Approaching Punctuation Errors in the New Proofreader of Czech

Vojtěch Mrkývka

Faculty of Arts, Masaryk University
Arne Nováka 1, 602 00 Brno, Czech Republic
mrkyvka@phil.muni.cz

Abstract. As the progress of a new online proofreader of Czech continues, so does the development of particular proofreading modules that make it whole. The position of the punctuation one is rather specific as its inner workings differ from the usual structure. This paper focuses on the design of the punctuation module, its specifics and obstacles which followed or still follow its development process.

Keywords: Proofreading · Punctuation · Regular expressions

1 Introduction

The new online proofreader of Czech is a new tool developing at the Faculty of Arts, Masaryk University since 2018. Contrary to similar products, this project aims to (hopefully) address a broader spectrum of errors, not ending with a spellchecker but starting with it. Using knowledge of both Czech language and computational linguistics gained at PLIN[1], the team aims to create a rule-based system by formalising existing basic research results supplemented by own findings. This paper will focus on one specific part of the tool – the punctuation module, its specific nature within the proofreader and obstacles that were or yet have to be overcome.

2 About the proofreader

Although the nature of the proofreader varied in time[2], the current (and hopefully final) solution – Plinkorektor[3] – has a form of singular API with a modular internal structure communicating with the user interface to present results (see Fig. 1). However, the final goal for the API is to be on any specific user interface fully independent.

As mentioned above, the API consists of multiple internal modules called simultaneously as soon as their requirements are fulfilled (see Fig. 6). Additionally, the current version allows the user to specify whether he or she wants to

[1] Computational linguistics study programme at Faculty of Arts, Masaryk University
[2] For more information, see my previous papers on the topic[1,2,3].
[3] https://korektor.plin.cz/

A. Horák, P. Rychlý, A. Rambousek (eds.): Proceedings of Recent Advances in Slavonic Natural Language Processing, RASLAN 2021, pp. 107–113, 2021.

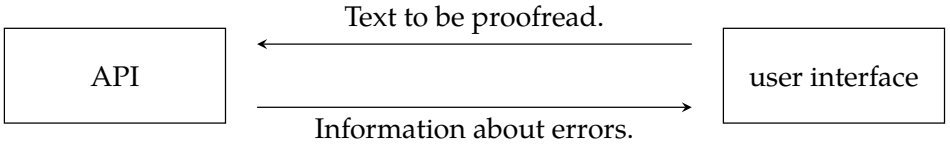

Fig. 1: Communication between API and the user interface.

call only a specific part of the module portfolio, omitting dependencies that are redundant for the selection.

The team working on the API creates the detection rules for the different types of user errors (spelling, commas, or subject-verb/subject-object grammatical agreement) that need to be provided with the correction overlays. Rules, which are the outcome of these overlays, are strongly tied to prior tokenisation operating solely on replacement operation.

3 The punctuation module

The punctuation module is based on the bachelor thesis of Zbyněk Michálek [4]. It contained a detailed list of regular expressions (44 in total), which can be used for automatic detection and correction of selected issues. These expressions were implemented in the user interface, automatically correcting some of the errors before calling the API. From the current point of view, this solution was unfortunate because it added (weak) API dependency on the user interface, preventing the Plinkorektor API from being fully used in a different environment. The only logical solution was to migrate these rules to the API.

Most of the proofreading modules natively work with the tokens using shallow parsing grammars for the SET analyser [5] to detect and mark the problematic sequences. However, this is not the case with the punctuation module. As mentioned above, its determination is based on regular expressions, so it is working with text independently of the tokens; however, the API needs the token mapping for the correct output production. Fortunately, the match object from Python's `re` package can provide information about on which character position the regular expression match starts, ends or both using `start()`, `end()` or `span()` methods respectively. Using a pointer array, the context of the matches could be determined easily[4] (see Fig. 2). However, the `re` package later showed to be insufficient, as it does not operate with POSIX classes. Fortunately, the alternative `regex` package can be used in its place. The immodest goal of the author is for `regex` to replace `re` in the future as it provides more functions (for example, already mentioned POSIX class compatibility) while maintaining maximum backwards compatibility with `re`[6]. Sadly, even the package replacement did not fully fulfil all the needs. Although the base of regular expressions is usually the same across the programming languages, further nuances can make the specific expression unusable within different

[4] Additional context limitation in case of some of the rules was to include additional groups into the expressions themselves for start/end/span methods to operate with.

environments. For example, for detection of space followed by a comma[5] Michálek uses the expression `[:blank:],`; however, in Python, POSIX classes have to be encapsulated in another pair of brackets as `[[:blank:]],` in this case.

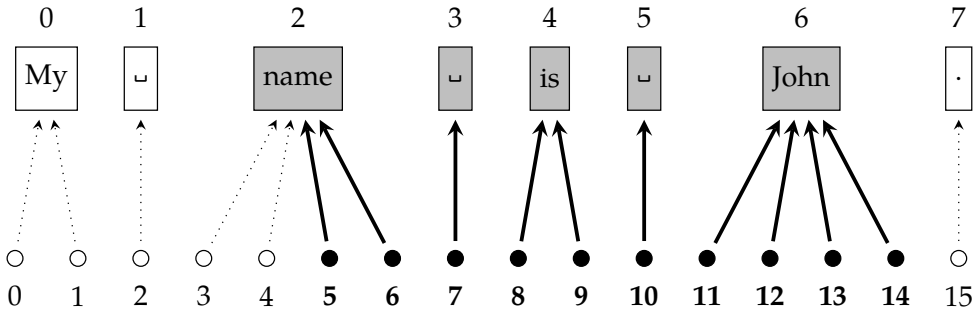

Fig. 2: An example of mapping the regular expression (`[me.*n]` in this case) on tokens.

The correction rules for these expressions can be divided into two approaches, one using regular expressions only for error detection and the other for both detection and replacement. Using the example mentioned above, the space token can be selected using capturing group (`([[:blank:]]),`) and removed by the simple *replace with nothing* rule (see Fig. 3).

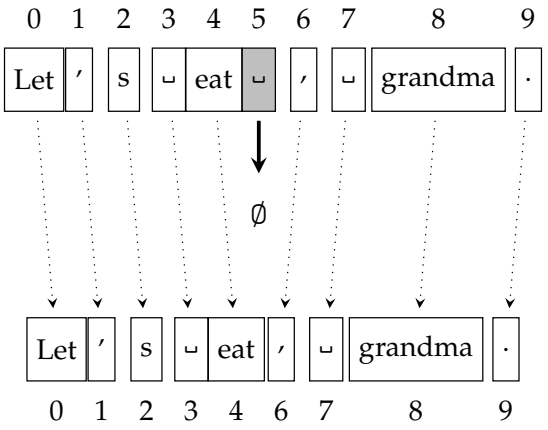

Fig. 3: An example of the *replace with nothing* rule. The space before comma in the sentence "Let's eat , grandma." is replaced with an empty string virtually removing the space as a result.

It should be mentioned that Michálek provided replacement patterns for all of his regular expressions; however, as the original intended usage was

[5] In Czech, there should never occur space before a comma. In the position after a comma, the space is usually present, but there is an exception for numeric expressions.

different[6] he uses capturing groups (if he uses them at all) for the parts of the expression that shall be kept after correction (e.g. to be used in replacement pattern) rather than the parts to replace. The second approach is based on using these rules ignoring the token structure of corrections by moving the result of the replacement pattern into the first affected token, leaving the others blank[7]. However, this approach is discouraged due to problematic compatibility with other modules, as some of the expressions can span over many tokens (see Fig. 4).

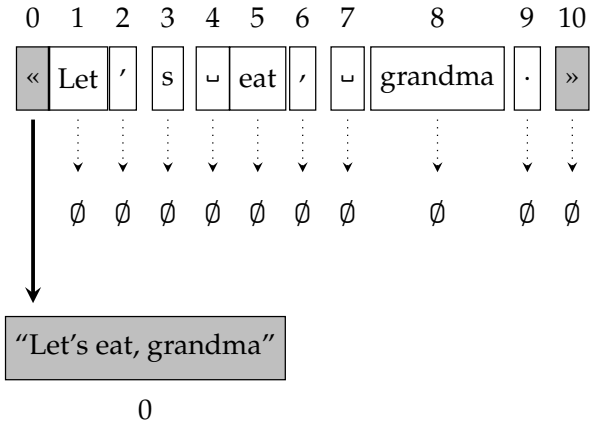

Fig. 4: An example of replacing the whole quotation segment (because of incorrect quotation marks) with single token when using regular expression replacement rules.

As on a related problem can be looked at the Michálek's expressions themselves. As mentioned, their intent was to be used strictly as automatic correction and do not always fulfil Plinkorektor needs. For example, the expression `\u00A7([:blank:]?)([0-9])` used to replace space after § for no-break space cannot be used as is, as no-break space is part of `[:blank:]` POSIX class and it would create the false-positive message. Aside from this, opinion-based issues need to be resolved when dealing with automatic corrections, but in other cases can be left for the user to decide. For example, Michálek uses the expression `\?\?+` to replace all cases of multiple question marks with exactly three[8]. However, in the case of Plinkorektor, the user can select if he or she wants to use one or three question marks when exactly two were input. Similarly, there is a question of whether the expression to remove additional whitespace before the colon and the one missing after it should be treated as one issue or two separate

[6] Michálek intended to use his rules solely for automatic correction of given issues, however, the philosophy of Plinkorektor is to provide users information about which corrections can be used as automatic but leaving the choice on them.

[7] The text is retokenised every time the API is called, so the change of token structure will not affect further API calls.

[8] He works similarly with exclamation marks.

ones. This can relate to the abovementioned dilemma whether use regular expressions also for the replacement purposes, as splitting of selected expressions (or using suitable capture groups) can help keep the tokens intact (compare Fig. 4 with Fig. 5).

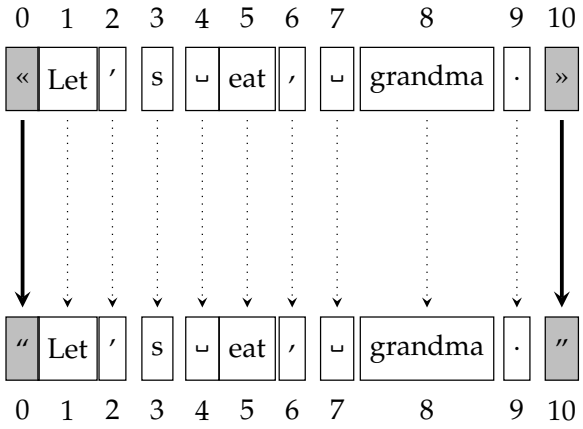

Fig. 5: An example of replacing the quotation segment by parts (because of incorrect quotation marks) with the most of the tokens left in place.

4 Common issues with the API

Lastly, there are issues with the API itself that prevent the punctuation module from being better as part of it instead of a separate tool (for example, as part of the user interface as it is right now). The main problem is that the current API is still relatively slow to be entirely usable in the production environment (see Fig. 6). Although there are still options to speed specific parts up, some modules will always be slower than others. The supplementary option is to give users better ways to call parts of the API independently to, for example, check the text with the fast modules first with additional correction by the slower modules after they finish their processing.

5 Conclusion

The new online proofreader of Czech still has many issues that need to be addressed, and the ongoing development of the punctuation module (currently at circa 10%) is no exception. The situation presented above and the whole of the Plinkorektor issues can be summarised as quantity over difficulty situation, meaning there is a minimal number of problems, which can be considered hard. However, easy ones come in such quantity that progress is not always optimal. On the other hand, looking at the overall work done versus to be done, the production-wise usable product is undoubtedly just around the corner.

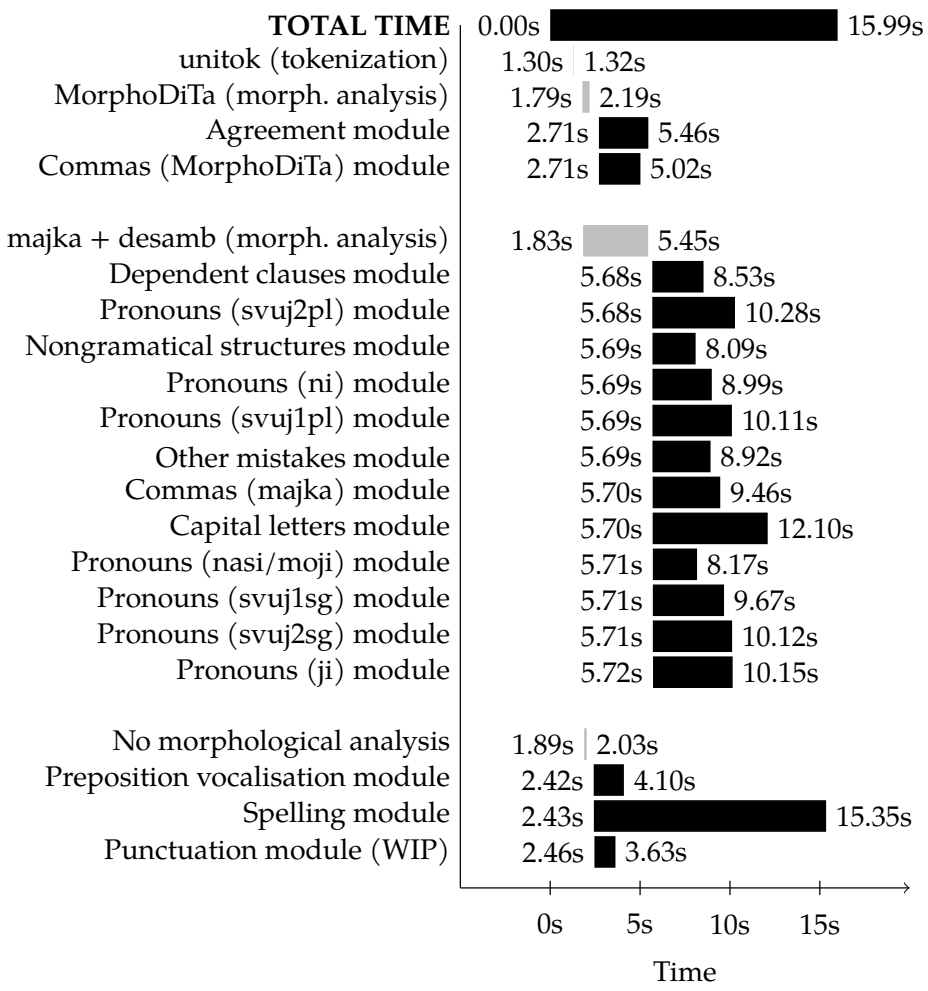

Fig. 6: Runtime of different modules in the API.

Acknowledgements. The work was supported by the project of specific research *Gramatika a lexikon češtiny* (Grammar and lexicon of Czech; project no. MUNI/A/1181/2020).

References

1. Mrkývka, V.: Webové rozhraní pro automatický jazykový korektor češtiny [online]. Diplomová práce, Masarykova univerzita, Filozofická fakulta, Brno (2018 [2021-10-28])
2. Mrkývka, V.: Towards the New Czech Grammar-checker. In Horák, A., Rychlý, P., Rambousek, A., eds.: Proceedings of the Twelfth Workshop on Recent Advances in Slavonic Natural Languages Processing, RASLAN 2018, Brno, Masaryk University (2018) 3–8
3. Mrkývka, V.: Recent Advancements of the New Online Proofreader of Czech. In Horák, A., Rychlý, P., Rambousek, A., eds.: Proceedings of the Thirteenth Workshop on Recent Advances in Slavonic Natural Languages Processing, RASLAN 2019, Brno, Masaryk University (2019) 43–47

4. Michálek, Z.: Algoritmizace hromadných oprav vybraných typograficko-pravopisných jevů českého jazyka [online]. Bakalářská práce, Masarykova univerzita, Filozofická fakulta, Brno (2016 [2021-10-28])
5. Kovář, V., Horák, A., Jakubíček, M.: Syntactic Analysis Using Finite Patterns: A New Parsing System for Czech. In: Human Language Technology. Challenges for Computer Science and Linguistics, Berlin/Heidelberg, Springer (2011) 161–171
6. Barnett, M.: regex · pypi [online] (2021 [2021-10-28])