Day 3: Mull It Over

Der Tag 3 ist eigentlich insgesamt nicht schwer, allerdings sollte man sich dazu am besten bereits mit **regulären Ausdrücken** (Regex = ***reg****ular* ***ex****pressions*) auskennen. Durch diese Einschränkung kann man die Aufgabe zur mittleren Schwierigkeit zählen.

Reguläre Ausdrücke (Regex)

Im Wiki ist eine gute Einführung vorhanden. In Kurzform beschrieben benötigt man Regex im Alltag für **Pattern Matching**, also zur Mustererkennung. Häufig (auch in dieser Aufgabe), muss man innerhalb eines großen Textstückes einzelne Bestandteile suchen, die auf eine vorgegebene Art und Weise formatiert sind.

Nun bietet zwar Java Befehle wie `contains()`, mit der ich überprüfen kann, ob ein ganz bestimmter String in einem anderen String enthalten ist, aber das genügt für diese Aufgabe hier schon nicht mehr, da z. B. nicht nur `mul(3,4)`, sondern auch `mul(122,283)` und zahlreiche andere Zahlenkombinationen erkannt werden müssen.

Kurzum, der hier benötigte reguläre Ausdruck für Java lautet, in nicht-formatierter (= besser lesbarer) Form:

Regex zur Erkennung der "Operatoren"

```
"mul\(\d{1,3},\d{1,3}\)"
```

- Mit `mul` wird der exakte String `mul` erkannt.
- Direkt darauf **muss** eine öffnende Klammer folgen. Diese muss mit dem vorangestellten Backslash "escaped" werden, andernfalls würde sie von der Regex nicht als Klammer-Zeichen, sondern als "Rechenregel" interpretiert werden.
- Anschließend muss eine Ziffer (`\d`) folgen. Die direkt dahinter stehende geschweifte Klammer gibt an, dass 1-3 Ziffern hintereinander stehen können.
- Dann folgt ein Komma `,`
- Dann folgen erneut 1-3 Ziffern.
- Abschließend muss die Klammer geschlossen werden.

Diese Regex erlaubt es, alle String-Stücke im durchsuchten Text zu finden, die der beschriebenen Form genügen. Je nachdem, welchen Java-Befehl man nutzt, kann man sich den genauen String im durchsuchten Text anzeigen lassen, jedes Vorkommnis durch einen anderen String ersetzen, die Häufigkeit dieses Strings zählen, usw. Mit einer Regex kann man also Ausschnitte in einem längeren String finden, die bestimmten Regeln folgen.

Teil 1

Mit dieser Vorrede wage ich es, direkt die Lösung zu zeigen, da es zäh ist, die einzelnen speziellen Java-Befehle aufzuschlüsseln.

Lösungsvorschlag

```
public void partOne() {
    // Speichere, welche Form die mul-Operatoren haben
    // Bei Java müssen Sonderzeichen doppelt-escaped werden: \\
    Pattern pattern = Pattern.compile("mul\\(\\d{1,3},\\d{1,3}\\)");

    long result = 0;

    for (String line: inputLines) {
        // Wende das Muster (pattern) auf die aktuelle Zeile an
        Matcher matcher = pattern.matcher(line);

        // solange noch ein weitere Muster (mul-Operator) gefunden wird...
        while (matcher.find()) {
            // ... schnappe dir diesen nächsten mul-Operator und zerlege ihn
in die linke/rechte Zahl
            String mul = matcher.group();
            int num1 = Integer.parseInt(mul.split(",")[0].split("\\(")[1]);
            int num2 = Integer.parseInt(mul.split(",")[1].split("\\)")[0]);
            result +=  num1 * num2;
        }
    }

    System.out.println(result);
}
```

Teil 2

Für Teil 2 ist zunächst essenziell, dass man die komplette Datei als **einen zusammenhängenden String** betrachtet - **nicht** Zeile für Zeile!

Die Regel mit `don't()` und `do()` kann man wie folgt angehen. Man kann weiterhin denselben Code wie aus Teil 1 nutzen, allerdings darf man dies eben nicht auf den gesamten Text anwenden, sondern man muss bestimmte Teil im Text entfernen. Was muss entfernt werden? Entfernt werden muss alles **nach** einem `don't()` und bis zum ersten folgenden `do()`.

Wenn man dies auf einen Eingabe-String `eingabe` anwenden will, funktioniert das mit folgendem Befehl: `eingabe.replaceAll("HIER_STEHT_DIE_REGEX", "")`. Das zweite Argument gibt an, durch was der gefundene String ersetzt werden soll (=Empty String, entpricht also einer Löschung). Wie sieht die Regex aus? `don't\\(\\).*?do\\(\\)`

Hier ist bereits berücksichtigt, dass in Java doppelt escaped wird. Sehr knifflig ist der mittlere Abschnitt `.*?`. Die ersten zwei Zeichen sind in der Kombination häufig anzutreffen: Zwischen don't() und do() dürfen beliebige Zeichen (`.`) beliebig häufig (*) vorkommen. Allerdings möchte man ja nur so wenig löschen, wie es geht. In einem String wie `don't()aaa_do()bbb_do()` soll nicht erst beim letzten `do()` gestoppt werden, sondern bereits beim ersten. Reguläre Ausdrücke arbeiten aber

standardmäßig **greedy** (also gierig) und suchen nach langen Mustern. Um sie **lazy** (faul) zu machen, muss man zusätzlich ein Fragezeichen ? anhängen. Die 3 Zeichen könnte man also lesen als *so viele Zeichen wie **nötig***.

Lösungsvorschlag

```
public void partTwo() {
    // Speichere, welche Form die mul-Operatoren haben
    // Bei Java müssen Sonderzeichen doppelt-escaped werden: \\
    Pattern pattern = Pattern.compile("mul\\(\\d{1,3},\\d{1,3}\\)");

    long result = 0;

    String allLines = "";
    for (String line: inputLines) {
        allLines += line;
    }

    allLines = allLines.replaceAll("don't\\(\\).*?do\\(\\)", "");

    /* Eventuell muss man hier noch nachträglich prüfen, ob es zum Ende ein
     * don't() OHNE nachfolgendes do() gibt. Dann muss der komplette Rest
     * "abgeschnitten" werden. Bei meinem Input war es nicht nötig.
     */

    Matcher matcher = pattern.matcher(allLines);

    while (matcher.find()) {
        String mul = matcher.group();
        int num1 = Integer.parseInt(mul.split(",")[0].split("\\(")[1]);
        int num2 = Integer.parseInt(mul.split(",")[1].split("\\)")[0]);
        result +=  num1 * num2;
    }

    System.out.println(result);
}
```

From:
https://www.info-bw.de/ -

Permanent link:
https://www.info-bw.de/faecher:informatik:oberstufe:java:aoc:aoc2024:day03:start

Last update: **03.12.2024 15:53**