A Lightweight Framework for Regex Verification

[ACM Student Research Competition, PLDI '17]

Xiao Liu
Advisor: Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University, University Park

1. PROBLEM AND MOTIVATION

Regular expressions are widely used in computer programs for pattern searching and string matching due to the high effectiveness and accuracy. According to Chapman and Stolee's study on regular expression usage, developers are in favor of regular expressions which are contained in 42% among nearly 4,000 open-sourced Python projects [5]. Researchers have also explored applying regular expressions to test case generations [1, 7, 8, 16], specifications for string constraint solver [10, 17], and queries for some data mining framework [3]. As a fundamental approach for lexical analysis, it has been further extended to advanced functions for security purposes, such as input validation [19] and network intrusion detection [13]. Despite the popularity, it is still considered complicated for users, even experienced programmers, due to the low readability and the hardness to construct correct ones. Because of the gradually increasing adoptions of agile software development, requirements and solutions usually evolve through quick self-organizing, which causes the software development more error-prone. In this situation, input validations using regular expressions are constantly being found inefficient due to errors in the constructed expressions, resulting in security vulnerabilities [11, 14].

Due in part to their shared use across software engineering and how susceptible regexes are to errors, many researchers and practitioners have developed tools to support more comprehensible verification. To verify the correctness of regular expressions, testing is widely adopted. These techniques solve the problem to some extent but are far from sufficient. In this study, we are interested in enabling end-users to verify the correctness of regular expressions. We present a novel lightweight framework that verifies the consistency between natural language requirements and corresponding regular expressions using equivalence checking. Incorrect regular expressions will be detected when inequivalences are found between the specifications and target expressions. To justify the novelty of our proposed framework, we build a prototype tool. To evaluate our idea, we conduct a small-scale lab study with 4 participants on 10 verification tasks. Both efficiency and usability are demonstrated.

2. BACKGROUND AND RELATED WORK

To verify the correctness of regular expressions, proposed approaches in previous studies focus on testing and debugging methods, which solve the problem to some extent but are far from sufficient. Black-box testing is mostly adopted. Assisted with online and offline tools, e.g. regex101 [6], developers can find bugs in regular expressions through a set of test cases. However, this kind of heavyweight frameworks increase their cognitive load to understand both requirements and the corresponding regular expressions and it requires a significant amount of human labor to build test suites. In addition, it is difficult to determine the test coverage of these self-generated test cases, especially for negative test cases and those for testing Kleene stars.

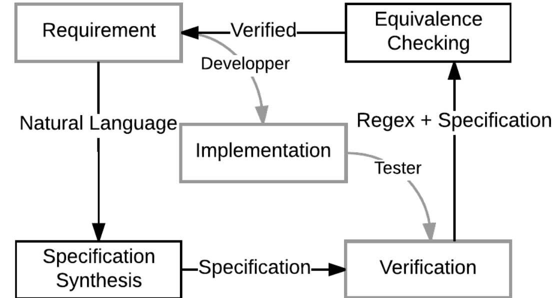

Figure 1: Framework for regex verification

White-box testing is also discussed by a few researchers and some visualization tools are developed as assistance. Regular expressions can be visualized, for instance, as a graph structure [4, 9, 12]. The demonstrated graph will present the path in a regular expression in a clearer way, therefore, saves human labor to compose cases and increases the readability despite the expressions without clear demarcation. Fabian et al. [2] also introduced the approach that provides advanced visual highlighting of matches in a sample text, which is easy to use and helps understanding regular expressions. However, these techniques still rely on test cases where the test set explosion problem still remains.

Naturally, we think of verification techniques which check the required properties based on mathematical proof but not dynamic test cases. However, only a few studies have been done for regular expressions. Existing papers mostly focus on verification of the syntax level properties. Static program analysis techniques like type systems [15] are adopted for exception detection for regular expressions, e.g. IndexOutOfBound, at the compile time. According to our knowledge, there is no previous work on verifying semantic level properties for regular expression.

3. APPROACH AND UNIQUENESS

In this paper, we propose a lightweight verification framework that enables end-users to verify regular expressions with requirements and specifications as shown in Figure 1.

In this flowchart, each block represents a module and the modules connected by gray arcs are fundamental processes in the typical software developments, e.g. waterfall. The cores of our framework are the *Equivalence Checking* module and the *Specification Synthesis* module. With the target regular expression and synthesized specifications also in regular language, we will perform equivalence checking based on an analysis of the equivalence between the two regular expressions to validate whether the specified properties hold or not and then report to the developers. And to lower the bar of entry for specification synthesis, a rule-based natural language processing technique is employed.

Consider the following regular expression for matching a valid password:

```
(?!^[0-9]*$)(?!^[a-zA-Z]*$)^([a-zA-Z0-9]{6,10})$
```

According to the requirements, *The password must be between 8 and 10 characters, contains at least one digit and one alphabetic character, and must not contain special characters.* Therefore, it matches password like `enduser101` and rejects `enduser1001`.

With the proposed framework, by partitioning the requirements and process the natural language descriptions, our tool will generate four specifications in the regex-like syntax. However, it detects an ambiguity in the requirement that it is not clear whether the endpoints should be included or not for the length interval. Therefore, the system will interactively ask the user to *specify for including or excluding endpoints in the first sentence.* After the user's response that both endpoints are included, a specification will be generated as follows corresponding to each sentence:

```
my @spec = (
.{8,10}             # 8 to 10 characters, ep include
(?=.*\d)            # must contain at least one digit
(?=.*[a-zA-Z])      # must contain one alphabet
(?=[a-zA-Z0-9]*)    # must not contain special char
);
```

With the synthesized specifications and the target regular expression, equivalence checking will be performed to analyze the correctness of the target regular expression. It will detect whether the properties specified by the specification are hold or not. In this example, results will show `Verification Failed` because of the incorrect length. The results will be reported to the developer for later debugging.

We not only propose a new approach, but also solve the ironic problem of regex verification. It's new because it's the first preliminary work that proposes to use natural language to validate regex. Why ironic? Typical specification languages are more powerful, or in other words, more complicated than regular languages. So verification of a regex using a specification in such languages is overkill and potentially makes things unnecessarily complicated. But if we simply create another regex according to requirements to act as the specification, and we'll simply need to check the equivalence. However, this solution overlooks the complexity of regular expressions. Actually a regex can be complicated even reach a size of 6.2kb [18]. Therefore, it's necessary to develop new methods for validating such big and complicated regexes.

4. RESULTS

To demonstrate the functionality and usability of the verification framework, we build a web-based prototype. We conducted a lab study with 4 participants on 10 verification tasks. By analyzing the success rate and time to find errors in regular expressions using different methods, composing test suites or using the verification method, we conclude that our prototype enhances both efficiency and usability.

Figure 2: Prototype with a web service

# of Test	Tester 1	Tester 2	Verifier 1	Verifier 2
1	132s	296s	75s	96s
2	32s	56s	90s	86s
3	fail	fail	75s	96s
4	100s	96s	80s	107s
5	80s	60s	90s	78s
6	229s	276s	90s	87s
7	150s	156s	80s	57s
8	73s	96s	58s	67s
9	220s	96s	70s	87s
10	fail	fail	fail	370s

Table 1: Experimental results

For this prototype, we care about whether incorrect regular expressions, those are not consistent with requirements, can be detected. In addition, is it more efficient than conventional testing methods? A small-scale lab study was conducted with users who are familiar with the regular expression testing and will be trained to write accurate requirements with vague descriptions and a limited number of examples. We collected 10 regular expressions and their requirements for different domains of uses which are documented in Table 1 with some well-known vulnerabilities, such as escape missing and range mismatch. These users were separated into two control groups and required to try their best to verify the set of regular expressions with testing and our verification techniques, respectively. Their performance concerning successfulness in detecting incorrect regular expressions and efficiency (time) for both tools were recorded. These data quantitatively indicated the functionality of the verification technique.

Since the accuracy for the specification synthesis is not 100%, researchers interrupted into the experiment when a wrong specification is synthesized, but error logs were kept for accuracy analysis. In another word, we ensured the consistency between the specifications and their corresponding requirements in the lab study. In this lab study, 10 verification tasks were conducted and all of them were detected with error correctly by users with our verification method, and 8 of them were found with the conventional testing method. However, we also recorded the time for them to complete the task which can be seen in Table 1. For Test 2, since these

participants are familiar with IP address testing and the error is a common one according to their experience, they are more confident to claim it erroneous in a shorter time. Except for Test 2, our verification method performed better than or as well as the conventional testing method to detect errors especially when there are many options or there is a range error. In these cases, natural language is more expressive than test cases that one sentence description can represent a few or more test cases.

5. CONTRIBUTION & CONCLUSION

We presented a lightweight framework for regex verification in this paper. It is based on an equivalence checking method between formal specifications and the target regular expressions. To enhance the usability of the proposed framework, we incorporated a specification synthesis module which automatically generates specifications in formal language from natural language descriptions of the requirements. We built a prototype for the proposed framework and conducted a lab study for evaluation. Our approach showed an enhancement regarding efficiency and usability.

6. REFERENCES

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, Aug. 2013.

[2] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf. RegViz: Visual debugging of regular expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 504–507. ACM, 2014.

[3] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[4] A. F. Blackwell. See what you need: Helping end-users to build abstractions. *Journal of Visual Languages & Computing*, 12(5):475–499, 2001.

[5] C. Chapman and K. T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 282–293, New York, NY, USA, 2016. ACM.

[6] F. Dib. Regex101. https://regex101.com/, 2014.

[7] S. J. Galler and B. K. Aichernig. Survey on test data generation tools. *International Journal on Software Tools for Technology Transfer*, 16(6):727–751, 2014.

[8] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang. Jst: An automatic test generation tool for industrial java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, Piscataway, NJ, USA, 2013. IEEE Press.

[9] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. In *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, pages 6–10, New York, NY, USA, 2000.

[10] A. Kieżun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering and Methodology*, 21(4):25:1–25:28, Nov. 2012.

[11] R. A. Martin. Common weakness enumeration. *Mitre Corporation*, 2007.

[12] N. Moreira and R. Reis. Interactive manipulation of regular objects with FAdo. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 335–339, New York, NY, USA, 2005.

[13] T. B. Project. The bro network security monitor. https://www.bro.org/, 2015.

[14] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM.

[15] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26. ACM, 2012.

[16] N. Tillmann, J. de Halleux, and T. Xie. Transferring an automated test generation tool to practice: From pex to fakes and code digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 385–396, New York, NY, USA, 2014. ACM.

[17] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, New York, NY, USA, 2014. ACM.

[18] P. Warren. Mail::rfc822::address: regexp-based address validation. http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html, 2012.

[19] A. S. Yeole and B. B. Meshram. Analysis of different technique for detection of sql injection. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, pages 963–966, New York, NY, USA, 2011. ACM.

APPENDIX

A. A REGULAR EXPRESSION EXAMPLE

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\
r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\
r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]
+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]
))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r
\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.
(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:
\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?
```

```
[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]
+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]
))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t]
)*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]
+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t]
)*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[
\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\
n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^
\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.
(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\
031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:
\\".\[\]]))|\[([^\[\] \r\\]|\\.)*\](?:(?:\r\n)?
[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:
\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\
.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\
.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:
\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]
))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\
n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^
\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.
(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\
031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:
\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
\t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\".
\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:
\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n
)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]
+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]
))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t
])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r
\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[
^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".
\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r
\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[ \]]))|\[(
[^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[
^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r
\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t]
)*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".
\[\]\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r
\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\
n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]
+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[
\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t]
)*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[
^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\
".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r
\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t]
)*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t ])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r
\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?
[ \t])*)(?:,\s*(?:(?:[^()<>@,;:\\".\[\]\000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.| (?:(?:\r\n)?[ \t]
))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\
n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^
\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[
\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:
[^()<>@,;:\\".\[\]\000-\031]+(?:(?:(?:\r\n)?[ \
t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r
\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:
\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\
.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<
(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \0
00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r
\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\
n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]
+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[
\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t]
)*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:
[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?
[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\
".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r
\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t]
)*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:
[^()<>@,;:\\".\[\]\000-\031]+(?:(?:(?:\r\n)?[ \
t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r
\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?
[ \t])*))*)?;\s*)
```