Taming The Dragon: An Introduction To Regular Expressions

Matthias Vill

New Stars of Data 2020

14-Aug-2020

Your sponsorship advertized here.

Agenda

About me

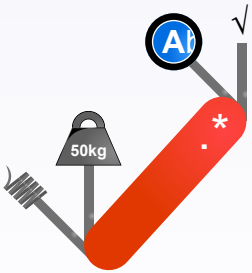
- Aged 33
- Diplom-Informatiker ($\simeq$ Master in Computer Science)
- First contact with coding in the 90s (VB 3.1)
- First contact with Regular Expressions and (My)SQL through PHP around 2000
- Currently employed as Software Developer with focus on T-SQL, but also using C# and PowerShell
- Experience in Java, Bash-, BAT-scripting, Ruby, ...

Regular Expressions

- IEEE POSIX defines 3 flavours: [1]
  - BRE (Basic Regular Expressions)
  - ERE (Extended Regular Expressions)
  - SRE (Simple Regular Expressions; deprecated)
- Closest to being the goto might be PCRE (= Perl Compatible Regular Expressions; similar to ERE) [2, 3]
- Like with SQL many things can easily be applied to most implementations
- **Note:** I will use a † to denote 'unsafe' stuff

Idea & Applications

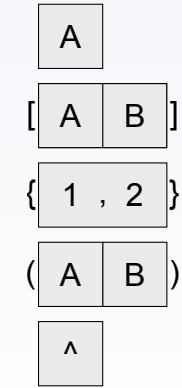
- Mathematical roots
- It is common to search in strings
  - needing more than a literal-match
  - not wanting a dedicated parser
- Flexibility comes at a price
- Not for everything, but often handy



Building Blocks

Regular Expressions are made up of

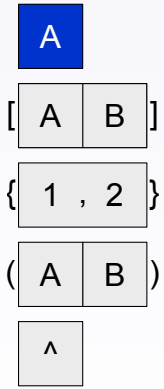
- **Literals**, matching themself
- **Classes**, matching one character out of a list
- **Quantifiers**, saying how often to match
- **Groups**, to treat sections separately
- **Anchors** and **Asserts**, aligning the match to the 'outside'
- and a few other things



Literals

Regular Expressions have escaping similar to C

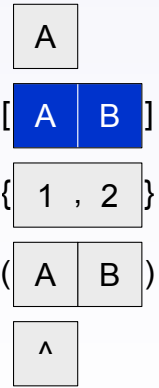
- most characters are literals (exceptions covered on the next slides)
- \ used to escape special characters and introduce special stuff
- \\, \r, \n, \t like C/JSON/...: backslash, carriage return, line feed, tab



Classes

You probably know classes from T-SQL's `LIKE`, but RegExp support a few predefined classes

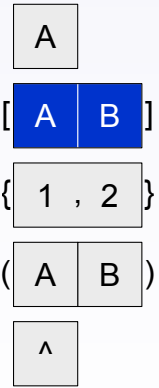
- `[acd]` matches `a`, `c` or `d`
- `[^acd]` matches a character except `a`, `c` or `d`
- `[a-d]` matches `a`, `b`, `c` or `d`
- `.` may† not match line-breaks, but everything else
- `\d` matches digits† (Unicode?!)
- `\w` matches 'word'-letters† (`[a-zA-Z0-9_]`)
- `\s` matches white-space
- `\S` matches non-white-space
- `[\da-fA-F]` matches hex-digits† (or `[0-9a-fA-F]`)
- refer to your implementation's documentation for more



Classes

You probably know classes from T-SQL's `LIKE`, but RegExp support a few predefined classes

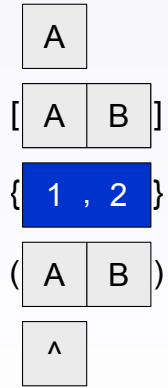
- `[acd]` matches `a`, `c` or `d`
- `[^acd]` matches a character except `a`, `c` or `d`
- `[a-d]` matches `a`, `b`, `c` or `d`
- `.` may† not match line-breaks, but everything else
- `\d` matches digits† (Unicode?!)
- `\w` matches 'word'-letters† (`[a-zA-Z0-9_]`)
- `\s` matches white-space
- `\S` matches non-white-space
- `[\da-fA-F]` matches hex-digits† (or `[0-9a-fA-F]`)
- refer to your implementation's documentation for more



Quantifiers

Things tend to repeat themselves. A lot.

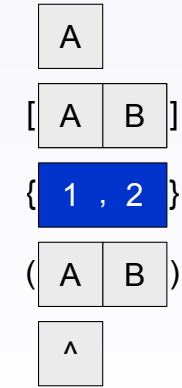
- `a?`: there will be one **a** or no **a**
- `a+`: there will be one **a** or more
- `a*`: there will be any number of **a**s or none
- `a{0,1}`: there will be between 0 and 1 **a**
- `a{1,}`: there will be one **a** or more
- `a{3}`: there will be three **a**s
- *special*: `a??`, `a+?`, `a*?` and `{}?` prefer fewer **a**s ('non-greedy'/reluctant)
- *special*: `a?+`, `a++`, `a*+` and `{}+` don't back-track (possessive)



Quantifiers

Things tend to repeat themselves. A lot.

- `a?`: there will be one **a** or no **a**
- `a+`: there will be one **a** or more
- `a*`: there will be any number of **a**s or none
- `a{0,1}`: there will be between 0 and 1 **a**
- `a{1,}`: there will be one **a** or more
- `a{3}`: there will be three **a**s
- *special*: `a??`, `a+?`, `a*?` and `{}?` prefer fewer **a**s ('non-greedy'/reluctant)
- *special*: `a?+`, `a++`, `a*+` and `{}+` don't back-track (possessive)

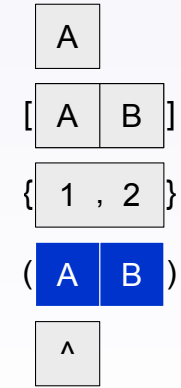


Groups

By themselves they don't do much, but you can use them to

- later refer to them:
  - `(['"])a\1`; in replacement `$1`
  - `(?<q>['"])a\k<q>` †; in replacement `${q}` † [4]
- have alternatives: `(ac|dc)`
- do styling: `(?:looks good\?)`
- change options: `(?i:sql)` or `(?-i)sql`
- repeat sections: `(?:[ad]c)*`

Noticed the ? at the beginning of some group-types?

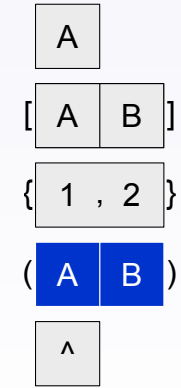


Groups

By themselves they don't do much, but you can use them to

- later refer to them:
  - `(['"])a\1`; in replacement `$1`
  - `(?<q>['"])a\k<q>` †; in replacement `${q}` † [4]
- have alternatives: `(ac|dc)`
- do styling: `(?:looks good\?)`
- change options: `(?i:sql)` or `(?-i)sql`
- repeat sections: `(?:[ad]c)*`

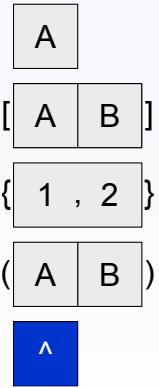
Noticed the `?` at the beginning of some group-types?



Anchors and Asserts

When something has to be there, but ...

- `^a`: **a** should be the first thing (on a line)
- `a$`: **a** should be the last thing (on a line)
- `a\b`: **a** should be at the end of a word†
- `/(?=DC)`: **DC** should come after /
- `/(?!AC)`: **AC** should not come after /
- `(?<=AC)/`: **AC** should come before /
- `(?<!DC)/`: **DC** should not come before /



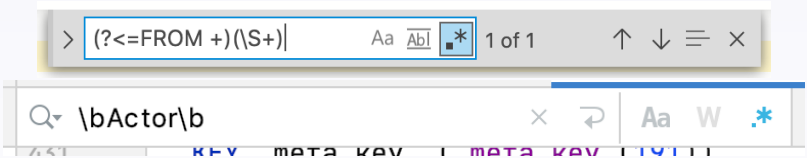
Note

Let's see some Regular Expressions.

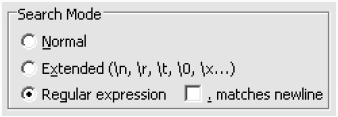
- Focus is on common usage
- It is easier to write Regular Expressions, than to read them
- Be aware of your context - you might need additional escapes
- Regular Expressions are programming: do at least some testing!
- Don't give up

In your editor

- If you are using a Microsoft/JetBrains editor/IDE: look for `.*` near the pattern input



- Notepad++ has 'Search Mode' at the bottom



- ...

In your code

These are only examples!

- PowerShell:
  - `'CONTOSO\Bob' -replace '\w+\\(?<user>\w+)', 'FABRIKAM\${user}'`
  - `Select-String -Path .\*.sql -Pattern 'FROM\s+(\S+)'`
- C#:
  - `Regex.Replace("CONTOSO\\Bob","\\w+\\\\(?<user>\\w+)", "FABRIKAM\\${user}");`
  - `Directory.EnumerateFiles(".", "*.sql") .Select(f => (f, c: File.ReadLines(f))) .SelectMany(t => t.c.Select((l,n) => (t.f, n, m: Regex.Match(l, @"FROM\s+(\S+)"))) .Where(mt => mt.m.Success))`

In your code

- MySQL 8:
  - **SELECT** REGEXP_REPLACE('CONTOSO\\Bob', '\w+\\\\(?<user>\w+)', 'FABRIKAM\\${user}');
  - ```
    SET @Haystack := 'SELECT * FROM Demo1; SELECT * FROM
        Demo2;';
    SET @Pattern := 'FROM\\s+(\\S+)';
    WITH RECURSIVE Matches AS (
        SELECT 1 AS Num,
            REGEXP_INSTR(@Haystack, @Pattern, 1, 1) AS Pos,
            REGEXP_SUBSTR(@Haystack, @Pattern, 1, 1) AS Str
        UNION ALL SELECT M.Num + 1,
            REGEXP_INSTR(@Haystack, @Pattern, 1, M.Num + 1),
            REGEXP_SUBSTR(@Haystack, @Pattern, 1, M.Num + 1)
        FROM Matches M WHERE M.Pos > 0)
    SELECT * FROM Matches M
    ```
- ...
    ```

Input Validation

- 'Our serial numbers always have 6 hex-digits'
- 'Names are made of two parts with a space in between'†
- 'a is repeated an "unprime" number of times'
- 'It is a valid E-Mail-address'

Input Validation

- 'Our serial numbers always have 6 hex-digits'
  `^[0-9a-fA-F]{6}$`
- 'Names are made of two parts with a space in between'†
- 'a is repeated an "unprime" number of times'
- 'It is a valid E-Mail-address'

Input Validation

- 'Our serial numbers always have 6 hex-digits'
  `^[0-9a-fA-F]{6}$`
- 'Names are made of two parts with a space in between'†
- 'a is repeated an "unprime" number of times'
- 'It is a valid E-Mail-address'

Input Validation

- 'Our serial numbers always have 6 hex-digits'
  `^[0-9a-fA-F]{6}$`
- 'Names are made of two parts with a space in between'†
  `^(\S+) (\S+)$`
- 'a is repeated an "unprime" number of times'
- 'It is a valid E-Mail-address'

Input Validation

- 'Our serial numbers always have 6 hex-digits'
  `^[0-9a-fA-F]{6}$`
- 'Names are made of two parts with a space in between'†
  `^(\S+) (\S+)$`
- '**a** is repeated an "unprime" number of times'
- 'It is a valid E-Mail-address'

Input Validation

- 'Our serial numbers always have 6 hex-digits'
  `^[0-9a-fA-F]{6}$`
- 'Names are made of two parts with a space in between'†
  `^(\S+) (\S+)$`
- '**a** is repeated an "unprime" number of times'
  `^(a{2,}?)\1++$`
- 'It is a valid E-Mail-address'

Input Validation

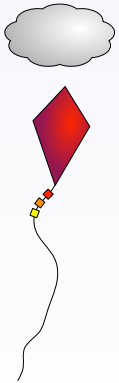
- 'Our serial numbers always have 6 hex-digits'
  `^[0-9a-fA-F]{6}$`
- 'Names are made of two parts with a space in between'†
  `^(\S+) (\S+)$`
- '**a** is repeated an "unprime" number of times'
  `^(a{2,}?)\1++$`
- 'It is a valid E-Mail-address'

Input Validation

- 'Our serial numbers always have 6 hex-digits'
  `^[0-9a-fA-F]{6}$`
- 'Names are made of two parts with a space in between'†
  `^(\S+) (\S+)$`
- '**a** is repeated an "unprime" number of times'
  `^(a{2,}?)\1++$`
- 'It is a valid E-Mail-address'
  There is a RegExp implementing RFC822 available at `https://www.ex-parrot.com/pdw/Mail-RFC822-Address.html`. A SO-answer suggests to check for an `@`, then send an E-Mail to verify

Editor Tasks - Demos

- Fold multiple spaces into one
- Find all usages of an object in a script (quoted, default schema, ...)
- Rename procedures: CREATE OR ALTER $\xrightarrow{\text{replaced by}}$ DROP IF EXISTS + CREATE OR ALTER
- Rewrite UPDATE to INSERT: UPDATE x SET y = ? $\xrightarrow{\text{replaced by}}$ INSERT x (y) VALUES (?)



Summary

- Regular Expressions are flexible and powerful
- MS SQL lacks native RegExp, but CLR can add it
- They are often easier to write than to read
- As with many things: use the right tool for the job

[5]

Feedback

Please provide feedback

Sources I


Regular expression. [Online]. Available: https://en.wikipedia.org/wiki/Regular_expression


Perl compatible regular expressions. [Online]. Available: https://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions


pcre2syntax man page. [Online]. Available: https://www.pcre.org/current/doc/html/pcre2syntax.html


Javascript built-in: Regexp: Named capture groups. [Online]. Available: https://caniuse.com/#feat=mdn-javascript_builtins_regexp_named_capture_groups


Garbatokid. How to regex (comic). [Online]. Available: https://twitter.com/garabatokid/status/1147063121678389253

Sources II


P. Resnick *et al.*, "Rfc 2822: Internet message format," 2001. [Online]. Available: https://www.rfc-editor.org/info/rfc2822


D. H. Crocker *et al.*, "Standard for the format of arpa internet text messages," 1982. [Online]. Available: https://www.rfc-editor.org/info/rfc822


Regular expression language - quick reference. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference


Runtime semantics: Getsubstitution. [Online]. Available: https://tc39.es/ecma262/#sec-getsubstitution


Regexp (regular expression) objects. [Online]. Available: https://tc39.es/ecma262/#sec-regexp-regular-expression-objects

Sources III


Class pattern. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html


Matcher.appendreplacement. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Matcher.html#appendReplacement(java.lang.StringBuilder,java.lang.String)


Idea-200402: Regex documentation is incorrect. [Online]. Available: https://youtrack.jetbrains.com/issue/IDEA-200402#focus=streamItem-27-3365527.0-0


Regular expressions. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions


Substitutions in regular expressions. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/base-types/substitutions-in-regular-expressions

Sources IV


About regular expressions. [Online]. Available: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_regular_expressions?view=powershell-7


regular expressions 101. [Online]. Available: https://regex101.com/