in the form of `asdf-`, but due to the negated character class pattern `[^:]` , the regex engine will backtrack one position to `asdf` and will match the next *hyphen* for `[^:]`. All the remaining text, that is, `:123`, will be matched using `.*`.

Let's try to solve this regex problem using the following possessive quantifier regex:

```
^[a-z-]++[^:].*$
```

This regex pattern will still match our valid input, but it will fail to match an invalid input because there is no backtracking in possessive quantifiers; hence, the regex engine will not go back any position after matching `asdf-` in the second example string. Since the next character is a colon and our regex sub-pattern is `[^:]`, the regex engine will stop matching and correctly declare our invalid input a failed match.

Possessive quantifiers are good for the performance of the underlying regex engine because the engine does not have to keep any backtracking information in memory. The performance increase is even more when a regex fails to match because possessive quantifiers fail faster.

Another example of describing a pattern with a generalized RE

An e-mail address is
- A sequence of letters, followed by
- the character "@", followed by
- the character "." , followed by a nonempty sequence of lowercase letters, followed by
- [any number of occurrences of the previous pattern]
- "edu" or "com" (others omitted for brevity).

Q. Give a generalized RE for e-mail addresses.

A. `[a-z]+@([a-z]+\.)+(edu|com)`

Exercise. Extend to handle `rs123@princeton.edu`, more suffixes such as `.org`, and any other extensions you can think of.

Next. Determining whether a given string matches a given RE.

GREP: a solution to the RE pattern matching problem

"GREP" (Generalized Regular Expression Pattern matcher).

- Developed by Ken Thompson, who designed and implemented Unix.
- Indispensable programming tool for decades.
- Found in most development environments, including Java.

Practical difficulty: The DFA might have *exponentially* many states.

Interested in details? Take a course in algorithms.

A more efficient algorithm: use Nondeterministic Finite Automata (NFA)

- Build the NFA corresponding to the given RE.
- Simulate the operation of the NFA.

Ken Thompson
1983 Turing Award

Applications of REs

Pattern matching and beyond.

- Compile a Java program.
- Scan for virus signatures.
- Crawl and index the Web.
- Process natural language.
- Access information in digital libraries.
- Search-and-replace in a word processors.
- Process NCBI and other scientific data files.
- Filter text (spam, NetNanny, ads, Carnivore, malware).
- Validate data-entry fields (dates, email, URL, credit card).
- Search for markers in human genome using PROSITE patterns.
- Automatically create Java documentation from Javadoc comments.

GREP and related facilities are built in to Java, Unix shell, PERL, Python ...

virtually *every* computing environment