USER DEFINED NAMED PLACEHOLDERS FOR REGULAR EXPRESSION SEGMENTS IN COMPLEX REGULAR EXPRESSIONS

Milan Tair

Singidunum University,
32 Danijelova Street, Belgrade, Serbia

Abstract:
This paper describes an own implementation of a regular expression pre-processor written in PHP. It extends the regular expression functionality by allowing users to define named segments. These segments include custom character classes, matching groups *etc.* The pre-processor allows for writing complex regular expressions that are simpler to maintain. In addition, this paper presents a use case of the practical utilisation of the pre-processor. Furthermore, it includes a comparison of expressions written with and without user-defined segments.

Key words:
regular expressions, user defined segments, php, pre-processor, language extension.

1. INTRODUCTION

Regular expressions are helpful tools that are implemented in many applications and solutions. These solutions include business applications, web crawlers, content indexing engines (Sulzmann & Zhuo Ming Lu, 2016). They are also used for data transformation in preparation for migration (Kuznetsov & Simdyanov, 2006), as well as data cleaning (Dasu & Johnson, 2003) *etc.* Many programs use regular expressions that are very complex (Friedl, 2006). Complex regular expressions tend to be hard to maintain and edit. This is especially the case for regular expressions edited by programmers who are not familiar with the purpose of specific regular expressions. Because of this, long regular expressions are commented and documented (Curioso, Bradford, & Galbraith, 2010). However, this may only add to the overall length and complexity. Instead, this can be done by simplifying expressions while retaining original purposes and matching capabilities. One way is to use symbolic representations instead of somewhat complex sections of an expression. These symbolic representations include character class indicators which replace character groups in regular expressions (Friedl, 2006), which are the most frequently used, alongside Unicode properties. Such constructs are supported by most regular expressions standards. However, even with the helpers available in standardised regular expression implementations, regular expressions tend to remain too complex to be able to maintain easily. Instead of using only standard, predefined symbolic placeholders for regular expression segments, this paper presents an own implementation of a concept that extends this philosophy. It introduces user defined regular expression

Correspondence:
Milan Tair

e-mail:
milan.tair@gmail.com

DOI: 10.15308/Sinteza-2016-10-14

segments which can be used in traditional regular expressions. Such regular expressions need to be run through a pre-processor, which replaces these custom placeholders with lengthy regular expression segments wherever they may occur. This enables programmers to write seemingly shorter regular expressions that are easier to read by programmers and are hence easier to maintain and edit.

2. BACKGROUND

Definitions and problem description

Regular expressions are series of characters, some with special meaning, sometimes abbreviated as regex, that form a search pattern used for pattern matching within sections of a string or for matching an entire string (Li, *et al.*, 2013).

Most useful regular expressions used in large information systems and applications that are used for either searching through data, transforming or validating it (Sulzmann & Zhuo Ming Lu, 2016) are complex (Friedl, 2006). As explained in the introduction section of this paper, some solutions exist that aim to make such regular expressions easier to maintain. This is done either by making them shorter and therefore easier to read and understand or by adding comments and documenting sections or entire expressions. The aim of this is helping the maintainer gain additional insight and understanding of the meaning behind some of its parts (Curioso, Bradford, & Galbraith, 2010).

A sample of a complex regular expression for finding occurrences of a postal address written in a standard format defined by the Serbian Postal Services (Post of Serbia) in Latin or Cyrillic scripts for the Serbian language is shown in Listing 1.

```
/(\t*([A-z\u0080-\u00FF\u0100-\u017F\u0180-
\u024F\u0400-\u04FF\u0500-\u052F\u1E00-
\u1EFF\u2C60-\u2C7F\uA720-\uA7FF\ \-']+[\
]?\n)(\t*([0-9]+\.\ )?([\.A-z\u0080-
\u00FF\u0100-\u017F\u0180-\u024F\u0400-
\u04FF\u0500-\u052F\u1E00-\u1EFF\u2C60-
\u2C7F\uA720-\uA7FF\ \-'])+)(([0-9]+)|(bb))?(((\
+stan\ +)|(\ +стан\ +)|(\ *\/\ *))[0-
9]+)?(\n\t*[A-z\u0080-\u00FF\u0100-\u017F\u0180-
\u024F\u0400-\u04FF\u0500-\u052F\u1E00-
\u1EFF\u2C60-\u2C7F\uA720-\uA7FF\ ]+\n)?(\t*[0-
9]{5}\ *)([A-z\u0080-\u00FF\u0100-\u017F\u0180-
\u024F\u0400-\u04FF\u0500-\u052F\u1E00-
\u1EFF\u2C60-\u2C7F\uA720-\uA7FF\ ]+\n))/
```

Listing 1. A regular expression for matching postal addresses written in the official Serbian address format.

The sample from listing 1 does not cover all conditions explained in the instructions (Post of Serbia). Nevertheless, it does cover the majority or addresses instances that occur in the postal system of Serbia. Military addresses, PO Box users' addresses and addresses without street names, such as those in smaller settlements, are not always matched by this regular expression. Nonetheless, it can be used to match the majority of Serbian postal address.

Even though there are much more complex expressions that are used frequently in larger specialized applications, this example is used to define the problem whose solution is explained in this report.

Existing solutions

A number of solutions exist that aim to increase the maintainability of long and hard to read regular expressions. The Perl programming language provides a mechanism of spreading a regular expression along multiple lines (Conway, 2009). It also supports adding comments. This method increases readability, which in term increases maintainability of the regular expression code. However, it does not reduce the length of the regular expression. Instead, quite the opposite, the length of the expression additionally increases. The lack of abstraction in regular expressions (Erwig & Goponath, 2012) is one of the key deficiencies. It causes scalability problems that result in regular expressions growing quite large quickly. Also, much of the regular expression code is sometimes redundant (Dasu & Johnson, 2003). A combined explanation for the date expression presented in (Erwig & Goponath, 2012) gives an interesting method of explanation. This method can easily be transformed in an SQL-like language. This language can be used for defining regular expression. However, asides from a presentation in the mentioned paper, no implementation has been reported to this day. Similar concepts exist in modern web applications developed using the Model-View-Controller pattern. This is particularly the case in the area of request routing, where the router component allows for defining route constraints. Segments of the route are named and later defined using regular expressions (Microsoft Development Network).

3. DISCUSSION

The user defined named regular expression segment placeholder pre-processor is written with the aim to provide programmers with a way to make otherwise complex and long regular expressions shorter. Thus,

this would also make them easier to read and interpret. This increases maintainability of the regular expression. In addition, named or symbolic segments can be shared between programmes across many different projects. Additionally, the implementation has two modes of writing and organising the expression code. The first is the multiline mode where all whitespaces before and after the line including line breaks are ignored. The second is a standard single line expression. The multiline mode supports adding comments at the end of the line, where a comment starts with the hash sign character and ends with the line feed. Each regular expression segment can be assigned a name. This is analogous with defining a custom type in some programming languages. The named segment is stored in its own file. These files exist in the segments directory with the extension .nrexss. The extension is an abbreviation of named regular expression segment source. The pre-processor is written in the PHP scripting language in form of a final class. The pre-processor code is contained in a single class file. Nevertheless, future revisions and expansions of the pre-processor functionalities might result in the code being refactored and organised into a complete PHP library. The pre-processor needs to be instantiated before use in the program and a sequence of methods with adequate arguments needs to be called before the pre-processor object is ready to compile a regular expression, which includes named segments that will be replaced with the corresponding full-length regular expression code. An example of a PHP code snippet that illustrates the use of the pre-processor is shown in Listing 2.

In the code shown in listing 2, the $pattern variable's value represents a regular expression with embedded named segments. The pattern still matches postal addresses identically to the one shown in listing 1. But, it is much easier to read and understand as well as to maintain and expand. The named segments are defined as plain text files with the .nrexss extension and stored in the ~/rxns/segments/ directory. The file names match the segment name between braces.

When the pre-processor's matchAll method is called with a reference to an array as an argument, the pattern is first normalised so that all whitespaces at the beginning and the end of each line are trimmed off and all lines are joined together into a single continuous line. After this operation is completed, a regular expression matching of all embedded named segments is performed. The method filters out duplicates. Names of the matched segments are extracted and a regular expressing named segment file with the matching filename is looked up in the defined segment directory path. All mentions of the embedded named segment are replaced with the trimmed content of the selected file. This is repeated for all named segments. If at least one named segment remains embedded in the pattern string due to its corresponding file not being found, the method will throw an object of a custom class FileNotFoundException. This class extends PHP Exception. If no error occurs and the pattern is prepared, regular expression matching is performed and the result set is normalised and transformed into a single numerically indexed array returned by reference.

```
<?php
  $pattern = '(
    {Personal name sr}
    (
      {Street name sr}
      {Building number}?
      {Apartment section sr}?
    )?
    {Municipality name sr}?
    {Postal code sr}
    {City name sr}
  )';

  $bigText = file_get_contents('big.txt');

  $pp = new RexExpPreprocessor();
  $pp->setSegmentPath('~/rxns/segments/');
  $pp->setPattern($pattern);
  $pp->setSubject($bigText);

  $matches = [];

  $pp->matchAll($matches);

  print_r($matches); # Show matched data
?>
```

Listing 2. Sample PHP code showing the use of the regular expression named segment pre-processor class for extracting postal addresses written in the official Serbian address format.

As seen in listing 2, after calling the pre-processor object's matchAll method, the $matches array is printed to screen for review. The output is a human-readably formatted array (Zend Technologies).

4. A USE CASE

Originally, the user defined named regular expression segment placeholder pre-processor was developed as part of a project to extract postal addresses, phone numbers and other structured information from large collections of plain text documents. These documents were created by running scanned pages of printed media through object character recognition (OCR). The aim of the project was to collect above-mentioned information for creating a database for marketing activities for an electronic marketing solutions software development company from Kragujevac, Serbia in 2008. Initially, the useful data extraction software used moderately complex regular expressions for matching certain standardised formats of different structured information. Over time, the complexity of regular expressions had increased. Also, the structure of the development team frequently changed and a need for a way to make regular expressions easier to maintain was recognized. The initial idea for the implementation explained in this report was presented and tested. The performance impact of an additional regular expression pre-processor was not evaluated in detail, but some ad hoc measurements had shown that it was inconsequential to the overall performance of the data extraction application. Likewise, it did not impact the schedule of the project.

Initial results

The initial results of using the pre-processor were positive. Regular expression coders had increased output. Also, they had moved up the useful data structure matcher pattern development schedule much faster than before. The learning curve was steep. Thus, the impact of the introduction of a new technology was almost of no consequence to the productivity of the development team. The final result was a beta version of the application that was fully automated. It extracted useful structured information from scanned pages as they entered a stack after being scanned. Document scanning was the only manual job in the system. Future solutions were to implement an automated book scanning hardware. Unfortunately, the company was closed in 2010 and no further development of this application has been done.

Known deficiencies

During the implementation and development process, some deficiencies of the initial design were identified. The first was the lack of recursive properties of the named segment replacement. This was both in the pattern and in other named segments was a major deficiency of the design. This resulted in many identical pattern segments being used redundantly in many segments. Preferably, they should have been written once as a small segment and then embedded in other segments. This deficiency was never solved even through the solution was proposed. It was due to the lack of approval from the project management. The second deficiency was the lack of a naming convention for segments. As for this use case, the development team structure changed frequently. This and the lack of a naming convention caused a situation where multiple programmers had written segments with different names that have matched the same pattern. After it had become obvious and began slowing down development a naming convention was introduced. The naming convention was changed multiple times during the development process. Ultimately, the final version was established. The segment was to be named as short as possible while still regaining enough information to clearly identify the purpose and the pattern it was supposed to match. All segments had to include a two-letter ISO 639-1 language code (Codes for the Representation of Names of Languages) at the end of their name to indicate which language they were specialised for. Named segments that were not specialised for any language, but were used for general pattern matching, such as e-mail were not to be suffixed at all. The third deficiency that was recognised early during the development process was the lack of the ability to group named segments into categories or namespaces. In this particular use case, the number of created named segments was greater than one hundred and was hard to organise by naming alone. A naming convention revision was suggested to attempt to deal with this issue, but was never approved.

5. FURTHER DEVELOPMENT

Official development of the pre-processor was halted in 2009. It was intermittently continued over the years for personal use by the author. There is some aspiration to publish a completely rewritten version of the pre-processor as an open source project. Hopefully, most of the unsolved deficiencies will be solved by design instead of convention.

6. CONCLUSION

This paper aimed to present an example of the use of regular expressions in an own implementation designed to extend the original functionality of the regular expression engine integrated into the PHP language. It had shown a use-case where this extension was used and examples of how these new features were designed and applied for solving the specific problem. The problem that was illustrated was the matching and extraction of postal addresses in the official format for the Serbian language from bodies of text. The main contribution of this implementation is the ability to write complex regular expressions that are simple to maintain by creating aliases and expression groups with symbolic names. These segments were shown in examples provided for illustration purposes.

REFERENCES

Codes for the Representation of Names of Languages. (n.d.). (ISO) Retrieved 1 17, 2016, from https://www.loc.gov/standards/iso639-2/php/code_list.php

Conway, D. (2009). *Perl Best Practices - Standards and Styles for Developing Maintainable Code*. O'Reilly Media.

Curioso, A., Bradford, R., & Galbraith, P. (2010). *Expert PHP and MySQL*. John Wiley & Sons.

Dasu, T., & Johnson, T. (2003). *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons.

Erwig, M., & Goponath, R. (2012). E*xplanations for regular expressions. Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering* (pp. 394-408). Berlin: Springer-Verlag. doi:10.1007/978-3-642-28872-2_27

Friedl, J. (2006). *Mastering Regular Expressions*. O'Reilly Media, Inc.

Kuznetsov, M., & Simdyanov, I. (2006). *Regular Expressions. In PHP Security & Cracking Puzzles* (pp. 125-128). BHV-Petersburg.

Li, W., Min, Y., Yuanpeng, Z., Xingyun, G., Danmin, Q., Kui, J., & Jiancheng, D. (2013). *A Rule-Based Algorithm for Extracting Medical Data from Text.* International Symposium on Signal Processing, Biomedical Engineering, and Informatics (SPBEI 2013). DEStech Publications.

Microsoft Development Network. (n.d.). *Route.Constraints Property*. (Microsoft) Retrieved 12 20, 2015, from https://msdn.microsoft.com/en-us/library/system.web.routing.route.constraints(v=vs.110).aspx

Post of Serbia. (n.d.). *Instruction for correct addressing and preparation of postal items before posting them to the PE Post of Serbia*. Retrieved 12 10, 2015, from http://www.posta.rs/dokumenta/eng/posalji/Adresovanje-posiljke.pdf

Sulzmann, M., & Zhuo Ming Lu, K. (2016). *Fixing Regular Expression Matching Failure*. Kochi: (unpublished) 13th International Symposium FLOPS. doi:10.13140/RG.2.1.2950.3440

Zend Technologies. (n.d.). *PHP: print_r - Manual*. Retrieved 12 20, 2015, from http://php.net/manual/en/function.print-r.php