Wyrażenia Regularne

Julia Mikrut

Czym są wyrażenia regularne?

Wyrażenia regularne (ang. **Regular Expressions**, skrót **regex** lub **regexp**) to sekwencje znaków tworzące wzorce wyszukiwania, które są wykorzystywane do dopasowywania i manipulowania tekstem. Wyrażenia regularne są powszechnie używane w programowaniu, edytorach tekstu oraz narzędziach do przetwarzania danych, aby odnaleźć i zastąpić fragmenty tekstu zgodnie ze zdefiniowanym wzorcem.

Wyrażenia regularne mają swoje korzenie w latach 50. XX wieku, kiedy to amerykański matematyk Stephen Cole Kleene opracował koncepcję regularnych zestawów. Kleene wprowadził notację, która pozwalała na opisanie wzorców w językach formalnych.

W latach 60. i 70. XX wieku wyrażenia regularne zaczęły być adaptowane do informatyki. Ken Thompson wprowadził wyrażenia regularne do języka programowania QED. Później, zaimplementował je również w edytorze tekstu ed, który stał się częścią systemu operacyjnego Unix. Narzędzia Unix, takie jak grep (Global Regular Expression Print), sed (Stream Editor) i awk były używane do przetwarzania i analizowania tekstu, co uczyniło je niezastąpionymi narzędziami dla administratorów systemów i programistów.

W latach 80. i 90. wyrażenia regularne zostały zaimplementowane w wielu językach programowania, z których jednym z najważniejszych był Perl, który z zaawansowanym wsparciem dla wyrażeń regularnych, znacząco przyczynił się do ich popularności.

Obecnie wyrażenia regularne są szeroko stosowane w programowaniu, przetwarzaniu tekstu, walidacji danych i analizie logów. Większość nowoczesnych języków programowania, takich jak Python, JavaScript, Java, C#, PHP i wiele innych, wspiera wyrażenia regularne.

Podstawowa składnia

Proste wzorce składają się ze znaków, dla których chcesz znaleźć bezpośrednie dopasowanie. Na przykład wzorzec **/abc/** dopasowuje kombinacje znaków w łańcuchach tylko wtedy, gdy występuje dokładna sekwencja „**abc**” (wszystkie znaki razem i w tej kolejności). Takie dopasowanie odniosłoby sukces w ciągach „**abc**defg” oraz „xyz**abc**”. W obu przypadkach dopasowanie następuje z podłańcuchem „abc”. Nie ma dopasowania w ciągu „bca”, ponieważ chociaż zawiera on znaki ‚a’, ‚b’ i ‚c’ nie są one w odpowiedniej kolejności.

Metacharacters

Jeśli wyszukiwanie wymaga czegoś więcej niż bezpośredniego dopasowania, na przykład znalezienia jednej lub więcej litery ’b’ albo znalezienia białych znaków, we wzorcu można umieścić **znaki specjalne.**

Metaznaki to znaki w wyrażeniach regularnych, które mają specjalne znaczenie i funkcje. Są one używane do definiowania wzorców dopasowywania, które są bardziej zaawansowane niż proste literały

Klasy znaków

Klasy znaków to zestawy znaków umieszczone w nawiasach kwadratowych **[ ]**, które dopasowują dowolny znak z zestawu.

Przykłady użycia:

[abcd], [a-d]	Dopasowuje **a, b**, **c**, lub **d**.
[0-9]	Dopasowuje dowolną cyfrę od 1 do 9
[^abcd], [^a-d], [^0-9]	Negacja klas - dopasowuje do znaków **nie** ujętych w nawiasy kwadratowe.
[A-Ca-c]	Dopasowuje zarówno **A, B, C** jak i **a, b, c.**
[A-Za-z0-9]	Dopasowuje dowolną literę lub cyfrę.

Predefiniowane klasy znaków

- **\d** dla cyfr: [0-9]
- **\D** dla znaków, które nie są cyframi: [^0-9]
- **\w** dla dowolnego znaku słownego z podstawowego alfabetu łacińskiego: [A-Za-z0-9_]
- **\W** dla dowolnego znaku, który nie jest znakiem słownym podstawowego alfabetu łacińskiego. [^A-Za-z0-9_]
- **\s** dla białych znaków (Znaki spacji, tabulatora, kanału liniowego (nowy wiersz), powrotu karetki, kanału formularza i pionowe znaki tabulatora):

Kwantyfikatory

Kwantyfikatory określają liczbę wystąpień poprzedzającego elementu w wyrażeniu regularnym.

- ***** (gwiazdka): Pasuje do zero lub więcej wystąpień poprzedzającego elementu.
  Przykład: **ab*c** pasuje do **ac**, **abc**, **abbc**, **abbbc**
- **+** (plus): Pasuje do jednego lub więcej wystąpień poprzedzającego elementu.
  Przykład: **ab+c** pasuje do **abc**, **abbc**, **abbbc** (ale nie ac).
- **?** (pytajnik): Pasuje do zero lub jednego wystąpienia poprzedzającego elementu.
  Przykład: **ab?c** pasuje do **ac**, **abc**
- **{n}** : Pasuje do dokładnie n wystąpień danego elementu.
  Przykład: **a{4}** pasuje do **aaaa**
- **{n,}**: Pasuje do n lub więcej wystąpień danego elementu.
  Przykład: **a{2,}** pasuje do **aa**, **aaa**, **aaaaaa**.
- **{n,m}**: Pasuje do wystąpień znaku między n a m.
  Przykład: **a{2,4}** pasuje do **aa**, **aaa**, **aaaa**.

Kotwice

Kotwice to metaznaki, które dopasowują określoną pozycję w ciągu znaków, a nie same znaki.

- **^** Dopasowuje początek ciągu .
  Przykład: **^abc** pasuje do "**abc**def", ale nie do „defabc".
- **$** Dopasowuje koniec ciągu.
  Przykład: **$abc** pasuje do „xyz**abc**".
- **\b** Dopasowuje tylko do granic słowa.
  Przykład: **\ba** pasuje do "**a**bcabc"
  Przykład: **a\b** pasuje do "abcabc**a**"
- **\B** (niegranica słowa): Dopasowuje pozycje, która nie jest granicą słowa
  Przykład: **\Ba\B** pasuje do "abc**a**bca"
  Przykład: **a\B** pasuje do "**a**bca"

Grupy i alternacja

- **|** (alternacja): Pasuje do elementu po lewej lub prawej stronie.
  Przykład: **a|b** pasuje do **a** lub **b**.
- **()** (nawiasy): Grupuje elementy razem.
  Przykład: **(ab)** pasuje do ”**ab**cdef”.

Sekwencje ucieczki (escape sequences)

Sekwencje ucieczki w wyrażeniach regularnych to specjalne kombinacje znaków, które pozwalają na traktowanie metaznaków jako zwykłych znaków. Metaznaki, takie jak kropka (.), gwiazdka (*), plus (+) czy nawiasy, mają specjalne znaczenie w kontekście wyrażeń regularnych. Aby użyć ich dosłownie, trzeba poprzedzić je znakiem backslash ****.

Python

W Pythonie do pracy z wyrażeniami regularnymi używany jest **moduł re**, który oferuje różne funkcje do dopasowywania wzorców, wyszukiwania i zamiany tekstu. Oto kilka podstawowych funkcji:

- **re.match()**: Dopasowuje wzorzec do początku ciągu.
- **re.search()**: Przeszukuje cały ciąg w poszukiwaniu dopasowania wzorca.
- **re.findall()**: Znajduje wszystkie wystąpienia wzorca w ciągu.
- **re.sub()**: Zastępuje dopasowania wzorca innym ciągiem.

```python
import re

text = "Kontakt: email@example.com, telefon: +48123456789."
email_pattern = r"[\w\.-]+@[a-zA-Z\d\.-]+\.[a-zA-Z]{2,6}"
emails = re.findall(email_pattern, text)
print("Znalezione adresy e-mail:", emails)
```

Java

W Javie regex są zaimplementowane w klasie **Pattern** i **Matcher** z pakietu **java.util.regex**. Pattern reprezentuje skompilowany wzorzec wyrażenia regularnego, a Matcher obiekt, który dopasowuje wzorzec do danego ciągu.

- **Pattern.compile():** Kompiluje wyrażenie regularne do obiektu
- **Pattern.Matcher.find():** Szuka kolejnego dopasowania w ciągu.
- **Matcher.matches():** Sprawdza, czy cały ciąg pasuje do wzorca.

```java
import java.util.regex.*;

public class RegexExample {
    public static void main(String[] args) {
        String text = "Kontakt: email@example.com, telefon: +48123456789.";
        String emailPattern = "[\\w\\.-]+@[a-zA-Z\\d\\.-]+\\.[a-zA-Z]{2,6}";
        Pattern pattern = Pattern.compile(emailPattern);
        Matcher matcher = pattern.matcher(text);
        while (matcher.find()) {
            System.out.println("Znaleziony e-mail: " + matcher.group());
        }
    }
}
```

JavaScript

W JavaScript wyrażenia regularne są obiektami **RegExp**, które można tworzyć za pomocą literałów **regex** lub konstruktora **RegExp.** Metody obiektu RegExp oraz metody wbudowane w obiekt String umożliwiają wyszukiwanie, dopasowywanie i zamianę tekstu.

- **RegExp.test():** Sprawdza, czy wzorzec pasuje do ciągu.
- **RegExp.exec():** Wykonuje wyszukiwanie wzorca w ciągu i zwraca informacje o dopasowaniu.
- Metody obiektu String: **match(), replace(), search(), split().**

```
const text = "Kontakt: email@example.com, telefon: +48123456789.";
const emailPattern = /[\w\.-]+@[a-zA-Z\d\.-]+\.[a-zA-Z]{2,6}/g;
const emails = text.match(emailPattern);
console.log("Znalezione adresy e-mail:", emails);
```

PHP

W PHP regex są zaimplementowane w funkcjach **preg_*,** które oferują wszechstronne możliwości pracy z wyrażeniami regularnymi.

- **preg_match():** Sprawdza, czy wzorzec pasuje do ciągu.
- **preg_match_all():** Znajduje wszystkie dopasowania wzorca w ciągu.
- **preg_replace():** Zastępuje dopasowania wzorca innym ciągiem.

```php
$text = "Kontakt: email@example.com, telefon: +48123456789.";
$emailPattern = "/[\w\.-]+@[a-zA-Z\d\.-]+\.[a-zA-Z]{2,6}/";
preg_match_all($emailPattern, $text, $matches);
echo "Znalezione adresy e-mail: ";
print_r($matches[0]);
```

Problemy przy używaniu wyrażeń regularnych

Przekombinowane wzorce:
Tworzenie zbyt skomplikowanych wyrażeń regularnych może prowadzić do trudności w ich zrozumieniu, utrzymaniu i debugowaniu.Prosty wzorzec jest zazwyczaj bardziej efektywny i łatwiejszy do zarządzania.

- Zbyt skomplikowany wzorzec: ((a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)+|(\d+))
- Prostszy wzorzec: [a-z]+|\d+

Problemy wydajnościowe z złożonymi regex:
Złożone wyrażenia regularne mogą być bardzo kosztowne obliczeniowo, zwłaszcza gdy są stosowane do dużych zbiorów danych.Niekiedy złożone wzorce mogą prowadzić do problemów z wydajnością, znanych jako "catastrophic backtracking". Optymalizacja wyrażeń regularnych oraz unikanie nadmiernie złożonych wzorców jest kluczowa dla zapewnienia wydajności.

Zachłanność vs. Niechłanność
Wyrażenia regularne są domyślnie zachłanne (greedy), co oznacza, że próbują dopasować jak najwięcej tekstu.Niechłanne (non-greedy) dopasowania kończą dopasowywanie, gdy tylko znajdą pierwsze możliwe dopasowanie.

- Greedy: **.*** dopasowuje jak najwięcej tekstu.
- Non-greedy: **.*?** dopasowuje jak najmniej tekstu.

Lookaheads i Lookbehinds

Lookaheads: Służą do przewidywania wzorca, który musi następować po dopasowaniu, ale nie jest częścią dopasowania.

- Pozytywne lookahead: **(?=...)** - Wzorzec musi być obecny po dopasowaniu.

Przykład: **\d(?=px)** dopasowuje cyfrę, która jest bezpośrednio przed px.

- Negatywne lookahead: **(?!...)** - Wzorzec nie może być obecny po dopasowaniu.

Przykład: **\d(?!px)** dopasowuje cyfrę, która nie jest bezpośrednio przed px.

Lookbehinds: Służą do przewidywania wzorca, który musi występować przed dopasowaniem, ale nie jest częścią dopasowania.

- Pozytywne lookbehind: **(?<=...)** - Wzorzec musi być obecny przed dopasowaniem.

Przykład: (?<=\$)\d+ dopasowuje liczbę, która jest bezpośrednio po znaku $.

- Negatywne lookbehind**: (?<!...)** - Wzorzec nie może być obecny przed dopasowaniem.

Przykład: **(?<!\$)\d+** dopasowuje liczbę, która nie jest bezpośrednio po znaku $.

Źródła

- https://en.wikipedia.org/wiki/Regular_expression
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions
- https://regexr.com/
- https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html#zz-2.6