Regular Expressions
Statistics 650/750
Week 10 Thursday

Christopher Genovese and Alex Reinhart

Thu 01 Nov 2018

Regular Expressions

Pattern Matching, Parsers, and Languages

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. – Jamie Zawinski, 1997

A regular expression (regex) is a pattern that specifies a family of **matching** strings. In theoretical computer science, a set of strings is called a **language**, and a language that can be described exactly by a regular expression is called a **regular language**.

`'^f(o|a)+ *[,;:]  (\w+\s)+\$'`

The pattern is expressed with a combination of: characters, that stand for themselves, meta-characters and control constructs, that stand for patterns with certain properties or that modify others, and escaped meta-characters, which are meta-characters with a modifier (backslash) that tells us to treat them as a character.

Three important types of control constructs are zero-length assertions, which match if the assertion is true at that point in the pattern, character classes, which match any in a set of characters, and groups, which aggregate parts of a pattern for modifiers or later reference.

Note: Python raw strings `r'asdfasdf...\n'`

Examples (using PCRE, see below)

- '`foo`' – matches any string containing foo
- '`foo?`' – matches fo or foo
- '`fo.`' – matches fo followed by any non-newline character.
- '`f(oo)*`' – matches f followed by 2n oo's for n >= 0.
- '`^foo *$`' – matches foo at the beginning of a line (or string), followed by zero or more spaces
- '`^f[aeiou]+`' – matches f at the beginning of a line (or string), followed by one or more vowels
- '`(foo|bar)`' – matches foo or bar

A regular language

```
L = { a^nb^10 : n >= 0 }         'a*b{10}'
L = { a^n b^n : 0 <= n <= 5 }   '(|ab|aabb|....|aaaaabbbbb)'
```

A non-regular language:

```
L = { a^n b^n : n >= 0 }
```

where aˆn means n concatenated copies of a symbol a, which matches

```
<empty>
ab
aabb
aaabbb
aaaabbbb
...
```

Regex Engines

- Perl-Compatible Regular Expressions (PCRE)

  Built on the regular expression constructs from the language Perl, this has a rich set of constructs and extra functionality (lazy regexes, backtracking control, named capture groups, recursive patterns, etc.). It has become the de facto standard.

- POSIX-compliant Regular Expressions (basic and extended)

  The IEEE standard for regexes; commonly used in commands like grep. A subclass of PCRE with some differences in the underlying engine and functionality.

- Shell globs

  `ls foo*`

- Plus many variants in extensions, editors, etc.

The Common Metacharacters

1. '`.`' – matches any (non-newline) character
2. '`?`' – matches zero or one of the preceding character or construct
3. '`*`' – matches zero or more of the preceding character or construct
4. '+' – matches one or more of the preceding character or construct
5. '`|`' – matches either the construct on its left or on its right (alternation, or)
6. '`[...]`' – character class, matches any of the characters within it.
7. '`[^...]`' – complementary character class, matches any character not within it
8. '`{m,n}`' – matches from m to n (inclusive) copies of previous construct
9. '`^`' or '`\A`' – matches at start of line or string
10. '`$`' or '`\z`' – matches at end of line or string
11. '()' – grouping pattern within, can be modified or referenced captured group by default, uncaptured group with '`(?:...)`' Some engines (e.g., python) support named groups like `(?foo:<pattern>)` –> refer to `\P{foo}` later

```
From:\s*([-A-Za-z@.0-9_]+)     From:  my\1
From:\s*(?:[-A-Za-z@.0-9_]+)   no capturing
```

12. '\1' or other number – refers to a captured group

13. Built in classes: `\d` and `\D`, `\s` and `\S`, `\w` and `\W`, `\b`

There are many, many more features, all with their own "codes." This makes regex's somewhat obscure and hard to read, but they are powerful.

Exercise: Find a regular expression that matches a comma-separated list of words in parentheses. (Similar: match a double-quoted string that contains no other quotes. What to do with extra quotes?)

```
Naive:    ".*"    Doesn't work -- greedy match

Solution 1:   "[^"]*"
Solution 2:   ".*?"


"asfadf\"asdfadf"
"asfadf\\"
"asfadf\\\\\\\"
```

Greediness and laziness

The exercise above illustrates a key point: Regular expression matching is greedy – it will match as many times as possible while still allowing the rest of the pattern to match.

In PCRE and it variants, you can specify lazy matchers with '`*?`' '`??`', '`+?`', and '`{}?`' in place of '`*`', '`?`', '`+`', and '`{}`'.

Modifiers to the match

- Case-insensitivity `"(?i)...."` `"/.../i"`
- Multi-line
- Global search
- Commented regex
- ...

Compiling

If you have a complicated pattern that is used many, many times, most languages will let you "compile" it to speed up matching.

This converts the regular expression into an optimized form that can be matched quickly.

For example, in Python, the `re.compile()` function in the `re` library will return a compiled pattern that can be used in place of a string pattern anywhere.

Language Support

Almost every modern programming language has support, either built in or as a library, for matching regular expressions. Most support the PCRE engine, or some extension thereof.

Examples:

- Python – the `re` library
  - `re.search()`, `re.match()` – Search for a pattern; `match` at begining only
  - `re.split()` – split a string at a pattern
  - `re.findall()` – find all occurrences of pattern
  - `re.sub()` – search and replace on a pattern
  - ...

  Use Python's raw strings for representing regex's.

```python
m = re.search(r'warning \w+: (\d+)\.(\d*)', 'warning code: 100.32')
if m:
    print('Code: Chapter {}, Section {}'.format(m.group(1), m.group(2)))
#=> Code: Chapter 100, Section 32
```

- R – the `stringr` package

  Includes functions to manipulate, convert, and search strings.
  - `str_detect()` – Is the pattern present in string(s)?
  - `str_count()` – Count the number of matches
  - `str_locate()` – Locate first position of match
  - `str_locate_all()` – Locate all positions of match
  - `str_extract()` – extract text of first match (for each string)
  - `str_extract_all()` – extract text of all matches (for each string)
  - `str_match()` – extract capture groups from first match
  - `str_match_all()` – extract capture groups from all matches
  - `str_replace()` – replace first match in strings
  - `str_replace_all()` – replace all matches in strings
  - ...

```r
strings <- c(
  "apple",
  "219 733 8965",
  "329-293-8753",
  "Work: 579-499-7527; Home: 543.355.3679"
)
phone <- "([2-9][0-9]{2})[- .]([0-9]{3})[- .]([0-9]{4})"

str_match(strings, phone)
```

```
#=>      [,1]           [,2]  [,3]  [,4]
#=> [1,] NA             NA    NA    NA
#=> [2,] "219 733 8965" "219" "733" "8965"
#=> [3,] "329-293-8753" "329" "293" "8753"
#=> [4,] "579-499-7527" "579" "499" "7527"
```

Examples

We will use `https://regexr.com/` to examine and analyze a variety of examples.

- A literal string: '`$200`'
- Two words separated by commas and optional spaces: e.g., '`battle, turtle`'
- The previous case but capture the words
- An arbitrary length list of words separated by commas and optional spaces
- The previous case but capture the words
- Phone numbers: e.g., '`412-555-1234`'
- Phone numbers, capturing the area code and the rest of the number
- LaTeX text of the form '`\notable{\textbf{stuff here}}`', capturing the stuff
- A double-quoted string with no quotes in it, capturing the string (without quotes). But: the string may be surrounded with other text.
- A line beginning with optional spaces, followed by '`#+`' and one of several words `TITLE`, `SUBTITLE`, `DATE`, `AUTHOR`, or `OPTIONS`.
- A literal floating-point number in R or Python