

# Регулярные выражения

Пугачёв Константин (K.V.Pugachev@inp.nsk.su)

2020-09-28 18h

# Грамматики

- Регулярные
  - наборы символов, повторения, без рекурсии
  - простые наборы символов вроде номеров телефонов
  - лексеры для ЯПов
- Контекстно-свободные
  - иерархическое описание сущностей, рекурсия
  - парсеры для ЯПов
- Контекстно-зависимые грамматики и Неограниченные грамматики
  - бедный парсер
  - реальные ЯПы
  - естественные языки

# Философия парсинга

- Распарсить до самой сути

'Всем привет' -> ['В', 'с', 'е', 'м', ' ', 'п', 'р', 'и', 'в', 'е', 'т'],  
плохо

'Всем привет' -> сообщение(  
    текст=приветствие('привет'),  
    адресат=любойадресат('всем')  
) – лучше

- Распарсить только

- только регулярными выражениями
- только \*\*\*ным автоматом
- только парсером КС грамматик
- только только

- Распарсить любой текст, удовлетворяющей грамматике
- Распарсить текст целиком

# Реальность парсинга

- Распарсить до какого-то момента, а там пусть разбираются

```
A B(C); // что это в C++?  
int func(int); // объявление функции?  
int val(5); // определение переменной?
```

- Распарсить набором средств в рамках программы
  - регулярными выражениями в цикле
  - парсером КС грамматик и прогнать результат через нейросеть
- Распарсить конкретный набор текстов, который не использует все фишки грамматики
- Распарсить нужный кусок текста

# HTML и регулярки

- Теория в руках теоретика
  - Грамматика HTML не является регулярной
  - Нельзя распарсить произвольный документ HTML целиком с помощью только одного регулярного выражения
- Теория в руках практика
  - Существует подмножество документов HTML, которые можно распарсить целиком или частично с помощью одного РВ
  - Программа под машину Тьюринга может содержать в себе использование РВ и как комплекс может в теории парсить всё, в т.ч. HTML
  - Если код с регуляркой парсит нужный HTML, прост, понятен и без багов - почему бы его не написать?

# Регулярные выражения (пример)

`/[0-9]{2}:[0-9]{2}/`

<code>00:00</code>	– OK
<code>00:01</code>	– OK
<code>0:00</code>	– FAIL
<code>a0:00</code>	– FAIL
<code>time is 10:30</code>	– OK
<code>10:20 is too early</code>	– OK

# Регулярные выражения (кирпичики)

<code>^</code>	начало строки
<code>\$</code>	конец строки
<code>.</code>	любой символ (кроме <code>\r</code> , <code>\n</code> )
<code>\.</code>	<code>.</code>
<code>\?</code>	<code>?</code>
<code>\r</code>	возврат каретки
<code>\n</code>	новая строка
<code>\d</code>	0, 1, 2, ... или ещё какие цифры
<code>\b</code>	граница слова
<code>\s</code>	пробельный символ (пробел, <code>\t</code> , <code>\r</code> , <code>\n</code> )
<code>\S</code>	непробельный символ
<code>a</code>	<code>a</code>
<code>[ab]</code>	<code>a</code> или <code>b</code>
<code>[^ab]</code>	не <code>a</code> и не <code>b</code>
<code>[a-d]</code>	<code>a</code> , <code>b</code> , <code>c</code> или <code>d</code>
<code>[^a-de-gz]</code>	всё, кроме <code>a,b,c,d,e,f,g,z</code>

# Регулярные выражения (комбинации)

a	a
ab	ab
ab+	ab или abb или abbb или abbbb, ...
ab*	a или ab или abb или abbb, ...
ab?	a или ab
ab{2,4}	abb или abbb или abbbb
a b	a или b
ab bc	ab или bc
(?:ab)+c	abc, ababc, abababc ...
x(?:ab bc)*	x, xab, xbc, xabbc, xabab, ...



# Группы

$([0-9]\{2\}):([0-9]\{2\})$

10:30 -> группа0 = 10:30  
          группа1 = 10  
          группа2 = 30

$([0-9]\{2\})((:[0-9]\{2\})+)$

12:34:56 -> группа0 = 12:34:56  
             группа1 = 12  
             группа2 = :34:56  
             группа3 = :56

# Жадность

`1.*34` -- жадный вариант (аналогично `1.+34`)

`12343333433`  $\rightarrow$  группа0 = `123433334`

`1.*?34` (аналогично `1.+?34`)

`12343333433`  $\rightarrow$  группа0 = `1234`

# Флаги

- /hello/ - РВ без флагов
- /hello/i - ignore case (hello = hEllo)
- /hello/m - multiline (иной смысл ^, \$)
- /hello/g - global (hellobyehello - найдётся всё\*)
- /hello/y - (hellohellohello - найдётся всё\*)
- /hello/ig - несколько флагов

# Средства JS

- класс RegExp
  - `new RegExp('hello', 'ig')`
  - литералы: `/hello/ig`
  - `exec: re.exec(str) -> null | [g0, g1, g2, ...]`
  - `test: re.test(str) -> true | false`
- методы String
  - `match: '1234'.match(/\d(\d)(\d)/) // ['123', '2', '3']`
  - `replace: '1234'.replace(/(\d)\d/, '[$1]') // '[1][3]'`
  - `replace: '1234'.replace(/(\d)\d/, (g0, g1) => g1+'1') // '1!3!'`
  - `search: '1234'.search(/2\d/) // 1`
  - `split: '1334'.split(/3/) // ['1', '', '4']`
  - `split: '1334'.split(/(3)/) // ['1', '3', '', '3', '4']`

# Состояние в регулярках - внимание!

Поле `lastIndex` - индекс следующего поиска (для g/y)

```
/a./g.exec('a1a2') // ['a1'], OK  
/a./g.exec('a1a2') // ['a1'], OK
```

```
let r1 = /a./;  
r1.exec('a1a2') // ['a1'], OK  
r1.exec('a1a2') // ['a1'], OK
```

```
let r2 = /a./g;  
r2.exec('a1a2') // ['a1'], OK  
r2.lastIndex    // 2  
r2.exec('a1a2') // ['a2'], OMG!  
r2.lastIndex    // 4
```

## Fun fact

В C++ (`boost::regex`) и PHP (стандартная библиотека) можно создавать регулярные выражения со ссылками (рекурсивными, если надо) на подвыражения:

```
boost::regex re(
    "(?(DEFINE)"
    "  "(?<num>[1-9]\\d*)"
    "  "(?<id>[xy])"
    "  "(?<val>\\((?&expr)\\)|(?&num)|(?&id))"
    "  "(?<expr>(?(?&prod)(\\+(?&prod))+|(?&prod)))"
    "  "(?<prod>(?(?&val)(\\*(?&val))+|(?&val)))"
    ")"
    "^(?&expr)$", boost::regex::perl);
```

Так можно создавать парсеры контекстно-свободных грамматик с помощью “регулярок”. И парсить XML назло хейтерам.