Information Engineering Express
International Institute of Applied Informatics
2026, Vol. 12, No. 1, IEE863

Adaptable Expression Search Framework with Customizable Pattern Matching for Language Studies

Tatsuya Katsura *, Koichi Takeuchi *

Abstract

This study introduces a novel design for a pattern matching system capable of extracting select words or phrases from texts. In the process of learning a foreign language, searching for instances of usage or grammatical structures within texts is a common requirement. While numerous systems, particularly concordancers, have been proposed in prior research, many of them lacked flexibility and posed challenges when attempting to combine specific search patterns. To address this limitation, we developed a new phrase search system that allows users to craft their search patterns by merging basic search templates. This paper presents a system that leverages Prolog predicates as a fundamental data structure, utilizing SWI-Prolog for processing. The system is capable of performing searches that integrate regular expressions with other combined patterns. Our performance test demonstrates the system can process 10,000 sentences without errors. User evaluation employing system usability scale indicates that while the current usability of our system requires enhancement, the feedback gathered from these evaluations not only confirms the system's robustness but also provides valuable insights for future improvements.

Keywords: pattern matching, concordance, browser-based pattern matcher, Prolog

1 Introduction

The extraction of phrases and expressions from texts is an essential function in language education. For example, in the Japanese language, case markers between a predicate and its object adhere to various rules and alterations. Language learners, therefore, need to search for predicate-argument examples in Japanese texts. To aid in text searching, several concordancers have been proposed. However, most of these concordancers have limitations in functionality. Take Sketch Engine [1, 2][1], a wellknown concordancer, for instance. It offers Corpus Query Language (CQL) [3, 4][2], which provides rich pattern matching templates for words and characters. Yet, most of these templates are mainly effective for English and are implemented at the level of regular expressions. Consequently, users can't utilize patterns at the level of Context-Free Grammar (CFG), which includes dependency parsing or predicateargument relations. From the perspective of Natural Language Processing (NLP) research, while dependency parsers such as

* Okayama University, Okayama, Japan
[1] https://www.sketchengine.eu/
[2] https://www.sketchengine.eu/documentation/corpus-querying/

KNP [5] [3], CaboCha [6][4], and GiNZA[5] have been developed and are available, building a pattern matching system from scratch using NLP tools is not straightforward.

Therefore, we propose a user-friendly environment where non-programmers can create and combine patterns for searching phrases or expressions in texts. Our system allows users to merge basic search templates connected to Prolog predicates containing all information about dependency, parts of speech (POS), and lemmas from sentences analyzed by NLP tools. These combined search templates are consistently structured, and the synthesized search pattern effectively locates the desired phrases in texts, thanks to the Prolog inference engine. The system features a browser-based interface built on Blockly[6], enabling users to visually combine patterns.

This study makes the following contributes[7]: 1) we introduce a pattern matching system that employs Prolog-based search templates for flexible pattern combinations, 2) the system can match patterns by combining regular expressions with other type of patterns, and 3) we conducted a user evaluation experiment to assess the effectiveness of the proposed system. This paper primarily focuses on the Japanese pattern matching system, as most concordancers are developed for English. The system is aimed not only at language learners but also at other NLP related tasks, e.g., searching expressions in text mining tools.

In our performance test, we demonstrate that the proposed system can handle large-scale texts (up to 10,000 sentences) and effectively apply combined patterns to these texts. User evaluation employing system usability scale indicates that while the current usability of our system requires enhancement, the feedback gathered from these evaluations provides valuable insights for future improvements. We discuss the system architecture, the flexibility of pattern matching including regular expressions, and evaluations of the system in this paper.

2 Related Works

Sketch Engine[8], a well-known concordancer, offers an environment where users can search for inflections, lemmas, words, part-of-speeches (POSes), and more using the Corpus Query Language (CQL), which operates at the level of regular grammar. While Sketch Engine also supports Japanese text searches[9], the templates based on Japanese morphological analyzers are somewhat basic and lack dependency parsing capabilities.

In addition to concordancers, there's ChaKi.NET [8], a corpus management system specifically designed for Japanese dependency structure annotated corpora. ChaKi.NET offers several search templates capable of capturing dependency structures, but its functionality is mainly geared towards annotation tasks, lacking features like pattern combination.

[3] https://nlp.ist.i.kyoto-u.ac.jp/?KNP
[4] https://taku910.github.io/cabocha/
[5] https://megagonlabs.github.io/ginza/
[6] https://developers.google.com/blockly?hl=ja
[7] A preliminary version of this work was published as a conference paper [7].
[8] https://www.sketchengine.eu/
[9] https://www.sketchengine.eu/documentation/corpus-querying/

Another tool, NPCMJ Explorer[10], provides search capabilities for the NINJAL Parsed Corpus of Modern Japanese (NPCMJ) [9, 10][11]. It is built on Tregex [11], a tool that can extract specific phrases or words from a parsed tree structure[12]. While NPCMJ Explorer's search patterns are effective and allow for various combinations, users must construct search formulas to find specific expressions.

In the field of NLP, StruAP [12] has been proposed as a pattern matching tool for parsed trees. StruAP enables the complex combination of nodes in syntax trees and comes with a web-based interface. However, patterns must be defined using a specific pattern language, and it is not accessible for general use since StruAP is a commercial product that's not publicly available.

3 Framework for Pattern Matching System

Our system is architected into two primary modules: the front-end, which offers users an intuitive interface for tasks like file uploads, pattern modifications, and result validations; and the back-end, which manages all textual operations, including storing files, processing them with NLP technologies, and returning search outcomes based on front-end directives. The independent nature of the back-end allows for seamless integration with multiple NLP resources, accessible via the front-end. Thanks to the base function of Blockly, patterns composed by users can be saved as an XML format file. This allows the other users to repurpose the patterns. This paper elucidates the formulation of a matching framework specialized for Japanese textual content.

3.1 Overview of the Pattern Matching Framework

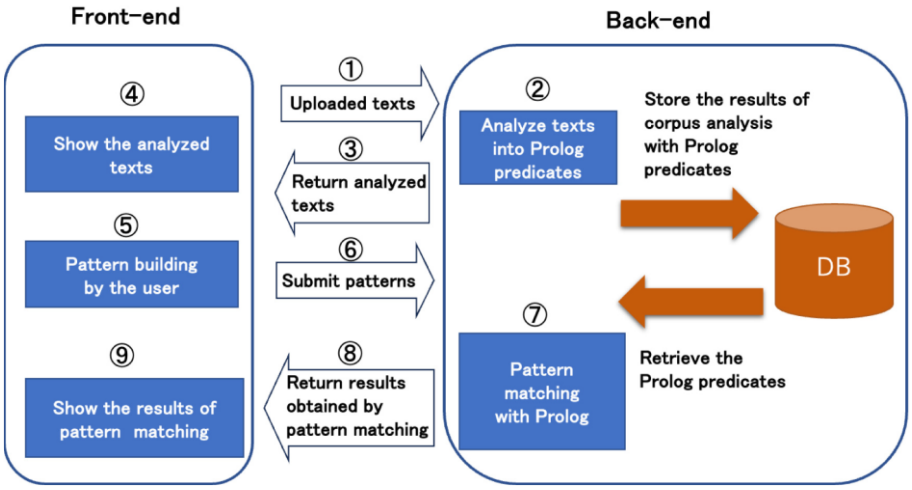


Figure 1: Overview of the pattern matching framework.

[10] https://npcmj.ninjal.ac.jp/explorer/
[11] https://npcmj.ninjal.ac.jp/index.html
[12] https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/trees/tregex/

The pattern matching system depicted in Figure 1 is outlined, with numbered circles highlighting its various components. The system's back-end is structured into three key modules: a Python-based NLP processing module, a Prolog processing module utilizing SWI-Prolog [13][13], and a database system developed with Elasticsearch. The process begins with text uploads at the front-end, which are then subjected to morphological analysis[14], dependency parsing[15], and argument structure analysis within the NLP module[16] [14]. This analysis breaks down the text into morphemes and chunks, each labeled with grammatical and semantic tags. This information is forwarded to the Prolog module, transformed into Prolog predicates, and stored in the database module.

Users input search patterns as queries in Prolog format through block-based JavaScript. The system retrieves the corresponding Prolog predicates for each sentence from the database and conducts pattern matching against the queries. The outcomes of this pattern matching are relayed from the back-end to the front-end, where matched phrases are highlighted in red in the text[17]. This highlight function can be built by recording the character positions of the matched parts with Prolog predicates. For additional information on the predicate format used in this system, we show two examples of pattern matching in Section 3.3.

3.2 Functionalities of Prolog-based Predicates

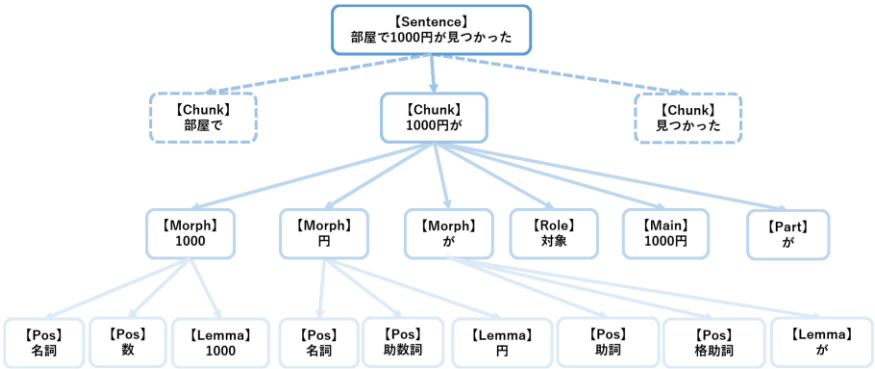


Figure 2: Visualization of a Japanese sentence analysis (e.g. "部屋で 1000 円が見つかった") using NLP tools (with English translation: "1000 yen bill was found in the room").

The anticipated requirements for textual searches encompass the extraction of chunks, phrases, and individual words, all while adhering to constraints that define dependencies between subjects, predicates, particles, and POSes. In response to these needs, texts undergo segmentation into morphemes, further characterized by lemmas and POSes through the use of a morphological analyzer. These morphemes are then assembled into chunks, enriched with dependency relationships generated by the dependency parser. After this assembly, the chunks are subjected to an analysis of their predicate-argument relation types, leading to the assignment of semantic roles[18] to the arguments linked to the predicates.

[13] https://www.swi-prolog.org/
[14] https://taku910.github.io/mecab/
[15] https://taku910.github.io/cabocha/
[16] https://github.com/Takeuchi-Lab-LM/python_asa
[17] The examples are shown in Figure 5.
[18] We currently implemented extended thematic roles [14, 15] 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/.

To effectively harness this wealth of information for search endeavors, we establish predicates that describes the interrelations among nodes. Figure 2 depicts a visual representation of a sentence, analyzed and represented using NLP tools.

Table 1: Prolog Predicates.

Predicates	Description
chunk(Sentence_ID, 0, Chunk_ID)	chunk IDs
morph(Sentence_ID, Chunk_ID, Morph_ID)	morpheme IDs
main(Sentence_ID, Chunk_ID, String)	main morpheme in the chunk
part(Sentence_ID, Chunk_ID, String)	particle in the chunk
role(Sentence_ID, Chunk_ID, String)	semantic role in the chunk
semantic(Sentence_ID, Chunk_ID, String)	sense of the predicate
surf(Sentence_ID, Node_ID, String)	surface string for the node ID
surfBF(Sentence_ID, Morph_ID, String)	base form or lemma
sloc(Sentence_ID, Morph/Chunk_ID, Position)	position of the chunk
pos(Sentence_ID, Morph_ID, String)	part of speech for the morpheme
dep(Sentence_ID, Chunk_ID, Chunk_ID)	dependent relation between chunks
re_match(Regex_Pattern,String)	regular expression pattern matching

Building on this, we define specific Prolog predicates that encapsulate the tree configuration of these analyzed outcomes, as shown in Table 1. Within this table, the *Node_ID* parameter within the surf predicate signifies the morpheme ID, the chunk ID, and a "0", which corresponds to the sentence node. An intricate design ensures that morpheme IDs and chunk IDs never coincide, ensuring their uniqueness upon them. The morpheme ID numbering commences subsequent to the concluding chunk number, ensuring the "surf" predicate can capture a surface string at any given node.

Navigating further into the sloc predicate, the Position argument represents a span, demarcated by start and end coordinates, indicative of the character count from the begin of sentence. To illustrate, in Figure 2, the chunk "彼が" possesses a position of "0_1". This facilitates pinpointing the string's exact position that aligns with the pattern. The predicate re_match can further specify the string to be extracted in the regular expression. For example, to specify the amount of money (yen) to be extracted from the variable YEN in the 4-digit number, you can define the predicate 're_match("^[1-9][0-9]{3}",YEN)'.

3.3 Example of Query

In the previous section, we established the predicates within the database to capture the tree structure. Leveraging these previously specified predicates, users are able to construct search queries. Each of these predicates is linked to the blocks found in Blockly, facilitating an integrated query-building process. This section provides examples of two queries that can be created with this system.

3.3.1 *Query of extracting expressions containing case particles*

Users can extract expressions from sentences by focusing on parts of speech and the semantic roles of words. Figure 3 shows that an example of prolog-based query to extract predicates and their arguments that have dependency relation with the case marker "が" (nominative case).

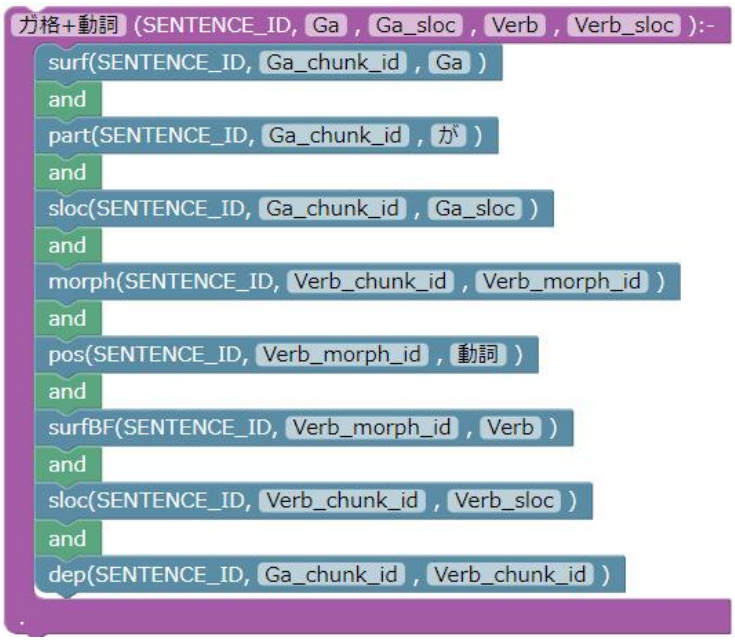
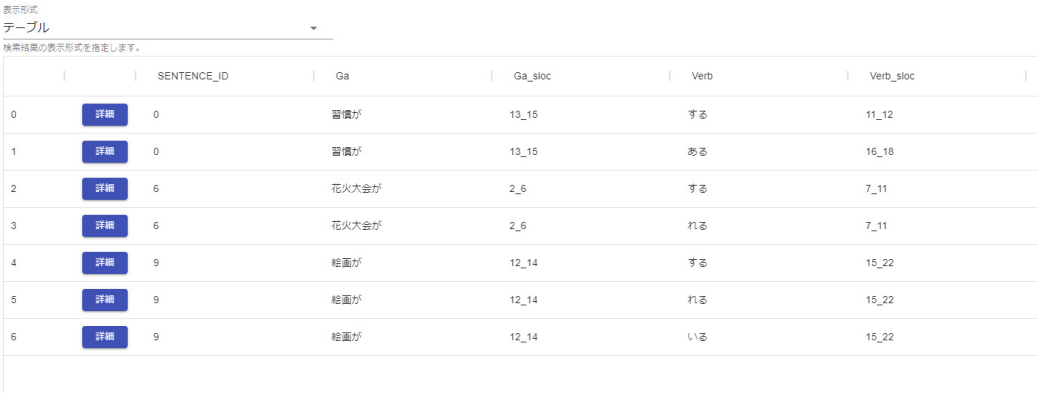


Figure 3: Example of a search query: to extract predicates and arguments that has a dependency relation with the case marker "が" (nominative case).

In Figure 3, dark blue blocks represent Prolog predicates set in back-end. The purple block represents a predicate defined by the user. Since all blocks conform to the Prolog format, the user-defined building blocks can be used and interlinked to form complex patterns.

Each search result is stored in variables that begin with a capital letter. In the query shown in Figure 3, the arguments Ga and Verb are used to extract the dependent source and its corresponding verb, respectively. Ga_sloc and Verb_sloc provide the location information, and the system highlights these sections when the user selects the _slock variable. All predicates, including pos, part, and morph, function as conjunctions. For example, the predicate "part(SENTENCE_ID, Ga_chunk_id, が)" means that pos has three arguments with the third one specifically set to "が" (nominative case), which only aligns with instances where particle in the chunk has the word "が", and the variables "SENTENCE_ID" and "Verb_chunk_id" are used to store the IDs through a process of unification.

When it comes to result representation, the system has three display formats: table, highlighted, and keyword-in-context (KWIC). Figure 4 shows the chunk matches in a table formats. Within this table, defined variables Ga and Ga_sloc are displayed independently. This feature allows users to specify the words or chunks to highlight during highlight mode, as demonstrated in Figure 5.


表示形式
テーブル
検索結果の表示形式を指定します。

		SENTENCE_ID	Ga	Ga_sloc	Verb	Verb_sloc
0	詳細	0	習慣が	13_15	する	11_12
1	詳細	0	習慣が	13_15	ある	16_18
2	詳細	6	花火大会が	2_6	する	7_11
3	詳細	6	花火大会が	2_6	れる	7_11
4	詳細	9	絵画が	12_14	する	15_22
5	詳細	9	絵画が	12_14	れる	15_22
6	詳細	9	絵画が	12_14	いる	15_22

Figure 4: Displaying the results of pattern matching for all variables: the resulting table lists the extracted variables, including the nominative argument "*Ga*"(e.g. "習 慣が"), the resulting predicate "*Verb*" (e.g. "ある"), and their corresponding character position markers "*Ga_sloc*"(e.g. "13_15") and "*Verb_sloc*"(e.g. "11_12") which specify text spans in the source sentences.

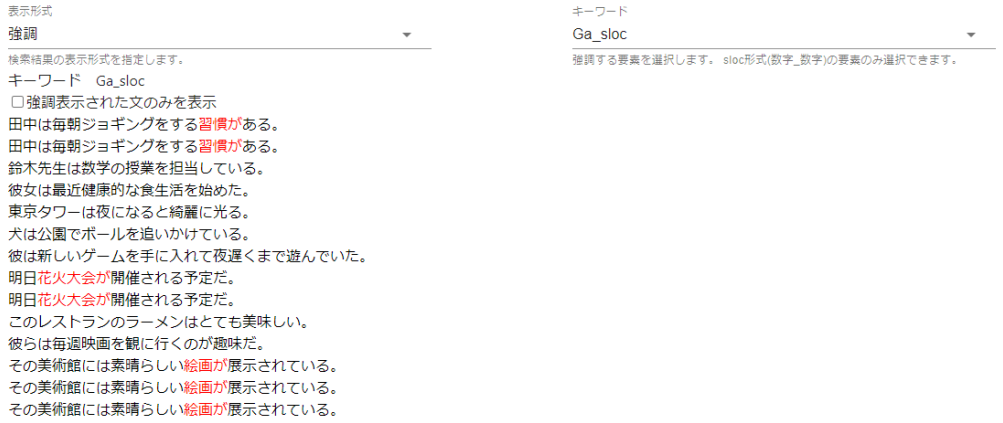

表示形式
強調
検索結果の表示形式を指定します。
キーワード　Ga_sloc
☐強調表示された文のみを表示
田中は毎朝ジョギングをする習慣がある。
田中は毎朝ジョギングをする習慣がある。
鈴木先生は数学の授業を担当している。
彼女は最近健康的な食生活を始めた。
東京タワーは夜になると綺麗に光る。
犬は公園でボールを追いかけている。
彼は新しいゲームを手に入れて夜遅くまで遊んでいた。
明日花火大会が開催される予定だ。
明日花火大会が開催される予定だ。
このレストランのラーメンはとても美味しい。
彼らは毎週映画を観に行くのが趣味だ。
その美術館には素晴らしい絵画が展示されている。
その美術館には素晴らしい絵画が展示されている。
その美術館には素晴らしい絵画が展示されている。
キーワード
Ga_sloc
強調する要素を選択します。 sloc形式(数字_数字)の要素のみ選択できます。

Figure 5: Displaying the results of pattern matching: highlighting the text spans in the source sentence that correspond to the variable "*Ga_sloc*".

Figure 7 highlights the *Ga_sloc* segments within the text. Users also have the latitude to illuminate verb variables by selecting the *Verb_sloc*, as showcased in Figure 6.

Such adaptability in the highlighting feature, allowing users to emphasize specific words or chunks, is a variable tool for both text mining as well as language learning.

Tanaka has the habit of jogging every morning.
Tanaka has the habit of jogging every morning.
Mr Suzuki is in charge of mathematics classes.
She has recently started eating healthy.
Tokyo Tower shines beautifully at night.
The dog is chasing a ball in the park.
He got a new game and played until late at night.
There is a fireworks display tomorrow.
There is a fireworks display tomorrow.
The ramen at this restaurant is very delicious.
Their hobby is going to see English every week.
The art museum has wonderful paintings on display.
The art museum has wonderful paintings on display.
The art museum has wonderful paintings on display.

Figure 6: Displaying English translations of the texts shown in Figure 5: highlighting the spans corresponding to "*Ga_sloc*" in translated texts.

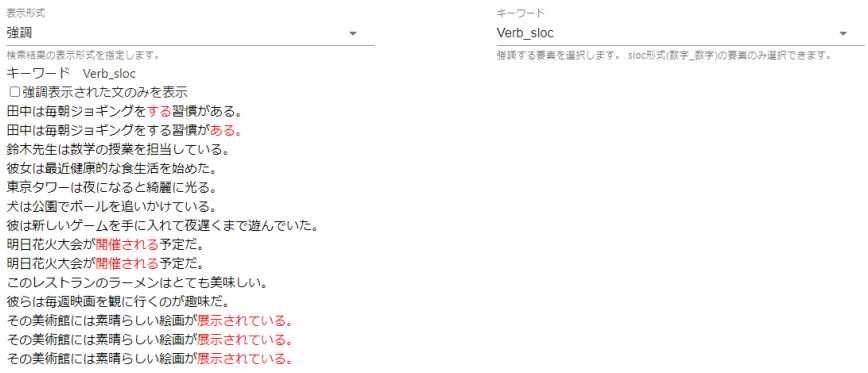


Figure 7: Displaying the results of pattern matching: highlighting the text spans in the source sentence that corresponds to the variable "*Verb_sloc*".

Tanaka has the habit of jogging every morning.
Tanaka has the habit of jogging every morning.
Mr Suzuki is in charge of mathematics classes.
She has recently started eating healthy.
Tokyo Tower shines beautifully at night.
The dog is chasing a ball in the park.
He got a new game and played until late at night.
There is a fireworks display tomorrow.
There is a fireworks display tomorrow.
The ramen at this restaurant is very delicious.
Their hobby is going to see English every week.
The art museum has wonderful paintings on display.
The art museum has wonderful paintings on display.
The art museum has wonderful paintings on display.

Figure 8: Displaying English translations of the texts shown in figure 7: highlighting the spans corresponding to "*Verb_sloc*" in translated texts.

This is especially useful in situations where understanding the full sentence, along with the emphasized words, is important for grasping the context.

3.3.2 Query of extracting specific expressions containing units with regex expressions.

User can further specify the expressions to be extracted from the text using regular expressions. Figure 9 shows that an example of prolog-based query that extracts expressions containing the unit "円" (yen) for the amount of money to be extracted in a 4-digit range.

The query shown in Figure 9 defines the argument Yen and the argument *Yen_sloc* that provides its location information. *Yen_sloc* provides the location of expressions containing the unit "円"(yen), and the system highlights these sections when the user selects the "*_sloc*" variable.

For example, "pos(SENTENCE_ID,YEN_MORPH_ID, 助数詞)" means that "pos" has three arguments with the third one specifically set to "助数詞" , which also only aligns with instances where POSes are counters for various categories, and the variables "SENTENCE_ID" and "YEN_MORPH_ID" are used to store the IDs through a process of unification. In addition, Figure 9 shows the deep orange block containing 're_match("^[1-9][0-9]{3}円",YEN)', where the second argument, *YEN*, is used to extract 4-digit numbers followed by the word "円".

Examples of the search results are shown in Figures 10 and 11 below, in KWIC and table formats, respectively.

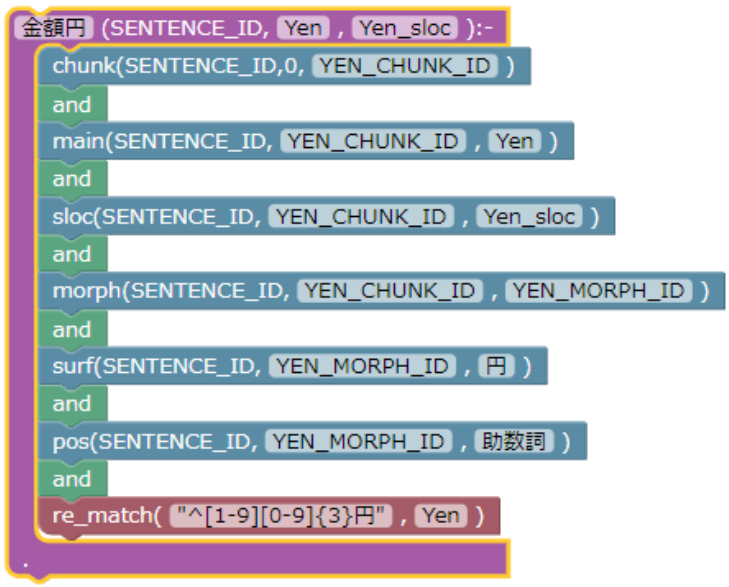


Figure 9: Example of search query: extracting expressions containing the unit "円"(yen) for the amount of money to be extracted in 4-digit numbers.

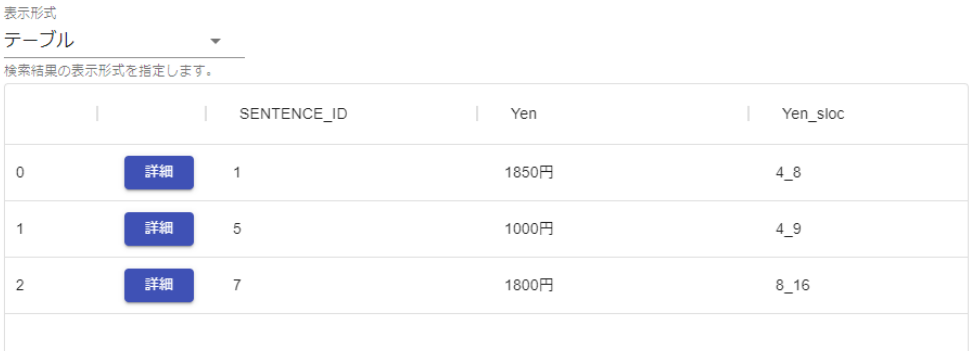


Figure 10: Example of search query: displaying the results of pattern matching for all variables. The extracted monetary arguments *Yen_sloc* in the unit "円" (yen) (e.g., "1850 円") within 4-digit numbers and their corresponding character position markers "*Yen_sloc*"(e.g. "4_8") which specify text spans in the source sentences.

Figure 11: Displaying the results of pattern matching: displaying the keyword "*Yen_sloc*" in a context.

Lunch cost 1850 yen.
I lost 1000 yen yesterday.
Tickets for the movie were 1800 yen.

Figure 12: Displaying English translations of the texts shown in Figure 11: highlighting the spans corresponding to "*Yen_sloc*" in translated texts.

4 Performance Test

All textual content within the system is archived in the Elasticsearch database. While the system is designed to handle extensive volumes of text, its current operational speed is relatively show. This is primarily because the present design prioritizes consistency and robustness rather than fast response. To verfify its robusteness, we conducted an evaluation using a relatively large input set of approximately 10,000 lines of text.

According to the test results, the system takes roughly six minutes to process this volume of text. The upload process entails sending the text to servers, applying NLP tools for textual processing, conveting the data into Prolog predicates, and subsequently storing them in the database. Executing a pattern match on this text, the system takes about five minutes. This matching process involves retrieving the specific predicates associated with each sentence, running the Prolog-based pattern matching and sending the results from the back-end to the user interface. In summary, while the system can handle 10,000 lines of text, the pattern matching step remains relatively time-consuming.

5 User Evaluation

We conduct a user evaluation experiment within our laboratory to assess the effectiveness of our system. Evaluators first receive explanations of the system through user manuals, instructional videos, and have time for a question-and-answer session, so that they are well-prepared to engage with the system effectively. The evaluators are then given a task searching for particular expressions within a selected text corpus, utilizing the system's capabilities to achieve this searching task. This task is intended to simulate real-world applications of the system in language education and text analysis. Finally, the evaluators are asked to complete a questionnaire that contains holistic evaluation and system usability scale [16] regarding the use of the system. The details are described in Section A. As results of the evaluation done by seven evaluators, the holistic score is 3.9 and SUS is 49. According to the previous study in SUS[19], the average SUS score from 500 studies is 68, and thus, the 49 is below the average. Although the usability of the proposed system needs improvement, the holistic score is close to 4, which is not too low. Since five of the seven evaluators have programming experience, it is conceivable that the score may have been high.

Feedback from the evaluators reveals several key points for improving the system usability. Notably, the operability and user interface of the system, particularly in relation to displaying analysis results and search outcomes, are identified as areas needing enhancement.

For instance, evaluators suggest the integration of a feature that would allow for easier acquisition of related blocks by simply clicking on elements within the analysis tree. Additionally, there is need to simplify how variables are defined within Prolog blocks, as the current requirement to explicitly specify elements like "*_sloc*" for highlighting is considered cumbersome.

[19] https://measuringu.com/sus/

Despite these challenges, the system's overall functionality is highly commended. The ability to combine patterns in a visual environment using Blockly-based interface and the flexibility offered by Prolog-based search templates are particularly appreciated. The evaluators show significant anticipation for future improvements in usability, which could further enhance the overall effectiveness of the system.

6 Discussions and Conclusion

The proposed system is adapted for extracting specific phrases or words from texts, leveraging a user-modifiable pattern matching framework. This system's ability to cohesively combine patterns is underpinned by the Prolog structure. Besides, thanks to SWI-Prolog, user can specify the expressions to be extracted from the text using regular expressions combined with the other patterns. Though this offers flexibility in pattern assembly, the Prolog format can pose challenges for users unfamiliar with its intricacies.

As elaborated in Section 3, our system necessitates the use of unique variables with "*_sloc*" that must be one of the reasons why the usability score is low. This design choice is a double-edged sword. On one hand, it provides users with the flexibility to select which segments should be highlighted. On the other side, this approach can introduce unnecessary complexity for those who prefer a more streamlined experience. Therefore, it may be benefical to desing a predicate that handles this positional variable.

As a future development, we improve the proposed system based on the suggested improvements. Additionally, we will enhance our user understanding by improving user manuals and explanatory videos.

Acknowledgement

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

A System Evaluation Metrics

We employ two types of evaluation metrics, the first items are holistic evaluations, i.e., we ask the evaluators to give scores regarding to the total goodness of the system[20]. By taking the average value, we use it as the user's holistic evaluation value.

The other evaluation is system usability scale (SUS). We instruct the evaluators to give scores from 1 (strongly disagree) to 5 (strongly agree) for 10 questions that consist of 5 positive and 5 negative ones. In terms of the 10 questions, we apply the standard version of the system usability scale[21]. The SUS for i-th question SUS_i is calculated by the following equation.

$$SUS_i = \big((ps_i - 1) + (5 - ns_i)\big) \times 2.5 \quad (1)$$

[20] We use two questions: "the functionality of this system was what I expected" and "overall I am satisfied with using this system".

[21] See https://en.wikipedia.org/wiki/System_usability_scale.

Where ps_i and ns_i denote scores of positive and negative i-th question, respectively. The SUS is the total of 10 questions of SUS_i. The perfect score is 100.

References

[1] Adam Kilgarriff and Pavel Rychly and Pavel Smrz and David Tugwell. The Sketch Engine. In *Proceedings of the Eleventh EURALEX*, pages 105–115, 2004.

[2] Adam Kilgarriff, Vít Baisa, Jan Bušta, Miloš Jakubíček, Vojtěch Kovář, Jan Michelfeit, Pavel Rychlý, and Vít Suchomel. The Sketch Engine: Ten Years On. *Lexicography*, 1:7–36, 2014.

[3] Oliver Christ and Bruno M Schulze. The IMS *Corpus Workbench: Corpus Query Processor (CQP) User's Manual*, 1994.

[4] Jakubíček, Miloš and Kilgarriff, Adam and McCarthy, Diana and Rychlý, Pavel. Fast Syntactic Searching in Very Large Corpora for Many Languages. In *Proceedings of the 24th Pacific Asia Conference on Language, Information and Computation*, pages 741–747, 2010.

[5] Daisuke Kawahara and Sadao Kurohashi. A Fully-Lexicalized Probabilistic Model for Japanese Syntactic and Case Structure Analysis. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 176–183, 2006.

[6] Taku Kudo and Yuji Matsumoto. Japanese Dependency Analysis using Cascaded Chunking. In *The 6th Conference on Natural Language Learning 2002 (CoNLL-2002)*, 2002.

[7] Tatsuya Katsura and Koichi Takeuchi. A platform for searching texts for desired expressions in a user-editable pattern matching environment for language learning. In *Proceeding of 2023 14th IIAI International Congress on Advanced Applied Informatics* (IIAI-AAI), pages 146–149, 2023.

[8] Masayuki Asahara, Yuji Matsumoto, and Toshio Morita. Demonstration of ChaKi.NET Beyond the Corpus Search System. In *Proceedings of the 26th International Conference on Computational Linguistics: System Demonstrations*, pages 49–53, 2016.

[9] National Institute of Japanese Language and Linguistics. *NINJAL Parsed Corpus of Modern Japanese*, 2016.

[10] Stephen Wright Horn, Iku Nagasaki, Alastair Butler, and Kei Yoshimoto. *Annotation Manual for the NPCMJ*. National Institute of Japanese Language and Linguistics, 2019.

[11] Roger Levy and Galen Andrew. Tregex and Tsurgeon: Tools for Querying and Manipulating Tree Data Structures. In *Proceedings of the fifth International Conference on Language Resources and Evaluation* (LREC 2006), pages 2231–2234, 2006.

[12] 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.

[13] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWIProlog. *Theory*

and Practice of Logic Programming, 12(1-2):67–96, 2012.

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

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

[16] John Brooke. USU – A Quick and Dirty Usability Scale. In *Usability Evaluation in Industry*, pages 189–194. Taylor and Francis, 1996.