Indian Institute of Science,
Bangalore

PROJECT SYNOPSIS

Extracting formal mathematics from text: Mizar as a model

Ankit Kumar Jaiswal
S.R. 09937, UG Department.

supervised by

Prof. Siddhartha Gadgil
Department of Mathematics
IISc Bangalore.

May 24, 2018

Acknowledgement

I would like to express my gratitude to my supervisor Prof. Siddhartha Gadgil for the useful guidance, comments, remarks and engagement throughout the learning process of this project. Furthermore, I would like to thank Mr. Mathias for very supportive documentation of *Parboiled2*. Also, I like to thank the *Mizar* Community for such an expressive yet formal language, and for developing almost up-to-date library of important theorems and their proofs, and keeping its source open. I would like to thank my loved ones, who have supported me throughout the process, both by keeping me harmonious and helping me putting pieces together.

1 Introduction

Mizar is the name of a formal language designed by Andrzej Trybulec for writing strictly formalized mathematical definitions and proofs. It is designed to be as close as possible to the language used in mathematical papers and at the same time to be automatically verifiable.[1] The two goals are achieved by selecting a set of English words which occur most often in informal mathematics; and eventually formulating a grammar which is rigorous enough (but human readable) to be handled with the present automated tools. In particular, the language includes the standard set of first order logical connectives and quantifier for forming formulas.

A text written in this grammar is referred as an Article. Since 1989, Mizar community has been developing a database for mathematics which currently includes more than 9400 definitions of mathematical concepts and more than 49000 theorems.[5]

Content extraction of a text partially or completely from a huge library like of *Mizar*, is achieved through computer programs which slices down the source contents into bits of tokens or vocabularies over which source's grammar is formulated and; outputs them as a *tree*. Therefore this project involves two steps; construction of target language, i.e. a *abstract tree* to store the parsed data and; construction of a parser using a *parser generator*, which would parse an article and simultaneously map the parsed entities to a suitable data structure. Both steps have been performed in *Scala* programming language.

Parsing is the process where we take a particular sequence of words and check to see if they correspond to the language defined by some grammar. In addition to that, one stores the desirable content by creating an instance of data structure corresponding to it. And construction of such data structures depends on what is being extracted, as well as the grammar.[3]

2 Mizar Grammar

Like any other formal language, *Mizar* Grammar can be seen containing following two sections[3]:

- The *lexical* Syntax, which defines which patterns of characters make up which word.
- The *Context-free* Syntax, which defines which sequences of words make up phrases from the language.

In particular, every word in a *Mizar* article is either of the following:

- *reserved words* some commonly used (in mathematical vernacular) English words such as, 'suppose', 'let', 'according', 'def', 'thm', 'implies', 'not' etc.
- *symbols* introduced in article's vocabulary file, which is a string of arbitrary characters excluding control characters, space, and double colon; symbols are used for operators, functions etc.
- *numerals* sequences of digits starting with a non-zero digit.
- *identifiers* strings of letters or digits that are neither reserved words, symbols, nor numerals; identifiers are used to name variables.
- *filename* uppercase strings of up to eight characters (alphanumeric and underscores); used in *environment directives* to import items from selected articles, and in *library references* within the main part of an article.

And a specific ordered collection (which is well formalised in CFG syntax) of words forms phrases inside the article. For example, following phrase is a *reservation segment* for an article; which basically reserves some information to the variable x, y and z:

Listing 1: Example 1

```
reserve x, y, z for Real;
```

where 'reserve' is a *reserved word*, 'x, y, z' is a list of *identifiers*, 'for' is a *reserved word*, 'Real' is a *symbol* declared inside $REAL\_1.voc$ file and, this special character ';' marks the end of such statement, hence comes under *reserved words*.

Structurally, a *Mizar* article has two parts namely:

- *environment-declaration* which includes some import statements for importing vocabulary (*.voc* files) inside which a desired *symbol* is declared.
- *textproper* which is basically the body part of an article.

For example, in order to write Example 1 one has to import the *symbol* 'Real' by adding following line to the environment-declaration part of an article:

```
vocabularies REAL_1;
```

where 'vocabularies' is a *reserved word*, 'REAL_1' is a *filename* and, ';' similar to above marks the end of statement.

Mathematical objects constructed using these *lexers* are definitions, theorem statements & their proofs etc. List of such objects separated by a ';' constitute the body part. For example, following is a function definition which defines the complement of a subset under the set:

Listing 2: Example 2

```
definition
let E be set, A be Subset of E;
func A' -> Subset of E equals
E \ A;
end;
```

where 'definition' and 'end' are *reserved words* and, first line is the declaration of the function arguments with its type (called in Mizar as, *loci*). The second line can be broken into 'func' & 'equals' as *reserved word*, 'A' as function argument, ' ' ' as function *symbol*, '->' as *reserved word* pointing towards the co-domain of the function, i.e. 'Subset of E' and, '$E \setminus A$' as the implementation of the function.

3 Target Language

Similar to the case of phrase construction conforming to the rules of grammar, we can create a data structures for each mathematical object (from atomic formulas to theorems and their proofs) by creating data structures for *lexers* and associating a unique data structure to the same specific ordered collection of *lexer*, which forms the phrases in Mizar. This way we obtain a *abstract tree* for storing parsed data.

In object-oriented-programming, a *class* is a program-code-template for creating objects or data structures. In particular, we used *extensible classes* to define data structures, with some of them extending other classes, meaning if *class A* extends *class B* then instance of A can also be treated as instance of B. With Scala's concise way of defining *types*, *sub-types* and, respective *constructors* makes it easy to construct the *abstract tree*. Also with the use of Scala's *case class* instead of *class* one can do easy pattern matching. A class in Scala defines two things namely, a *type* denoted with the name of the class and; a *constructor* (i.e. a function) with the same name operating on arguments written inside the brackets.

For example, following defines a data structure to encompass *reservation statements* (which is a kind of *Text-Item* in Mizar) like in Example 1:

```
case class ReserveSegment(reserveIden: Identifiers , sym: Symbol)
case class Identifier(alphaNum: String)
case class Symbol(voc: String)
```

And tree structure is obtain using class extensions. For example, following defines a tree of data structures for, what could be a Text-Item:

```
trait Text-Item
case class Reservation(segments: List[ReserveSegment])
     extends Text-Item
case class DefinitionalItem(defList: List[DefinitionalBlock])
     extends TextItem
case class NotationItem(noteList: List[NotationBlock])
     extends TextItem
case class Theorem(thm: CompactStatement)
     extends TextItem
```

4 Parser using *Parboiled2*

Parboiled2 is a macro-based *parser generator* for Parsing Expression Grammar (PEGs) which implements top-down-parser. PEG here has been used to write the parsing rule. A PEG parser consists of a number of rules that logically forms a "tree", with one "root" rule at the top calling zero or more lower-level rules, which can each call other rules and so on. When a rule is executed against the current position in an input buffer, it applies its specific matching logic to the input, which can either succeed or fail. In the success case the parser advances the input position (the cursor) and potentially executes the next rule. Otherwise, when the rule fails, the cursor is reset and the parser backtracks in search of another parsing alternative that might succeed.[4]

Formally, a parsing grammar consists of a set of each of the following:

- *nonterminals* here, the lexers of Mizar.
- *terminals* here, the name of the phrase, like *reservation segment* in Example 1.
- *parsing rule* constructed for both *terminals* and *nonterminal* using Mizar's context-free syntax.

A parsing rule is a hierarchical expression. Parsing rules for terminals are basically their pattern of characters inside the *rule*{} macro. Given any existing

parsing rule e, e1, and e2, a new one is constructed using the following operators; Sequence: $e1 \sim e2$, Ordered choice: $e1 \mid e2$, Zero-or-more(e), One-or-more(e), One-or-more(e), Optional(e), And-predicate: &(e), Not-predicate: !(e).

For example, following defines the parsing rule for parsing *reservation* given in Example 1:

```
def reservation = rule{"reserve" ~
    oneOrMore(reserveSegment) ~ ";"}
def reserveSegment = rule{oneOrMore(identifier) ~
    "for" ~ symbol}
def symbol = rule{"Real" | ...other vocabularies}
def identifier = rule{oneOrMore(AlphaNum)}
```

Extraction is achieved through following two parser actions:

- *capture*(a) captures the matched input by rule a in the form of strings, for further storage.
- $a \sim>$ () applies the function given as argument to the captured string, to construct objects of classes defined above.

Therefore, the previous example is modified to output the desired object. Similarly, parsing rules for all other *nonterminals* have been constructed.

5 Issues

Left Recursion: A grammar is left recursive if we can find some nonterminal A which will eventually derive a *sentential form* with itself as the left-most symbol.

Left recursion often poses problems for top-down-parsers because it leads them into infinite recursion. Therefore, the parsing rule grammar had to be modified to remove all cases of left recursion. Following shows an example of a left recursive parsing rule and its replacement below it; where A, A' are *nonterminals*, α_i are nonempty sequence of *terminals* and *nonterminals*, β_i are sequence of *terminals* and *nonterminals* that does not start with A, and ϵ is the null string (*terminal*).[3]

$$A \rightarrow A\alpha_1 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m \tag{1}$$

were replaced with,

$$A \rightarrow \beta_1 A' \mid ... \mid \beta_m A' \tag{2}$$

and,

$$A' \rightarrow \alpha_1 A' | \ldots | \alpha_n A' | \epsilon \tag{3}$$

References

[1] Grabowski, A., Kornilowicz, A., Naumowicz, A., *Mizar in a Nutshell*, Journal of Formalized Reasoning 3(2), pp. 153-245, 2010.

[2] , Wiedijk, F., *Writing a Mizar article in nine easy steps.*

[3] Power J., *Notes on Formal Language Theory and Parsing*, Maynooth, Co. Kildare, Ireland., 2002

[4] Mathias, *parboiled2/README.rst*, `https://github.com/sirthias/parboiled2/blob/release-2.1/README.rst`

[5] Mizar Community, *Mizar Mathematical Library*, `http://mizar.org/version/current/mml/`