Technologia Programowania 2017/2018
Wykład 9
Wyrażenia regularne

Jakub Lemiesz

Wyrażenie regularne

1. Wyrażenie regularne (regular expression)
   to wzorce opisujące zbiór słów nad alfabetem i służące m.in. do przeszukiwania i przetwarzania tekstów, weryfikacji dopasowania

2. Regexp weszły do powszechnego zastosowania na początku lat siedemdziesiątych
   (*global regular expression print w Unix*)

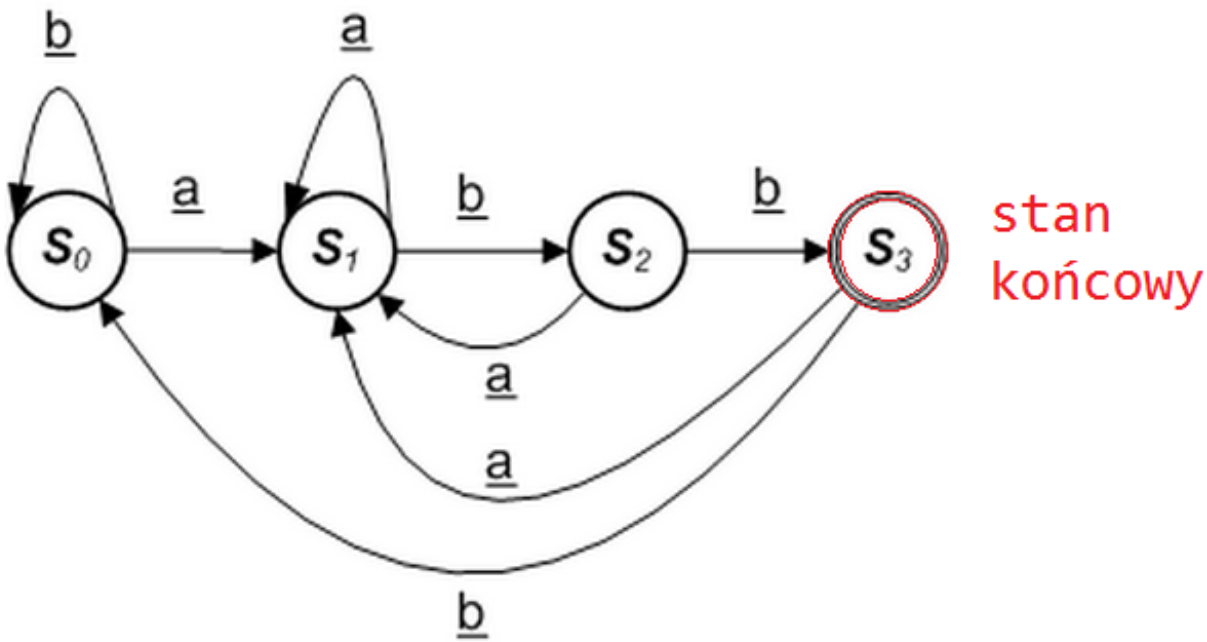
3. Regexp znalazły się następnie w bibliotekach praktycznie wszystkich języków programowania
   (*składnia jest z grubsza podobna*)

Wyrażenie regularne a języki formalne

- Regexp opisuje język: zbiór słów nad alfabetem
  (*np.* $L = \{\varepsilon, 1, 11, 111, 1111, ...\}$ *ma opis* 1^*)
- Czy każdy język można opisać przez regexp?
- Jedynie języki regularne, czyli takie, które można rozpoznać za pomocą automatu skończonego
  (*automat* $\approx$ *diagram stanów*)
- Język nieregularny to np. $L = \{a^n b^n : n \geqslant 0\}$
  (*automat musi „pamiętać" ile wystąpień 'a'*)

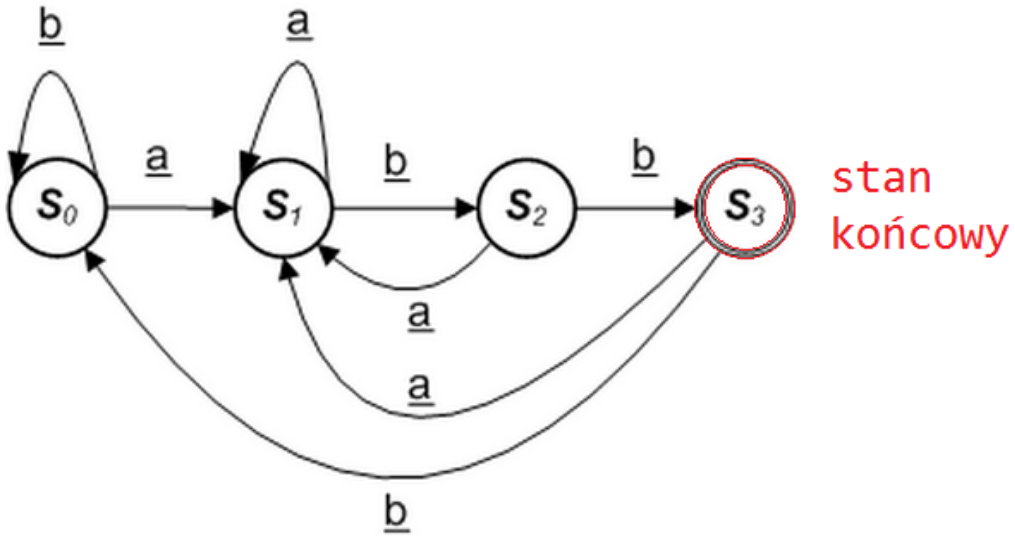
Automaty skończone — idea

1. Ustalmy alfabet $\Sigma = \{a, b\}$.
   Czy dowolny tekst T kończy się ciągiem abb?
2. Piszemy *regexp* = `[ab]*abb`
3. Konstruowany jest automat



Automaty skończone — idea

1. Ustalmy alfabet $\Sigma = \{a, b\}$.
   Czy dowolny tekst T zawiera ciąg abb?
2. Piszemy *regexp* = `[ab]*abb`
3. Konstruowany jest automat



Przykład: $T = aababbbabb$

Wyrażenia regularne w Java

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

String regex = "([ab]*)(abb)";
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher(myText);

Boolean b = m.matches();
String group1 = matcher.group(1);

m.reset();
while(m.find())
  println(m.start() + "-" + m.end());
```

Języki regularne mogą być opisane matematycznie

Wszystkie języki regularne nad alfabetem Σ

1. Język pusty $\emptyset$
2. Język $\{\alpha_i\}$ dla $\alpha_i \in \Sigma$
3. Jeśli A i B są regularne, to suma $A \cup B$, konkatenacja $A \cdot B = \{ab : a \in A \wedge b \in B\}$ oraz domknięcie Kleene'ego A^* są regularne

Domknięcie Kleene'ego zbioru

Najmniejszy zbiór zawierający słowo puste ε i domknięty względem operacji konkatenacji, np.:

$$\{ab, c\}^* = \{\varepsilon, ab, c, abab, abc, cab, cc, ababab, ...\}$$

Wyrażenia regularne w praktyce

	teoria	praktyka
dowolny znak	$\{a, b, c, ...\}$	.
zakres znaków	$\{c, d, e, f\}$	[a-f]
dowolna liczba wystąpień	$\{a\}^*$	a^*
0 lub 1 raz	$\{a\} \cup \{\varepsilon\}$	$a?$
1 lub więcej razy	$\{a\} \cdot \{a\}^*$	$a+$
$1 - 3$ razy	$\{a\} \cdot \{a, \varepsilon\} \cdot \{a, \varepsilon\}$	$a\{1, 3\}$

Wyrażenia regularne w praktyce są zapisywane za pomocą bogatszej i łatwiejszej w użyciu składni niż ta z rozważań teoretycznych

Składnia wyrażeń regularnych

Typowo wyrażenie regularne może się składać z

1. literałów,
2. klas znaków,
3. kwantyfikatorów,
4. granic,
5. flag
6. grup i odniesienia zwrotnych

Literały

1. Literały: ciągi znaków, np. "abc"
2. \ ^ $. | ? * + () [] {}
   mają specjalne znaczenie, by użyć ich jako literałów potrzebny znak wyjścia \

```
Wzorzec: "(x"
Tekst: "(x"
Exception: unclosed group near index 2
```

```
Wzorzec: "(x)"
Tekst: "(x)"
matches(): cały tekst nie pasuje do wzorca.
find(): dopasowano podłańcuch "x"
```

Literały

```
Wzorzec: "\(x"
Tekst: "(x"
matches(): Cały tekst pasuje do wzorca.
```

3. Inne zastosowania znaku wyjscia: $\backslash p, \backslash b, \backslash d, \ldots$ (*o tym za chwile*)
4. Java wymaga podwójnego backslash'a bo się nie skompiluje (*error: invalid escape sequance*), np. zamiast "2\+2" musimy napisać "2\\+2"
5. Wewnątrz definicji klas znaków [...] niepotrzebny znak wyjscia, np. $[*a-c]$ zamiast $[\backslash * a-c]$

Klasy znaków

1. "xxx|yyy|zzz" – logiczna alternatywa, wyszukiwany dowolny z "xxx","yyy","zzz"
2. Nawiasy kwadratowe [...] definiują klasy znaków
3. Np. [ab( ] jest równoważne wyrażeniu a|b|\(czyli oznacza jedną z wymienionych liter
4. Negacja klasy – jeśli pierwszym znakiem w klasie jest ^ dopasowanie nastąpi dla każdego znaku poza wymienionymi, [^*abc*] to każdy poza a, b i c

Klasy znaków

1. Zakresy wewnątrz klasy definiujemy przez znak –
2. Np.
   $[0-9]$ – dowolna cyfra,
   $[a-zA-Z]$ – mała lub duża litera,
   $[\hat{}0-2a-z]$ – ?
3. Kolejność znaków w klasie nie jest istotna ale zakresy muszą być w porządku rosnącym
4. Cześć wspólna i różnica zbiorów:
   [1–9&&3–5] i [1–9&&[^345]]

Klasy znaków

Przykładowe zadanie na kolokiwum

Za pomoca regexp opisz jezyk nad alfabetem złozonym z cyfr oraz małych liter alfabetu bez k,l,m, zawierajacy jedynie słowa w których wystepuja co najmniej trzy cyfry 7.

Klasy predefiniowane

- . dowolny znak (domyślnie bez znaku końca wiersza)
- $\backslash d = [0-9]$, $\backslash D = [\hat{}\backslash d]$
- $\backslash s = [\backslash t\backslash n\backslash x0B\backslash f\backslash r]$, $\backslash S = [\hat{}\backslash S]$
- $\backslash w = [a-zA-Z0-9]$, $\backslash W = [\hat{}\backslash w]$
- $\backslash p\{Lower\} = [a-z]$, $\backslash p\{Upper\} = [A-Z]$
- $\backslash p\{Punct\} = !"\#\$\%\&'()*+,-./:;<=>?@[\backslash]\hat{}\_\{|\}\sim$
- $\backslash p\{ASCII\}$ - znak ASCII
- $\backslash p\{InNazwaBlokuUnicode\}$ - np. polskie znaki są w $\backslash p\{InLatinExtended\text{–}A\}$

Kwantyfikatory

Do określania liczby powtórzeń literałów, elementów klasy lub "podwyrażenia" służą kwantyfikatory:

- ? – jeden raz lub wcale
- $*$ – zero lub więcej razy
- $+$ – raz lub więcej razy
- $\{n\}$ – dokładnie n razy
- $\{n,\}$ – co najmniej n razy
- $\{n, m\}$ – co najmniej n ale nie więcej niż m razy

Przykład

abc+ [abc]+ (abc)+ abc$\{2, 3\}$

Który z wzorców pasuje do tekstu "abcc"?

Tryby działania kwantyfikatorów

1. Domyślnie kwantyfikator jest zachłanny(greedy) — sprawdza dopasowanie do całego tekstu, jeśli nie ma, usuwa kolejne znaki z końca tekstu
2. Kwantyfikator można uczynić wstrzemięźliwym (reluctant) lub zaborczym (possesive) poprzez dopisanie po kwantyfikatorze znaku ? lub +
3. Wstrzemięźliwy — rozpoczyna od początku tekstu i pobiera znak po znaku aż dopasuje
4. Zaborczy — sprawdza zgodność wyrażenia z całym tekstem (wszystko albo nic)

Tryby działania kwantyfikatorów

Tekst: ”yyyy XXX yyy XXX”

- Wzorzec: .*XXX
  matches():
  find():

- Wzorzec: .*?XXX
  matches():
  find():

- Wzorzec: .*+XXX
  matches():
  find():

Tryby działania kwantyfikatorów

Tekst: "yyyy XXX yyy XXX"

- Wzorzec: .*XXX
  matches(): cały tekst pasuje do wzorca
  find(): "yyyy XXX yyy XXX" od pozycji 0

- Wzorzec: .*?XXX
  matches():
  find():

- Wzorzec: .*+XXX
  matches():
  find():

Tryby działania kwantyfikatorów

Tekst: "yyyy XXX yyy XXX"

- Wzorzec: .*XXX
  matches(): cały tekst pasuje do wzorca
  find(): "yyyy XXX yyy XXX" od pozycji 0

- Wzorzec: .*?XXX
  matches(): cały tekst pasuje do wzorca
  find(): dopasowano "yyyy XXX" od pozycji 0
  find(): dopasowano " yyy XXX" od pozycji 8

- Wzorzec: .*+XXX
  matches():
  find():

Tryby działania kwantyfikatorów

Tekst: "yyyy XXX yyy XXX"

- Wzorzec: .*XXX
  matches(): cały tekst pasuje do wzorca
  find(): "yyyy XXX yyy XXX" od pozycji 0

- Wzorzec: .*?XXX
  matches(): cały tekst pasuje do wzorca
  find(): dopasowano "yyyy XXX" od pozycji 0
  find(): dopasowano " yyy XXX" od pozycji 8

- Wzorzec: .*+XXX
  matches(): cały tekst NIE pasuje do wzorca
  find(): nie znaleziono żadnego wystąpienia wzorca

Kwantyfikatory — przykłady

Przykład 1

Napisz wyrażenie, które pozwoli Ci wyodrębnić z tekstu ostatnie trzy następujące po sobie cyfry.

$.* (\backslash d \backslash d \backslash d).*$

Przykład 2

Napisz wyrażenie, które pozwoli Ci uzyskać ostatnią liczbę całkowitą w tekście, niezależnie do tego ile ma cyfr.

$.*?(\backslash d+)\backslash D*$

Grupowanie

1. Kolejność wykonywania działań:
   \, (), kwantyfikatory, konkatenacja, alternatywa
2. Dodatkowo użycie nawiasów okrągłych pozwala odwołać się do części tekstu (grupy)

Przykład

```
Regex ((a)((b)(c)))(d) ma 6 grup o numerach
1 - grupa odpowiadająca wyrażeniu (a)((b)(c))
2 - grupa odpowiadająca wyrażeniu a
3 - grupa odpowiadająca wyrażeniu  (b)(c)
4 - grupa odpowiadająca wyrażeniu  b
5 - grupa odpowiadająca wyrażeniu c
6 - grupa odpowiadająca wyrażeniu  d
```

Grupowanie w Java

```
regex = "([0-9]{2})/([0-9]{2})/([0-9]{4})"
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher(dateString);

if (m.matches()) {
   int day   = Integer.parseInt(m.group(1));
   int month = Integer.parseInt(m.group(2));
   int year  = Integer.parseInt(m.group(3));
}
```

Grupowanie i odniesienia do grup (backreference)

1. Niektóre biblioteki umożliwiają odniesienie do zawartości grup w samym wyrażeniu regularnym
2. Wprowadzamy je za pomoca znaku wyjscia z następującym numerem grupy (od 1 do 9)
3. W tym miejscu wzorca powinien wystąpić dokładnie taki sam ciąg znaków, jaki został zapamiętany w grupie o podanym numerze

Przykład (szukanie powtórzen)

$$(\backslash b\ \backslash w+)\ [\backslash s\backslash p\{Punct\}]+\backslash 1$$

Kotwice

Znak	Znaczenie
$\hat{}$	początek linii
$	koniec linii
$\backslash b$	na granicy słowa
$\backslash B$	nie na granicy słowa
$\backslash A$	początek wejścia
$\backslash G$	koniec poprzedniego dopasowania
$\backslash Z$	koniec wejścia

Znowu znak "$\hat{}$" ?

Kotwice — przyklady

Tekst: ”yyy 123”

Wzorzec: $\backslash d+.*$
find(): dopasowano ”123” od pozycji 4

Wzorzec: $\hat{}\ \backslash d+.*$
find(): nie znaleziono żadnego wystąpienia wzorca

Tekst: ”XXX yyyyXXXyyyy yyyXXX”

Wzorzec: XXX find():

Wzorzec: $\backslash bXXX\backslash b$ find():

Wzorzec: $\backslash BXXX\backslash B$ find():

Jak znaleźć ”XXX” na końcu słowa?

Kotwice — przyklady

Wzorzec: *\Gabc*

Tekst: "abc abc abc"
find(): dopasowano "abc" od pozycji 0

Tekst: "abcabcabc"
find(): dopasowano "abc" od pozycji 0
find(): dopasowano "abc" od pozycji 3
find(): dopasowano "abc" od pozycji 6

Flagi

Interpretacji regexp można zmienić przez flagi, np.

(?i) – ignoruje wielkości liter

(?i)te(?-i)st pasuje do "TEst", ale nie do "teST"

(?u) – ignorując wielkości uwzględnia Unicode

Wzorzec: "(?i)(?u)mądrala"
Tekst: "MĄDRALA"
matches(): cały tekst pasuje do wzorca

Czytanie i tworzenie regexp bywa czasem nietrywialne

Prosty przykład. Co robi ten kod?

```
String text = ...
String pattern = "(\\w)(\\s+)([.,])";
text.replaceAll(pattern,"$1$3");
```

Przykład: usuwanie komentarzy blokowych /* ... */

```
(/\\*([^*]|(\\*+[^*/]))*\\*+/)|(//.*)
```