A Platform for Searching Texts for Desired Expressions in a User-Editable Pattern Matching Environment for Language Learning

Tatsuya Katsura
Graduate School of Environmental, Life, Natural Science and Technology
Okayama University
Okayama, Japan
pcn05fd3@s.okayama-u.ac.jp

Koichi Takeuchi
Faculty of Environmental, Life, Natural Science and Technology
Okayama University
Okayama, Japan
takeuc-k@okayama-u.ac.jp

***Abstract*—In this paper we propose a platform of pattern matching system that can extract required phrases or sentences in texts. Finding certain expressions in texts are often needed in language learning, e.g., examples of case markers between a predicate and its argument, or possible nouns in subject of a verb in a certain meaning. In previous studies, several types of systems, containing concordancers, are proposed. However, it is not easy to apply combined patterns because the pattern matching templates are previously fixed. Thus, we propose a flexible phrase searching system in which users can create search patterns by combining blocks of basic search templates. One of the characteristics the proposed system is that user can also specify where to be highlighted in texts with the blocks. To realize the function of combining patterns by users, the proposed system employs Prolog as the base of the data structure. The platform of the searching system is implemented on an Web server with JavaScript-based interface and database system. In the performance test, we show that the proposed system can deal with relatively large scale texts (10,000 sentences), and also demonstrate the combined patterns can be applied to the texts. In this paper we discuss the system architecture and the extendability of the pattern matching.**

***Index Terms*—pattern matching, concordancer, browser-based pattern matcher, Prolog**

I. Introduction

Extraction of some phrases and expressions in texts is considered essential function in language education. For example, case markers located between a predicate and its object in Japanese language are various rules and alternations[1], then language learners need to search for examples of predicate-argument examples in Japanese texts. As a tool for searching texts, various kinds of concordancer are proposed, however, most concordancers have the limitations in the use of their functionality. For example, Sketch Engine[2], one of the famous concordancers, proposes Corpus Query Language (CQL)[3] that has rich templates of pattern matching for words and characters, however, most of the templates are mainly effective for English and the templates are realized at the level of regular expressions. Thus, users are unable to employ patterns at the level of Context Free Grammar (CFG) which incorporates dependency parsing or predicate-argument relations. From the viewpoint of Natural Language Processing (NLP) research, dependency parsers (e.g., KNP[4], CaboCha[5], GiNZA[6]) have already been developed and are available, but it is not easy to build a pattern matching system with NLP tools from scratch.

Thus, we propose an environment that users can combine patterns searching for phrases or expressions in texts without programming. The system allows users to combine the basic search templates connected with Prolog predicates that have all of the information about dependency, part of speech (POS), and lemmas of sentences analyzed by NLP tools. The combination of search templates are consistently composed and the combined search pattern works to find the intended phrases in texts thanks to the Prolog inference engine. The system has a browser-based interface based on Blockly[7] that allows users to combine patterns in a visual environment.

The contributions of this study are as below: 1) we propose a new type pattern matching system providing the environment in which users can combine patterns, 2) Prolog-based search templates allow users flexibility of pattern combinations, and 3) browser-based system that can be suitable for non-programmers. In this paper we focus on Japanese pattern matching system because most of the concordancers are build for English, and the target of the system is not only for language learners but also for analyzing texts to check the student essays.

In the performance test, we shows that the proposed system can deal with relatively large scale texts (10,000 sentences), and also demonstrate the combined patterns can be applied to

[1]The verbs *morau* (get) and *eru* (obtain) are almost the same meaning but *morau* has alternation between the case markers *ni/kara* as *kare ni/kara morau* ((I) get (it) from him.), and *eru* can only takes *kara* in this meaning as *kare kara/*ni eru*.

[2]https://www.sketchengine.eu/

[3]https://www.sketchengine.eu/documentation/corpus-querying/

[4]https://nlp.ist.i.kyoto-u.ac.jp/?KNP

[5]https://taku910.github.io/cabocha/

[6]https://megagonlabs.github.io/ginza/

[7]https://developers.google.com/blockly?hl=ja

the texts. In this paper we discuss the system architecture and the flexibility of the pattern matching.

II. Related Work

One of the famous concordancers is Sketch Engine. Sketch Engine provides an environment that user can search for inflections, lemmas, words, part-of-speeches (POSes), and more with the CQL that works at the level of regular grammar. Sketch Engine offers a Japanese search environment as well[8]; however, the search templates based on Japanese morphological analyzers appear a little simple and do not cover dependency parsing

Other than concordancers, a corpus management system ChaKi.NET [1] has been constructed for Japanese dependency structure annotated corpora. Chaki.NET provides several search templates that can capture dependency structure; but the functions are limited and aimed at annotation task. Thus, there is no function such as combining patterns.

As a tool that provides search function for NINJAL Parsed Corpus of Modern Japanese (NPCMJ) [9] [2], NPCMJ Explorer[10] has been developed. The Explorer is implemented based on Tregex [3] that can extract designated phrases or words form parsed tree structure[11]. The possible search patterns are effective and the several combinations of pattern are available; however, the users are required to build a search formula to capture the desired expressions[12]

In NLP study, StruAP [4] which is a pattern matching tool for parsed trees is proposed. StruAP can designate complex combination of nodes in syntax trees and has a Web-based interface. The patterns, however, should be defined by the pattern language, and the tool is a commercial product and not publicly available. Thus, we cannot use the tool.

III. Platfrom of Pattern Matching System

To realize the environment that users can edit patterns for extracting desired expressions with visual operations, the system is composed of a front-end system and a back-end system. The front-end system provides a visual interface on which user can conduct all of the operations such as uploading text files, editing patterns and confirming the extracted results. On the other hand, the back-end system provides all of the text processing tasks such as saving the uploaded texts, analyzing texts with NLP tools, and returning results of search texts with the pattern requested from the front-end.

Since the back-end system is independent, it is possible to incorporate several NLP tools and use them from the front-end. The composed patterns by users can be saved and reused by the other users. Thus, the users can concentrate on creating patterns that match their requests, and obtain the search results.

This paper describes the development of a matching system for Japanese texts. The following sections provide detailed explanations of the Japanese text matching system.

A. Overview of the Pattern Matching System

Figure 1 shows that the overview of the proposed pattern matching system. The encircled numbers indicate the number of steps in the workflow of pattern matching. The front-end system is a JavaScript-based interface which has the functions of uploading texts, showing the results of analyzed texts, composing patterns, and showing the results of matched phrases with texts. The block-based environment is implemented using Blockly. For the management of composed patterns, the frond-end system has also the functions of downloading and uploading composed patterns.

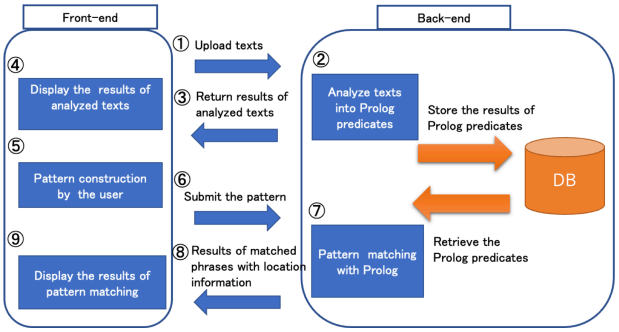

Fig. 1. Overview of the pattern matching system.

The back-end composed of the three modules, the Python-based NLP processing module, Prolog processing module implemented using SWI Prolog[13], and database system implemented using Elasticsearch[14].

According to the workflow, after texts are uploaded at the front-end, morphological analysis [15] and dependency parsing [16], and argument structure analysis[17] [5] are applied into the texts at the NLP module. By this processing, the texts are segmented into morphemes and chunks with grammatical and semantic tags. All of the data are sent to the Prolog module and then, they are converted to the Prolog predicates. The predicates are stored in the database at the database module.

After the user submits the search patterns, i.e., queries in Prolog format using block-based JavaScript, the system extracts Prolog predicates corresponding to each sentence from the database, and executes the pattern matching of Prolog on the predicates with the queries.

The results of pattern matching are sent from the back-end to the front-end. The matched expressions are displayed in red in the text. This function can be realized because the position

[8]https://www.sketchengine.eu/corpora-and-languages/japanese-text-corpora/

[9]https://npcmj.ninjal.ac.jp/index.html

[10]https://npcmj.ninjal.ac.jp/explorer/

[11]https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/trees/tregex/TregexPattern.html

[12]For example, the pattern "__ !< __" indicates extraction of words from the tree corpus. As this pattern shows, users are required to have enough knowledge of pattern definitions to build a combined pattern. Thus, frequently used patterns are already defined and available to users in NPCMJ Explorer.

[13]https://www.swi-prolog.org/

[14]https://www.elastic.co/jp/elasticsearch/

[15]https://taku910.github.io/mecab/

[16]https://taku910.github.io/cabocha/

[17]https://github.com/Takeuchi-Lab-LM/python_asa

information is recorded using Prolog predicates. The details of format of the predicates in this task are described in Section III-B.

B. Prolog-based Data Structure

The assumed demands of searching texts are extracting chunks, phrases and words with combination of constrains specifying dependencies of subjects and predicates, as well as particles and POSes. To meet these demands, texts are segmented to morphemes with lemmas and POSes using the morphological analyzer, and the morphemes are grouped into chunks added with dependency relations using the dependency parser. After the chunks are analyzed their predicate-argument relations, semantic roles [18] are assigned to the arguments of the predicates.

In order to capture and utilize the extensive information for search purposes, we define the predicates based on the relations between nodes. Figure 2 depicts a tree structure of a sentence with analyzed results using NLP tools.

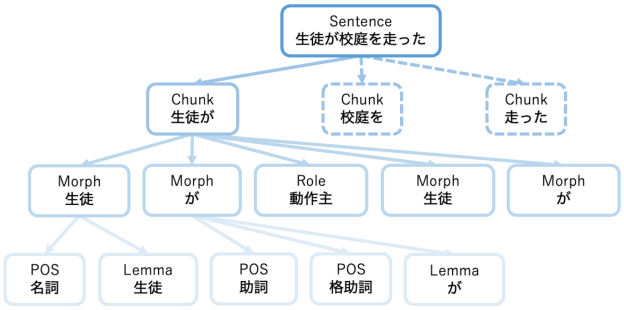

Fig. 2. Tree structure of a sentence with analyzed results using NLP tools.

Thus, we define the following Prolog predicates to capture the tree structure of the analyzed results as Table I.

TABLE I
PROLOG PREDICATES

Predicates	Descrption
chunk(SentID, 0, ChunkID)	chunk IDs
morph(SentID, ChunkID, MorphID)	morpheme IDs
main(SentID, ChunkID, String)	main morpheme in the chunk
part(SentID, ChunkID, String)	particle in the chunk
role(SentID, ChunkID, String)	semantic role in the chunk
semantic(SentID, ChunkID, String)	sense of the predicate
surf(SentID, NodeID, String)	surface string for the node ID
surfBF(SentID, MorphID, String)	base form or lemma
sloc(SentID, Morph/ChunkID, Position)	position of the chunk
pos(SentID, MorphID, String)	part of speech for the morpheme
dep(SentID, ChunkID, ChunkID)	dependent relation between chunks

In the table, the *NodeID* in the predicate *surf* denotes morpheme ID, chunk ID, and 0 that indicates sentence node. The morpheme IDs and chunk IDs are set so that they do not overlap, Thus, their IDs are unique. The morpheme ID starts after the last number of chunk. Then the predicate *surf* can record a surface string at any node.

In the predicate *sloc*, the argument *Position* denotes the position that consists of start and end positions that express number of characters from the beginning of the sentence. For example, the position of the first chunk "生徒が" in Figure 2 is "0_2". This provides the position of the string that are matched to the pattern.

C. Example of Query

In the previous section, we defined the predicates in database for recording tree structure. Using the predicates defined in the previous section, users can build a search query. All of the predicates are connected to the blocks in Blockly. Figure 3 shows that an example of prolog-based query to extract predicates and their arguments that have dependency relation with the case marker "を" (accusative case).

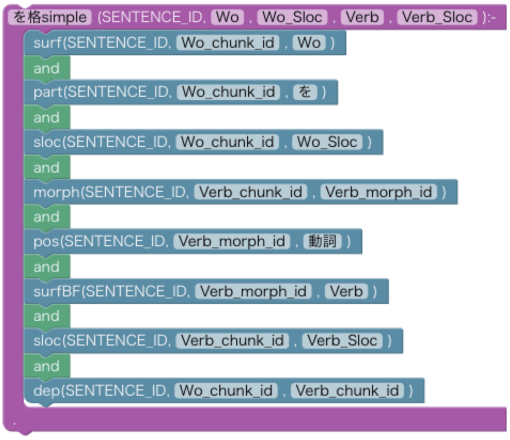

Fig. 3. Example of a search query: to extract predicates and arguments that has dependency relation with the case marker "を" (accusative case).

In Figure 3, the dark blue blocks denote the prolog predicates that are defined in the database. The purple block denotes the user defined predicates. Since all of the blocks follow the Prolog format, they can be combined to make more complicate patterns based on the user define blocks.

All of the search results are recorded to the variables whose name is starting a capital character. In the query of Figure 3, the arguments *Wo* and *Verb* are intended to extract source of dependent and its verb, respectively. The *Wo_Sloc* and *Verb_Sloc* denotes the position information and the system will highlight the parts when the user designates the *_Slock* variable. All of the predicates such as *pos*, *part* and *morph* works a conjunctions. For example, the predicate "pos(SENTENCE_ID, Verb_morph_id, 動詞)" indicates that the predicate *pos* has three arguments and the third arguments are fixed as "動詞" (verb). Thus, this predicate matches only those whose POSes are verbs; then the variables

[18]We currently implemented extended thematic roles [5], [6] such as 動作主 (Agent), 対象 (Theme), 時間 (Goal) and so on. The details are in Predicate Thesaurus https://pth.cl.cs.okayama-u.ac.jp/.

SENTENCE_ID and *Verb_morph_id* will record the IDs by unification.

In the displaying the results, the system has three display modes, i.e., table, highlight and keyword in context (KWIC). Figure 4 shows the results of matched chunks with table format.

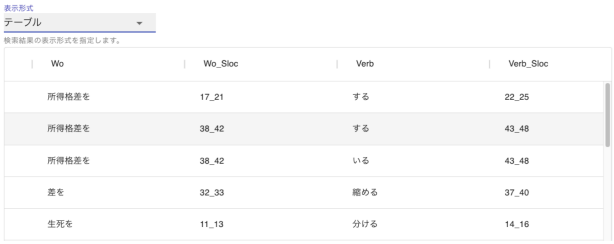

Fig. 4. Displaying results of pattern matching: all variables.

In the table, the variables *Wo* and *Wo_Sloc* are extracted independently. Thus, the user can designate which words or chunks should be highlighted in the highlight mode using the *_Sloc* variable (Figure 5).

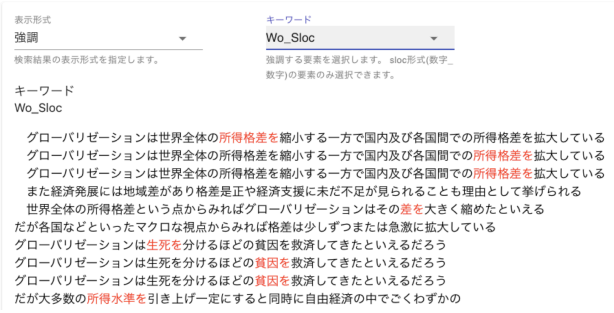

Fig. 5. Displaying the results of pattern matching: highlighting the argument *Wo_Sloc*.

Figure 5 shows the highliting *Wo_Sloc* parts in the texts[19]. The user can also see the highighting of the verb variables by selecting the *Verb_Sloc* (Figure 6).

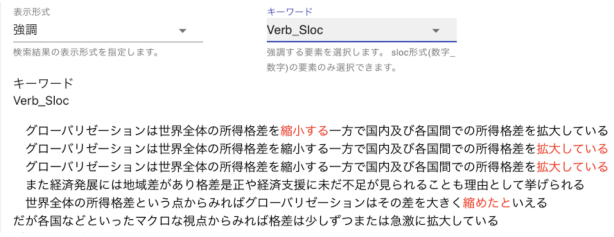

Fig. 6. Displaying the results of pattern matching: highlighting the argument *Verb_Sloc*.

The flexibility of highlight function displaying must be an advantage for checking students' essays as well as learning languages.

[19]The duplicated sentences are displayed due to the bidirectional dependencies settings.

IV. Performance Tests

The system stores all of the texts to database in Elasticsearch. Thus, the system is expected to work for the large scale texts; however, the system works rather slow because the current implementation does not focus on speed but focuses on achieving consistent behavior. As a performance test experiment, we evaluate the processing time when inputting a little large size of text, such as 10,000 lines.

As a result of the experiment, it took about six minutes to upload the text. The task of upload consists of sending texts to servers, applying NLP tools to the texts, converting all information to Prolog predicates and storing them to the database. The time to run the pattern match on the text was about five minutes. The task of pattern match consists of extracting each predicates corresponding to the sentence, executing Prolog matching, and sending all of the results from the back-end to the front-end. Thus, the system works on the 10,000 lines of text but it takes a little long time to run the pattern matching.

V. Discussions and Conclusions

The proposed system can extract required phrases or sentences in texts with the user-editable pattern matching environment. The function of consistent pattern combinations depends on the Prolog format. While the pattern combination is flexible, it can be somewhat difficult for users to build patterns because users may be not familiar with the Prolog format.

As described in Section III-C, the system requires special variables with *_Sloc* suffix that indicate the position of the target expressions. This feature have both advantages and disadvantages. The position variable provides a flexibility of selection of highlight part, however, it may seem to be redundant for users who want easier operation. Thus, it may be necessary to prepare a predicate that hides position variable.

In the current implementation, the patching system does not accept ambiguous matching such as "*"; but the SWI-Prolog provides a function to deal with ambiguous matching of regular expressions. This must be an issue for the future.

Acknowledgments

A part of this research is supported by JSPS KAKENHI (Grant Number 22K00530).

References

[1] Masayuki Asahara, Yuji Matsumoto, and Toshio Morita. Demonstration of chaki.net – beyond the corpus search system. In *Proceedings of of COLING 2016, the 26th International Conference on Computational Linguistics: System Demonstrations*, pages 49–53, 2016.

[2] Stephen Wright Horn, Alastair Butler, Iku Nagasaki, and Kei Yoshimoto. Derived mappings for FrameNet construction from a parsed corpus of Japanese. In *LREC 2018 Proceedings, International FrameNet Workshop, 11th edition of the Language Resources and Evaluation Conference*, pages 28–32, Miyazaki, Japan, 2018.

[3] Roger Levy and Galen Andrew. Tregex and tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of the th International Conference on Language Resources and Evaluation (LREC 2006)*, pages 2231–2234, 2006.

[4] Kohsuke Yanai, Misa Sato, Toshihiko Yanase, Kenzo Kurotsuchi, Yuta Koreeda, and Yoshiki Niwa. StruAP: A Tool for Bundling Linguistic Trees through Structure-based Abstract Pattern. In *Proceedings of the 2017 EMNLP System Demonstrations*, pages 31–36, 2017.

[5] Koichi Takeuchi, Suguru Tsuchiyama, Masato Moriya, Yuuki Moriyasu, and Koichi Satoh. Verb Sense Disambiguation Based on Thesaurus of Predicate-Argument Structure. In *Proc. of the International Conference on Knowledge Engineering and Ontology Development*, 2011. 208–213.

[6] C. J. Fillmore. *The Case for Case*, pages 1–89. New York: Holt, Rinehart, and Winston, 1968.