Proceedings of the Federated Conference on Computer Science and Information Systems pp. 77–80
DOI: 10.15439/2019F151
ISSN 2300-5963 ACSIS, Vol. 18

Development of a Flexible Mizar Tokenizer and Parser for Information Retrieval System

Kazuhisa Nakasho
Yamaguchi University
in Yamaguchi
2-16-1, Tokiwa-dai, Ube City, Yamaguchi, Japan
Email: nakasho@yamaguchi-u.ac.jp

***Abstract*—In this paper, we explain the development of a new Mizar tokenizer and parser program as a component of a search system that works on the Mizar Mathematical Library. The existing Mizar tokenizer and parser can handle only an article as a whole written in the Mizar language, however, the newly developed program can deal with a snippet of a Mizar article.**

In particular, since it is possible to handle a snippet of an article without specifying a vocabulary section of an environment part, it is expected that user input efforts will be greatly reduced.

I. Motivation

THE AUTHOR is developing a new information retrieval system that works on the Mizar Mathematical Library (MML) [1]. In this paper, we explain a developed tokenizer and parser program of the Mizar language as a component of our search system.

A. MML Query

Currently, MML Query [2] developed by Grzegorz Bancerek in 2001 is widely used as an MML theorem search system. MML Query is the forerunner of the search systems for formalized mathematical libraries. Even today, it is the only active system that can search comprehensively large formalized mathematical libraries [3]. MML Query realizes pattern matching according to the grammatical structure of the Mizar language with its own language to specify a search object. This feature allows the users to input search patterns that have more expressive power than that of regular expressions. However, the users have to spend a considerable amount of time to learn the grammar of its own search language. Furthermore, since a mathematical theorem can be transformed into an infinite number of patterns by equivalent rewriting, it often causes retrieval omission in pattern matching. As mentioned above, MML Query has succeeded in reducing laborious retrieval work in the MML, however, there is still rooms for improvement.

B. Developing search system

In order to learn from the drawbacks of MML Query, the newly developing search system extracts features of the input data and compares them with that of theorems and definitions registered in the MML. For the feature comparison, we use a algorithm designed to output a logical distance between two expressions. In addition, the search history will be collected and reused as corpus data for machine learning so that the distance calculation algorithm will be tuned to fit user trend.

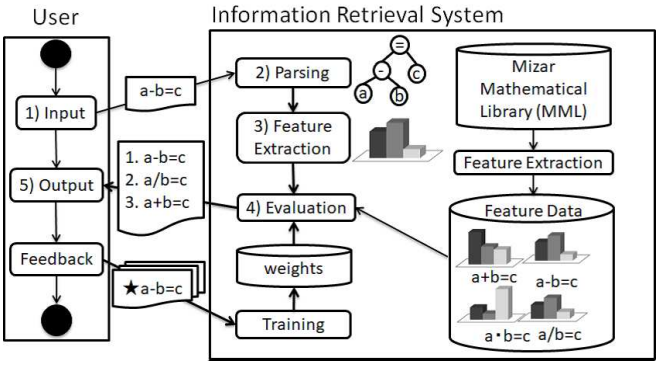

Fig. 1. Diagram of our information retrieval system

The flow of our search system is as follows:

1) A user inputs a search target such as a theorem in the Mizar language.
2) The search system parses the input data.
3) The search system calculates the features of the input data such as a number of occurrences and positional relationship of symbols and variables by analyzing the syntax tree.
4) The search system compares the above features with that of theorems and definitions registered in the MML, and ranks and displays the matching rates.
5) The user of the search system browses the search result displayed on the system.

Fig. 1 shows a diagram of our search system.

C. Necessity of a new tokenizer and parser

The developed tokenizer and parser program correspond to process 2) in Fig. 1. The existing parser used in the proof verification program of the Mizar system first reads an article and converts it into the Weakly Strict Mizar (WS-Mizar) language [4], [5]. In the WS-Mizar language, all terms are fully parenthesized, therefore, there is no ambiguity in operator precedence. After this process, the article in the WS-Mizar language is converted into XML intermediate representation by a parser program generated with Bison. The existing parser

program is supposed to handle full text of a Mizar article, therefore, it cannot process a snippet like a theorem, which is expected as the main input of our search system. That is the reason why we needed to develop a new tokenizer and parser program for the Mizar language.

II. Requirements of new tokenizer and parser

The Mizar language consists of a context sensitive grammar, and a set of valid symbols are determined according to Mizar articles enumerated in a vocabulary section of an environment part. However, in the construction of a Mizar article, it is said that the most difficult process is to create an environment part correctly. Therefore, it is not practical to enforce a search system user to input an environment part for every search. In this project, we aimed at constructing a tokenizer and parser program that works practically without an environment part. However, since the omission of an environment part may cause unexpected syntax errors, our search system needs to provide interfaces that enable the users to grasp and correct any syntax errors easily.

A. Tokenizer

As mentioned earlier, in the Mizar language, valid symbols are determined according to the Mizar articles enumerated in a vocabulary section. It means that a vocabulary section has an ability to determine word boundaries in lexical analysis. When a vocabulary section is omitted, our tokenizer extracts tokens according to the longest match rule on the assumption that every symbol registered in the MML is valid. As a result, a token that is not an originally valid symbol might be mistakenly recognized as a symbol. However, we succeeded in reducing token recognition errors by implementing special interpretation rules to recognize a token placed immediately after a certain keyword such as *let* or *reserve* as a variable.

B. Parser

Bison, which is used as a parser generator for the existing Mizar parser, adopts LALR parsing known as one of the most practical bottom-up parsing algorithms. Generally, bottom-up parser generators produce more efficient and smaller programs than top-down parser generators. However, since bottom-up parsers have difficulty in constructing a rough tree structure in the middle of parsing process, they sometimes tend to output meaningless error messages when grammatically incorrect input is given. The existing Mizar proof verification system also tends to output grammatical error messages that are difficult for beginners to understand. Based on these reasons and recent performance improvement of computer hardware, there have been increasing cases where top-down parsers such as LL parser and packrat parser are used in recent years. We also adopted ANTLR, which is based on Adaptive LL (*) parsing and known as one of the most powerful top-down parser generators, because incomplete input will often be given to our search system. ANTLR supports many output languages such as Java, C++, Python2, Python3, Go, Swift, JavaScript and C#. Whenever we need to develop Mizar tools that work on Web browser or modern editors such as Atom or Visual Studio Code, it can generate a parser written in JavaScript immediately.

III. Program specification

This section explains input, output and the flow of our tokenizer and parser program. Fig. 2 shows the flow chart of our program.

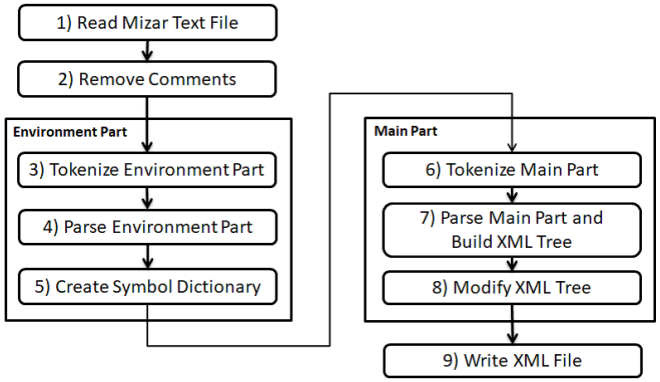

Fig. 2. Flow of our tokenizer and parser program

A. Input and output

Our program accepts not only full text of a Mizar article but also various types of blocks such as theorem, definition, registration, notation, and scheme as input data. Our program outputs a parsing result in XML format as well as the existing Mizar parser program. The output XML faithfully reproduces the structure of the official BNF grammar provided at `mizar.org`. Applying the official grammar rules to output XML will promote the secondary use of our tokenizer and parser program.

B. Tokenizer specification

In lexical analysis, when a vocabulary section is not specified, it is assumed that all the symbols defined in the MML are valid.

All the symbols in the MML are recorded in *mml.vct* attached to Mizar distribution binaries. In our tokenizer program, a symbol dictionary is built by extracting symbol information from *mml.vct* in a pre-processing step.

Our tokenizer program first removes comments. Next, it reads tokens such as symbols, keywords, numbers, identifiers, etc. from the left hand side according to the longest match rule. If the token matches a symbol registered in the symbol dictionary, our tokenizer program appends prefix *"_ _ «symbol type» «symbol priority» _"* to the token. Owing to the prefix, the following parser program is able to distinguish symbol types in the parsing process. Table I shows the correspondence between symbol types and their meanings in the Mizar language.

When a token is cut out according to the longest match rule on the assumption that all the MML symbols are valid, there

TABLE I
CORRESPONDENCE BETWEEN SYMBOL TYPES AND THEIR MEANINGS

Symbol type	Meaning
R	Predicate
O	Functor
M	Mode
G	Structure
U	Selector
V	Attribute
K	Left Functor Bracket
L	Right Functor Bracket

is a risk that a variable is misinterpreted as a symbol. For this reason, when a token comes immediately after a certain keyword such as *let* or *reserve*, our tokenizer regards the token as a variable identifier, and its symbol validity is temporarily turned off within the scope of the variable. Our program writes out token-separated text at the end of the process.

C. Parser specification

The official syntax of the Mizar language is written in BNF. We transformed the BNF syntax definition into ANTLR grammar form, then passed it to ANTLR parser generator. Normally, LL parsers require left recursion removal, although ANTLR automatically resolves direct left recursions. For this reason, we only needed to remove indirect left recursions. There are only two indirect left recursion in the Mizar official syntax definition. We repaired them and transformed the grammar rules from BNF to ANTLR grammar form.

The Mizar language has a feature that allows users to define prioritized infix operators (functors). While this feature has given a significant advantage of the readability of the Mizar language, it has also made lexical and syntactic analysis more difficult. Historically, this grammatical complexity has often become a bottleneck in the development of support tools for the Mizar system [6]. The existing Mizar system converts the Mizar language into the WS-Mizar language, thus all terms are parenthesized. Thanks to this process, the existing parser can avoid ambiguity in associativity and precedence of infix operators. In our program, its syntax tree structure is re-edited according to infix operator priorities in post-process. This strategy is also used in a parser of Standard ML [7]. At the end of parsing process, our parser removes prefixes attached by a tokenizer to symbols, then outputs an XML file.

D. Choice of programming languages

We selected C++ for our parser implementation language because the parsing process takes most of the execution time and requires a high performance implementation. On the other hand, we also chose Python3 for other processes to realize smooth linkage with other programs and increase the productivity of the implementation. As for the parser, when we tried both C++ and Python3 as ANTLR output languages, we confirmed that C++ is about 10 times faster than Python3. In the C++ version, the parsing process occupies about 50 to 70 percent of the whole execution time. To bridge the gap between Python3 and C++, we used the C++ extension feature of Python3 so that the data exchange is performed on memory.

IV. EVALUATION

The source code of our program is published and managed on GitHub under the MIT license[1].

A. Functionality

Most of our program is written in Python3 and is composed of highly extensible modules. Furthermore, since our program faithfully outputs an XML file that follows the official grammar rules written in BNF, it is easy to reuse its source code for development of other support tools of the Mizar system. Currently, although the platform on which our program runs is limited to UNIX, we also plan to support Windows and Mac OS in the future.

B. Performance

Table II shows a number of words and file size of each Mizar file used for a performance test. *jordan:95* is input data for a test case of our search system so that this file consists of a single theorem and a vocabulary section is not included. The file size of *ring_1* is standard and that of *jgraph_4* is the largest in the MML, respectively.

TABLE II
SPECIFICATION OF MIZAR ARTICLES

	number of words	size
jordan:95	168	0.575 kB
ring_1	11558	37.6 kB
jgraph_4	185895	492 kB

Table III shows the specification of the test environment used in the performance test.

TABLE III
TEST ENVIRONMENT

Item	Specification
CPU	Intel®Core™i7-7500U @ 2.70 GHz
Memory	16 GB
OS	Ubuntu 16.04 LTS
Compilier	GCC version 7

Table IV shows the execution time of each step of our program. Each item in the list corresponds to the labelled process shown in Fig. 2. From this table, it is confirmed that most of the execution time is occupied by process 7), that is, occupied by an ANTLR generated parser. According to the measurement results, in the case of an article of about 10,000 words like *ring_1*, the time consumption is less than one second, which means it is enough to be used in practical applications. However, when it comes to an article with more than 100,000 words like *jgraph_4*, parsing time exceeds 10 seconds. Currently, we suppose the application of our program is limited to the analysis of small input data

TABLE IV
TIME CONSUMPTION OF EACH PROCESS

	jordan:95	ring_1	jordan_1
1) Read Mizar Text File	0.0029 s	0.0004 s	0.0102 s
2) Remove Comments	0.0000 s	0.0007 s	0.0039 s
3) Tokenize Environment Part	0.0000 s	0.0019 s	0.0026 s
4) Parse Environment Part	0.0000 s	0.0030 s	0.0017 s
5) Create Symbol Dictionary	0.0196 s	0.0061 s	0.0077 s
6) Tokenize Environement Part	0.0015 s	0.1054 s	2.2488 s
7) Parse Main Part & Build XML	0.0235 s	0.4444 s	15.1322 s
8) Modify XML Tree	0.0013 s	0.0731 s	1.3009 s
9) Write XML File	0.0022 s	0.0275 s	0.4751 s
Total	0.0511 s	0.6626 s	19.1831 s

like *jordan:95*. Hence, we conclude our program already has enough performance for the application.

Table V shows the comparison result of performance measurement between our program and the existing Mizar parser. Even though this performance comparison is unfair because the existing Mizar parser has additional features such as indexing variables, it is enough to check approximate relative performance of these two programs. This comparison made it clear that our program tends to be slower than the existing parser as the input file size becomes larger. This tendency mainly comes from the difference of parser algorithms between the conventional bottom-up parsing and top-down parsing. The official grammar rules of the Mizar language include a significant number of left recursions that cause backtracking in the process of top-down LL(*) parsing and performance deterioration. This performance deterioration is a well known issue that occurs when ANTLR generates parsers for languages with complex grammar rules.

TABLE V
PERFORMANCE COMPARISON BETWEEN CURRENT AND NEW VERSIONS

	current version	new version
ring_1	3.013 s	0.662 s
jgraph_4	3.610 s	19.183 s

V. REMAINING CHALLENGES

A. Display of parsing results

Since token interpretation of the Mizar language depends on the entries in a vocabulary section of an environment part, there is a possibility that the program produces incorrect results against user intention when it is applied to input data without a vocabulary section. Therefore, when parsing a snippet by our program, it is necessary to provide a graphical user interface (GUI) that allows users to check the parsed result visually. In the development of our search system, we are planning to build up a component that converts a parsing result into an HTML document with highlights of syntax errors, hyperlinks to symbol definitions, and so on.

[1]https://github.com/mimosa-project/emparser

B. Type checking

The Mizar language allows symbol overloading and mode inheritance as well as Java and C++ languages. Therefore, type checking or type inference must be performed in the semantic analysis. Improving the precision of the semantic analysis is expected to greatly contribute to improve on the accuracy of our search system. There is a preceding research on type inference without an environment part by Cezary Kaliszyk et al. [8].

C. Performance improvement

We are planning to improve the performance for the case where our program is applied to other than our search system in the future. According to the performance measurements, it is supposed that effective remedies are to change the parser algorithm to bottom-up parsing or to optimize a grammar file passed to ANTLR. However, the replacement to a bottom-up parsing makes error handling more difficult, and optimization of the ANTLR grammar file has a disadvantage of impairing readability of the syntax rules. Another improvement plan is to rewrite program components written in Python, such as tokenizer, by using C++ extensions.

ACKNOWLEDGMENT

I would like to express my gratitude to Adam Naumowicz, Artur Korniłowicz and Radosław Piliszek, who explained the specification of the existing Mizar tokenizer and parser programs and provided a part of their source code.

REFERENCES

[1] G. Bancerek, C. Byliński, A. Grabowski, A. Korniłowicz, R. Matuszewski, A. Naumowicz, and K. Pąk, "The role of the Mizar Mathematical Library for interactive proof development in Mizar," *Journal of Automated Reasoning*, vol. 61, no. 1, pp. 9–32, Jun 2018. [Online]. Available: https://doi.org/10.1007/s10817-017-9440-6

[2] G. Bancerek, "Information retrieval and rendering with MML query," in *Mathematical Knowledge Management*, J. M. Borwein and W. M. Farmer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 266–279. [Online]. Available: https://doi.org/10.1007/11812289_21

[3] F. Guidi and C. Sacerdoti Coen, "A survey on retrieval of mathematical knowledge," *Mathematics in Computer Science*, vol. 10, no. 4, pp. 409–427, Dec 2016. [Online]. Available: https://doi.org/10.1007/s11786-016-0274-0

[4] C. Bylinski and J. Alama, "New developments in parsing Mizar," in *Intelligent Computer Mathematics*, J. Jeuring, J. A. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 427–431. [Online]. Available: https://doi.org/10.1007/978-3-642-31374-5_30

[5] A. Naumowicz and R. Piliszek, "Accessing the Mizar library with a weakly strict Mizar parser," in *Intelligent Computer Mathematics*, M. Kohlhase, M. Johansson, B. Miller, L. de Moura, and F. Tompa, Eds. Cham: Springer International Publishing, 2016, pp. 77–82. [Online]. Available: https://doi.org/10.1007/978-3-319-42547-4_6

[6] P. Cairns and J. Gow, "Integrating searching and authoring in Mizar," *Journal of Automated Reasoning*, vol. 39, no. 2, pp. 141–160, Aug 2007. [Online]. Available: https://doi.org/10.1007/s10817-007-9073-2

[7] A. W. Appel and D. B. MacQueen, "Standard ML of New Jersey," in *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 1991, pp. 1–13. [Online]. Available: https://doi.org/10.1007/3-540-54444-5_83

[8] C. Kaliszyk, J. Urban, and J. Vyskočil, "Learning to parse on aligned corpora (rough diamond)," in *International Conference on Interactive Theorem Proving*. Springer, 2015, pp. 227–233. [Online]. Available: https://doi.org/10.1007/978-3-319-22102-1_15