Searching for Patterns with Regular Expressions

Michael Wayne Goodman goodmami@uw.edu
Nanyang Technological University, Singapore

2019-10-18

Presentation agenda

Introduction

For a class I teach, I asked students to provide interesting examples of netspeak, such as *b4* meaning *before*. Many of them offered laughter sounds in many languages:

Thai	55
Spanish	jeje
Japanese	ww; 笑笑
Chinese	哈哈; 呵呵
Korean	keke; kk

Q: If I want to parse webcrawl data for laughter, how can I match all of these? Searching for each individually takes too long.

Introduction

First I'll define a grammar:

```
Start  := "5" Tha
        | "ha" Eng
        | "je" Spa
        | "w" Jp1
        | "笑" Jp2
        | "哈" Ch1
        | "呵" Ch2
        | "ke" Ko1
        | "k" Ko2
Tha    := "5" Tha | "5"
Eng    := "ha" Eng | "ha"
Spa    := "je" Spa | "je"
Jp1    := "w" Jp1 | "w"
Jp2    := "笑" Jp2 | "笑"
Ch1    := "哈" Ch1 | "哈"
Ch2    := "呵" Ch2 | "呵"
Ko1    := "ke" Ko1 | "ke"
Ko2    := "k" Ko2 | "k"
```

I could parse it using Python:

```python
def match_laughter(s):
    i = 0
    if s.startswith('55'):
        i = match_thai(s, 2)
    elif s.startswith('haha'):
        i = match_english(s, 4)
    elif ...
    # etc...
    if i > 0:
        return s[:i]
    else:
        return None


def match_thai(s, i):
    if s[i] == '5':
        i = match_thai(s, i+1)
    return i
#etc...
```

Introduction

Or I could write my grammar as a regular expression:

```
55+|ha(ha)+|je(je)+|ww+|笑笑+|哈哈+|呵呵+|ke(ke)+|kk+
```

Regex to the Rescue

https://xkcd.com/208/

Problems

But regular expressions are a skill to learn and take time to master, leading to (slightly demotivating) quotes like the following:

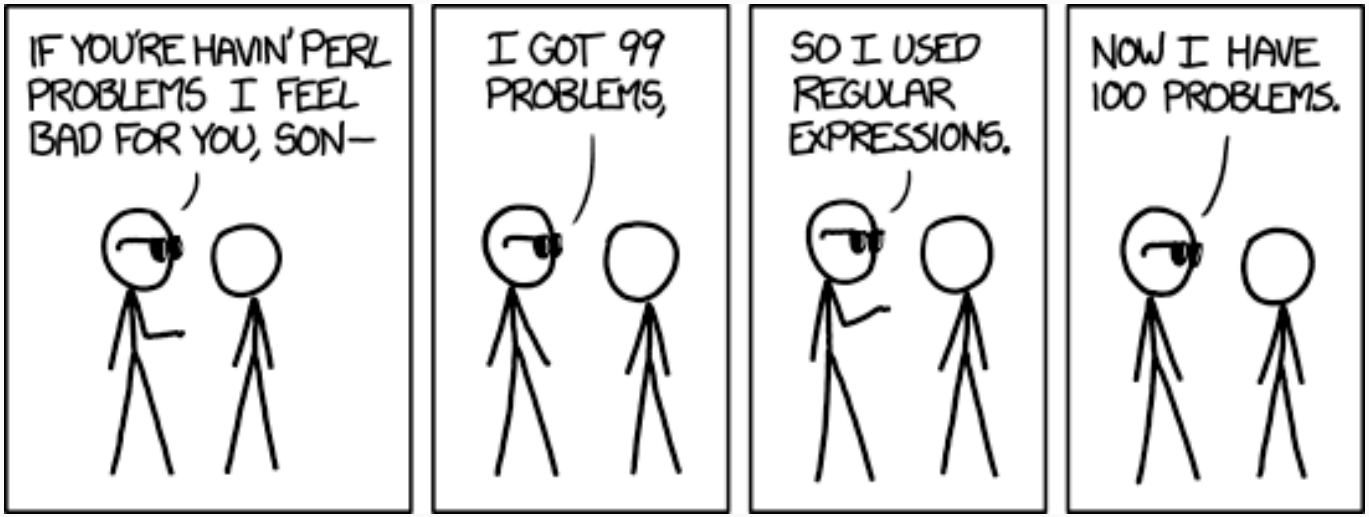
> **On 12 August, 1997, Jamie Zawinski said:**[1]
>
> *Some people, when confronted with a problem, think*
> *"I know, I'll use regular expressions."*
> *Now they have two problems.*

[1] Paraphrasing D. Tilbrook; Source:
`http://regex.info/blog/2006-09-15/247`

99 Problems

... which is often referenced, repeated, and recycled.

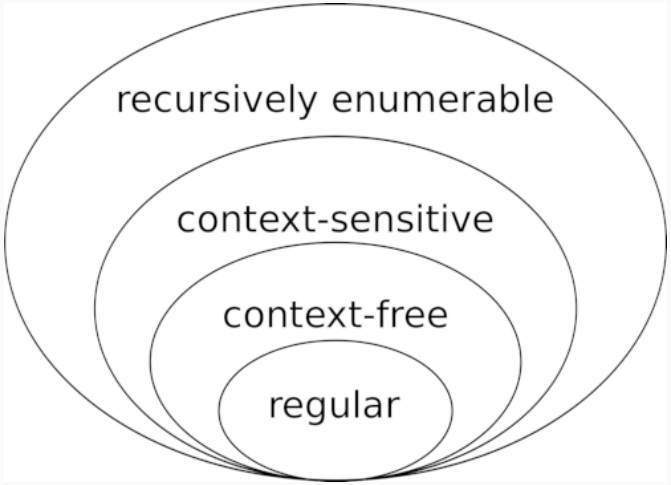
For example:



https://xkcd.com/1171/

Regular Expressions: What are they?

Regular expressions are a mini-language that compactly encode grammars for matching strings. They came out of the theoretical idea of regular grammars, which are the simplest kind of grammar in the Chomsky Hierarchy.



Modern regular expression engines, however, allow for non-regular features as well, such as lookahead and back-references.

Regular Expressions: What are they good for?

Regular expressions are great at finding matches that go beyond literal matches. For example, finding something that repeats, spelling alternations, flexible word collocations, optional matches, etc.

Regular Expressions: What are they not good for?

But regular expressions still have their limits. They are still mostly unable to do context-sensitive matching. For instance, you cannot use them to parse HTML data.

It’s all fun and games…

Solving a regular expression can be like solving a puzzle. It’s fun! Some go as far as making it a game:

- `https://regexcrossword.com/`
- `https://alf.nu/RegexGolf`

`https://xkcd.com/1313/`

Presentation agenda

Crafting Regular Expressions

Now we will cover a number of regular expression features.

For this part, I recommend having a regular expression tool open, such as:

`https://regex101.com/`

Basic Patterns

Sequences, Choices, and Greedy Matching

- Sequential sub-patterns match sequentially
- Choices, or alternations, delimited with |
- Matches are greedy: they consume as much as possible

```
Pattern   : abc|cba

Input     : abcba
Match     : abc
Remainder:    ba  <-- does not match cba

Input     : cbabc
Match     : cba
Remainder:    bc  <-- does not match abc
```

Repetition

Characters and subpatterns can be repeated via several mechanisms. The most basic are `*` and `+` (Kleene star/plus[2]) and `?`, but finer control is possible:

- `a*` : match "a" zero or more times
- `a+` : match "a" one or more times
- `a?` : match "a" zero or one time (optionality)
- `a{3}` : match "a" 3 times exactly
- `a{3,5}` : match "a" between 3 and 5 times
- `a{3,}‘` : match "a" 3 or more times
- `a{,5}` : match "a" 5 or fewer times

[2] `https://en.wikipedia.org/wiki/Kleene_star`

Anchors

Anchors are used to match only in certain contexts:

- `^` : match from the beginning of the string
- `$` : match to the end of the string
- `\b` : match word boundaries

Dot

The dot character (`.`) is a special character that matches any single character in the input. This is often useful for getting context. For example, the following matches up to 20 characters before and after the word *China*:

```
.{,20} China .{,20}
```

Flexible Patterns

Character Classes

Character classes, or character sets, match one of a set of characters. They are specified in brackets `[ ]`, hyphens (`-`) denote a range, and a caret (`^`) at the beginning inverts the set.

- `[abc]` : match `a`, `b`, or `c`
- `[a-z]` : match `a`, `b`, ..., or `z`
- `[^abc]` : match anything that is not `a`, `b`, or `c`

Escapes

Now we've seen some characters that regex treats specially (we'll get to the last two in a minute):

```
| * + { } [ ] ^ \$ \ . ( )
```

But if you want to match these literal characters, you must escape them with \.

```
\| \* \+ \{ \} \[ \] \^ \$ \\ \. \( \)
```

Special Escapes

Escapes are not only used to match special characters literally, but also to match literal characters specially. We've already seen one, `\b` for matching word boundaries. Some others are:

- `\w` : match a word character
- `\d` : match a digit character
- `\s` : match a whitespace character

These have negated forms, as well:

- `\W` : match a non-word character
- `\D` : match a non-digit character
- `\S` : match a non-whitespace character

Matching with Groups

Groups

Parentheses (`( )`) are used for groups, which have several uses:

- they let you create alternations in a local context
- they let you specify repetitions of subpatterns
- they can be used for back references (backslash number, like `\1` for the first group, etc.)

Example:

```
(they|he|she) did(n't| not)
```

Matches *they didn't*, *he did not*, etc.

Groups

More examples:

```
\w+(, \w+)*,? and \w+
```

Matches *apples and bananas*; *Singapore, Malaysia, Brunei, and Indonesia*; etc.

```
(\w)\1
```

Matches single-character repetition, as in the *o* of *foot*, or 人人, 謝謝, etc.

Repeated Groups

The groups we've seen are called capturing groups because the matched text is captured for use in back-references, etc.

When a group is repeated, only the last match is captured. Consider if you want to match English reduplication as in *I live in a house house, not a flat.*

```
a (\w)+ \1
```

This would match '*a house e*' (because only the *e* of *house* is referenced).

Instead put the repetition inside the group:

```
a (\w+) \1
```

Advanced Groups 2

Nested groups are possible, but note that the matched contents will overlap:

```
Pattern: (Hi, (\w+))!
Input  : Hi, Kim!
\1     : Hi, Kim
\2     : Kim
```

Advanced Groups 3

There are also non-capturing groups which have the benefits of groups but do not capture the text and are not assigned back-reference numbers. They are declared with `?:` at the beginning of the group.

```
(\w+(?:, \w+)*,? and \w+)
```

Here, the inner group is non-capturing and repeated, so the outer group captures the entire conjunctive phrase.

Presentation agenda

Substitution

Regular expression engines usually allow for substitution as well as matching. In the replacement pattern, back-references are allowed to insert captured groups.

Match:

```
(I|you|they)'ve
```

Replace with:

```
\1 have
```

This replaces *I've* with *I have*, *you've* with *you have*, etc.

Presentation agenda

Tools

Here are some tools for regular expressions:

- `grep` (Linux and macOS, Windows with a download)
- `http://www.regexbuddy.com/` (Windows)
- Many text editors:
  - `https://www.sublimetext.com/3`
  - `https://code.visualstudio.com/`
  - `https://www.gnu.org/software/emacs/`
  - …
- Web-based editors:
  - `https://regex101.com/`
  - `https://regexr.com/`
  - …
- Browser plugins let you search web pages
- Most programming languages have a regex module

Thanks

Thank you!