MathSemantifier - a Notation-based Semantification Study

Toloaca Ion
Prof. Dr. Michael Kohlhase
Jacobs University Bremen

May 27, 2016

Abstract

Mathematical formulae are a highly ambiguous natural language fragments for which typesetting systems as LaTeX store only the rendering information. MathSemantifier is an open-source notation-based mathematical formula semantification system that attempts to tackle the problem of ambiguity in mathematical documents and produce knowledge-rich equivalents. The system extracts formulae (from formats such as LaTeX or MathML) and produces content-rich results (Content MathML) that contain no ambiguity. The disambiguation is achieved by matching the input formulae against a known database of approximately 700 notation definitions, which is aggregated into a Context Free Grammar of about 2800 rules. This paper outlines an implementation of MathSemantifier that focuses on helping researchers in semantifying their works. The ultimate goal of MathSemantifier is a scalable implementation that would need minimal help from a human. Such a system could be used to semantify large collections of mathematical papers such as arXiv [arX15].

1 Introduction

The scientific community produces a large number of mathematical papers (approximately 108.000 new papers per year [arX]), which raises the importance of machine based processing of such documents. Unfortunately, the most popular formats in which these papers are found (for instance, LaTeX) do not contain much information that would allow the computers to infer the human-understandable knowledge contained within a paper. Since, at this point, changing these formats is not practically possible, the other solution is to add a semantic flavor to the existing documents by translating them into a more suitable format, for instance, Content MathML.

1.1 History and Motivation

As a system as complex as the current scientific community was created, it went through a series of evolutions in the attempt to introduce the best method of writing scientific documents. This process was highly influenced by the invention and spreading of the internet. Scientists understood the necessity of a standard that could help them write and exchange their findings in an efficient way. A lot of effort has gone into translating books into digital documents.

The next step in this evolution is translating digital documents into knowledge-rich digital documents. This next step can only happen if a new feasible way of transition appears, which **MathSemantifier** attempts to become.

1.2 Ambiguity in Mathematical Documents

An important concept necessary in order to understand why semantification is a complex process is ambiguity. Mathematical documents are not a simple collection of symbols. The main use of these documents emerges only when the intended semantics of a document is accessible. However, humans tend to be lazy in writing down the whole graph, but instead rely on implicit human knowledge to decipher these documents. This is where ambiguity comes into play, when the author relies on the ability of the human to use the context of document in order to pinpoint the actual meaning an expression. Ambiguities can be largely divided into two: structural and idiomatic ambiguities.

1.2.1 Structural Ambiguities

A simple example that demonstrates the concept of structural ambiguities can be sin x / 2. It can mean one of the following:

1. sin applied to $\frac{x}{2}$
2. $\frac{1}{2}$ times sin applied to x

1.2.2 Idiomatic Ambiguities

Contrary to structural ambiguities, idiomatic ambiguities are not due to different parse trees. Given one single parse tree, some formulae allow for multiple readings. A standard example would be B_n. This could be:

1. The sequence of Bernoulli numbers
2. A user defined sequence
3. The vertex of one of a series of geometric objects

1.3 Extracting Semantics from Mathematical Documents

In this section we propose a solution to the ambiguity problem described above. Consider $c(a(b))$. The most natural interpretation is c of a of b. However, there arise multiple meanings if we consider that the application could be interpreted as multiplication. The human reader discards such meanings by convention and experience of handling mathematical documents, however, teaching this to a computer is a complex task. The approach that **MathSemantifier** takes is simply extracting all the possible meanings, while trying to apply heuristics to weed out impossible meanings.

1.4 An Introduction to MathSemantifier

In order to understand what **MathSemantifier** does, the Presentation Algorithm needs to be explained first. The reason for that is that **MathSemantifier** is the exact opposite of the Presentation Algorithm, trying to convert PMML to CMML, as opposed to CMML to PMML.

1.4.1 The Presentation Algorithm

The presentation algorithm has its main goal to covert Content MathML to Presentation MathML, using a database of notation renderings. In Figure 1 a typical example of what the presentation algorithm produces is displayed. The first child of the **semantics** node contains the PMML that corresponds to the CMML contained in the **annotation-xml** node. In order to produce this output, the algorithm used the notation **natarith addition** (shown in Figure 1 as well). OMDoc Notations like **natarith addition** are initially written in sTEX, and then converted to OMDoc using LATEXML. Further sections will reveal more detail both about the origin of the notation (see subsection 2.1), as well as about the LATEXML conversion (see subsection 2.2).

1.4.2 Approach to Semantification

MathSemantifier converts PMML into valid CMML as described above. In order to perform this task, it needs to match PMML against a list of notations. This is achieved by

Figure 1: Presentation Algorithm Output and the corresponding Notation Definition

compiling the notations into a Context Free Grammar, and using a CFG Parsing Engine to parse the PMML. A parse returns a list of possible parse trees, out of which **MathSemantifier** extracts information regarding what notations matched at top level. This is then done recursively for the arguments of the found notation. The **Implementation** section describes in a lot more detail how the Notations are compiled into a CFG, and how the parse trees are converted to CMML. Parsing using CFGs is a known problem, so, instead of writing a Parsing Engine, a more reasonable approach is using an existing one. For that purpose, the **Marpa Grammar Engine** is used.

1.4.3 Possible Applications of Semantification

- **MathWebSearch [HPK14]** (discussed in more detail in **??**) is a search engine for mathematical formulas. Such search engines could greatly benefit from semantification. The idea is that a search engine is only as good as the database of information is. By improving the information it can search through by adding a semantic flavor to it, a new kind of queries could be possible - semantic queries. Rather than searching for strings, or formulas with free form subterms, the user could specify the meaning of the sought expression. This would improve a lot the relevance of the results, since there will be no result that matched just because it was presented in a similar manner.

- Another possibility for using semantification is theorem proving and correctness checking. One possible application would be realtime feedback to the user writing a paper about the correctness of the expressions used.

- The possibilities extend even beyond this. Using semantified content, rather than having a database of CMML expressions, it is possible to create a smart knowledge management system that could be used to create expert systems. The user could then ask questions, or create complex queries, to exploit the full power of semantic content.

Semantification will not make all of the above directly possible, however it is a necessary step towards achieving goals similar to the ones described above, that require more knowledge about the used content than just how it is rendered.

2 Preliminaries

2.1 The Notation Database of MMT

MathSemantifier uses a Notation Database to create the grammar it uses. Therefore, it needs to be explained where those notations come from. MMT [Rab] is a language developed as a scalable representation and interchange language for mathematical knowledge. The decisive factor about MMT is that there is already a large database of notations written in sTEX, which is transformed and stored in MMT in an original format that MathSemantifier is processing in order to generate a Context Free Grammar.

SMGloM [GIJ+] is a part of the notation database of MMT that contains the notations used by **MathSemantifier**. SMGloM is also available online [Kohb].

2.2 LaTeXML Pre-processing

Since sTEX, as an extension of LaTeX, is just as complex to parse, MMT uses LaTeXML in order to convert sTEX to OMDoc [Koha]. OMDoc is then processed and stored in MMT.

2.3 Marpa Context Free Grammar Parser

MathSemantifier takes the an auto-generated CFG and input from the **Web UI** in order to produce possible parse trees of the input. Marpa Grammar Engine is the proposed. The main advantages are that Marpa handles ambiguous grammars and provides control over the parsing process. The author of Marpa provides a more detailed analysis of the advantages of Marpa (see [Keg]).

3 The MathSemantifier System

The major idea of **MathSemantifier** is, as already described in the introduction, finding possible Content MathML readings for Presentation MathML input expressions.

The general flow of a single semantification can be described as follows:

1. Context Free Grammar generation from the MMT Notations
2. Parsing using the Marpa Grammar Engine and the generated CFG to detect the top level notation
3. Parsing the arguments of the top level notation recursively

4. Using the parse trees from step 2 and 3 to generate an internal representation of the meaning trees

5. Converting the meaning trees to Content MathML

6. Displaying the Content MathML trees in the frontend

The reason it was decided to use a CFG based solution is that there exist already parsing frameworks like the Marpa Grammar Engine. It solves all the parsing related technical problems, like parsing ambiguous expressions, different kinds of recursion, while also providing a high degree of freedom.

This sections goes into the details of how exactly semantification is accomplished. First, a general overview of the goals of the project and its high level architecture is given. Then, each component is described in further detail.

3.1 Project Goals

The main goals of the project can be expressed in a concise manner as follows:

1. Generating the correct set of parses efficiently and effectively

2. Providing opportunities for improvement for further research in the area

This following subsections how **MathSemantifier** achieves the above goals.

3.2 **MathSemantifier** Architecture

The architecture can be roughly divided into four parts as shown in Figure 2. The components will be discussed in more details in the further subsections.

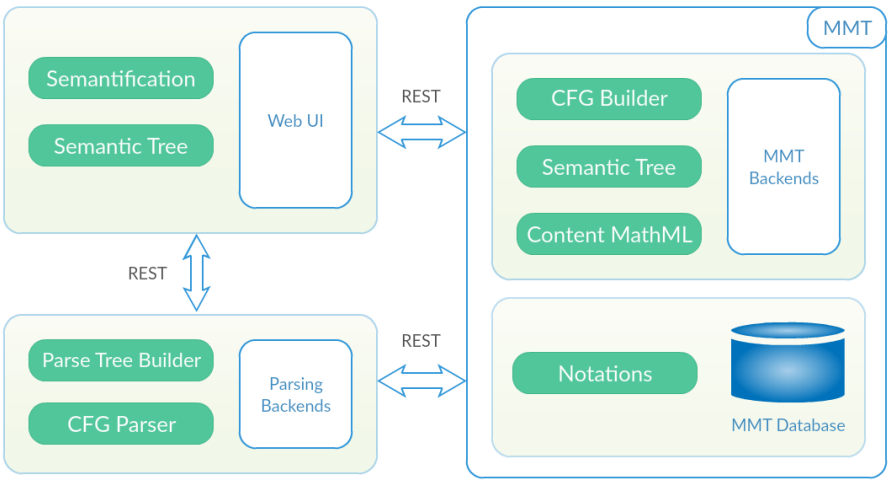

Figure 2: **MathSemantifier** Architecture

3.3 Web User Interface

The **Web UI** is a core component of **MathSemantifier.** It is intended to be a lightweight solution that queries a server for the results of more computationally intensive tasks.

The interface consists of an input area, where **MathML** needs to be inputted, and three options:

1. Semantify (The user can guide the semantification of the top symbol directly, by choosing the correct matching range, notation and argument positions)
2. Show Semantic Tree (The other option is to ask for all the possibilities and get all the semantic trees)
3. Evaluation (In this case, by repeatedly pressing this, the user is walked through a series of examples to demo the functionality)

The **Evaluation** option will be discussed in more detail in section 5.

The repository containing the **Web UI** can be found on GitHub [Tolb].

3.3.1 User Guided Semantification

The user provides Presentation MathML as input to the system, then uses the **Semantify** button to reveal a list of top level notation names. The names of the notations are derived from the notation paths as follows: **archive name** + **symbol name**, for example, **natarith addition** refers to the addition of natural numbers. After determining which notation is the correct one, the user needs to make sure that the arguments were detected properly and, finally, the resulting Content MathML tree is displayed (see Figure 3 for a similar representation of the result).

3.3.2 Semantic Tree Generation

The easier but computationally more expensive alternative is to simply generate all the possible parse trees at once and display them. The user simply needs to click the **Show Semantic Tree** option.

Figure 3: Semantic Tree Results

For the example shown in Figure 3, there is a total of 342 different readings. This can be easily explained, since **Invisible Times**, **Arithmetic Plus** and **Mod** have each multiple notations definitions (that originate from different MMT archives, for instance), and there are no limitations on what kind of notation definitions can go together in the same CMML tree.

3.3.3 Term Sharing

In order to minimize the Content MathML, the standard allows subtree sharing. To enable this option, the **Use term sharing** checkbox should be checked. In that case, the terms is shared along different readings.

3.4 MMT Backend

The MMT Backend is a **Server Extension** that is part of MMT. As shown in Figure 2, it is linked via **REST** with the **Web UI** and the **Parsing Backends.**

Its role can be summarized to the following core functions:

1. Compile the **MMT Notations** into a CFG Grammar
2. Receive the input from the **Web UI** and build its Semantic Tree
3. Delegate the parsing to the **Parsing Backends**

I decided to put the core logic of the application in MMT in order to make it easier to interoperate with the MMT Notation Database, as well as with any other MMT components that may want to need **MathSemantifier.** By the current design, using **MathSemantifier** within MMT is as simple as a function call that takes the input as a string.

The code can be found as part of the MMT codebase [KT].

Let us look at the components of the MMT Backends in more detail below.

```
#GRAMMAR ENTRY POINT
:default ::= action => [name, start, length, values]
lexeme default = latm => 1
:start ::= Expression
ExpressionList ::= Expression+
Expression ::= Presentation
             | Notation

#Notation Precedences
Notation ::= prec0
prec0 ::= prec1 | #Rules with precedence zero
prec1 ::= prec2 | #Rules with precedence one
...

#Argument Precedences (depends on the number of precedences)
argRuleP0 ::= prec0 | Presentation
argRuleP1 ::= prec1 | Presentation
argRuleP2 ::= prec2 | Presentation
...

#Presentation MathML
Presentation ::= mrowB Notation mrowE
 | mrowB ExpressionList mrowE
 | moB '(' moE ExpressionList moB ')' moE
 | moB text moE
 | miB text miE
 | mnB text mnE
 ...

#Presentation MathML Parts
mrowB ::=  '<mrow' attribs '>'
mrowE ::=  '</mrow>'
mathB ::=  '<math' attribs '>'
mathE ::=  '</math>'
miB ::=  '<mi' attribs '>'
miE ::=  '</mi>'
...

#Lexemes
ws ::= spaces
spaces ~ space+
space ~ [\s]
text ::= textRule
textRule::= char | char textRule
char ~ [^<>]
...
```

Figure 4: CFG Preamble

3.4.1 CFG Generator

The Grammar Generator aggregates all the knowledge contained in the MMT Notations into one Context Free Grammar. The grammar is shaped into the normal form accepted by the Marpa Grammar Engine. To achieve this, the format used to store the notations in MMT is decomposed into CFG rules. Otherwise said, the tree-like structure of each formula, that is stored as nested applications of **MMT Markers** (discussed previously) needs to be serialized into CFG rules.

This is done in several steps:

1. Break apart the **MMT Marker** trees into level by level representations
2. Transform the intermediate representation into valid CFG rules

3. Optimize if possible (will be discussed in detail in the subsequent sections)

The fundamental structure of the CFG is established using a preamble as shown in Figure 4. Note the default action is the **Grammar Entry Point.** It is precisely what tells the Grammar Engine to build parse trees, and also determines their structure.

The entry point into the grammar is the **Expression** rule. It can be an MMT Notation, or Presentation MathML. The **prec0** rule should be read as precedence zero, that is - the lowest precedence there is.

Precedence handling is done using a commonly used method for including it in CFGs. The **Notation Precedences** section in Figure 4 gives a quick glance at how exactly it is done. The number of precedences needs to be known in advance, then, for each precedence value a corresponding **precN** rule is created.

```
1  _natarith_additionP7N213::= rule443
2  rule443::= argRuleN213A1ArgSeq rule32 rule443_
3  rule443_::= argRuleN213A1ArgSeq | argRuleN213A1ArgSeq rule32 rule443_
4  rule32::= moB '+' moE
```

Figure 5: MMT Notation in CFG

The rule contains all the notations with that precedence, and, one of the alternatives is going to a higher precedence value. In the example below there are 15 used precedence values (**prec0** corresponds to precedence $-\infty$). However, the **Notation Precedences** section shows only half of the concept.

To make this approach work, what is also needed is that the arguments in a rule of a certain precedence N can only contain notations with precedence K if $K > N$. This is done by the **Argument Precedences** part in the preamble (as shown in Figure 4).

Finally, Figure 5 presents an example of what the **natharith addition** MMT notation from the **smglom/numberfields** archive translated to CFG rules looks like.

3.4.2 Semantic Tree Generator

The **Semantic Tree Generator** works by recursively querying the **Parsing Backend** and using the result to construct a the tree of possible meanings.

The parse trees stored in MMT have the following structure:

- **Variants -** represents a list of possible readings. It is always be the top node in any parse tree.
- **Notation -** a notation detected in the input. It contains its name and arguments.
- **Argument -** an argument of a notation. The plugin is recursively called on it to construct its meaning subtree as well.
- **RawString -** the ground term representation

3.4.3 Content MathML Generator

The final part of any of the semantification processes is converting the internal representation to a standard one, which is CMML in this case. MMT provides a simple API which requires:

1. The MMT notation path
2. The argument maps (maps from the argument number to the corresponding substring)

Both of which are available in my representation structure. The argument path is obtained by extracting the argument number from the argument name, and looking up in a map of paths created at the time of grammar creation. This implies the grammar rule names are overloaded with meaning, however, the possibilities are very limited in this aspect since the parsing framework used does not give complete control over the parsing process.

3.5 Parsing Backend

The Parsing Backend Figure 2 receives requests from the **Web UI** directly for Guided Semantification and from the **MMT Backend** for Semantic Tree Generation. All the parsing related work is delegated to this backend.

The code of the Parsing Backend can be found on GitHub [Tola].

The parsing backend consists of two parts.

3.5.1 Context Free Grammar Parser

First of all, the the CFG needs to be queried and parsed. This is implemented using lazy evaluation, which means that it is only done when a request actually comes.

The serialized CFG is unpacked and feed to the Marpa Parser Generator.

3.5.2 Parse Tree Generator

The more complex of the two parts is actually going through the parse trees and extracting useful information. Note that going through all the parse trees is not practical, so only the first N (currently 1000) parse trees are processed. This still gives the correct results in most cases since the grammar rules are optimized for giving preference to parse trees that are more likely to be correct (discussed in more detail in section 6).

4 Optimizations and Heuristics

This section describes the attempts made in order to deal with the overwhelming ambiguity of mathematical notations and produce actual results.

1. Only notations on the whole input are matched. Subterms are matched directly by the **Semantic Tree** plugin.

2. The number of parse trees is limited to $N = 1000$. The number of parse trees was empirically determined to include the set of correct parses for the examples used (see the **Evaluation** section**).**

3. The non-empty alternatives in the CFG are put first. The Grammar Parser has a predictable behavior of going through the alternatives left to right, and since empty rules match **always**, it is imperative that other alternative are attempted first. Note that only after this optimization any kind of results were possible to achieve on non-trivial input.

4. The CFG includes precedences correctly. This weeds out a large number of incorrect readings.

5. Sub-term sharing is part of the **MathML** standard, and within one request, if sub-term sharing is enabled, the response tries to share its terms as much as possible in order to compress the output.

6. **The Semantic Tree** plugin makes use of memoization to reduce one dimension of the exponential blowup.

5 Evaluation

This section presents an evaluation of **MathSemantifier** from the point of view of efficiency and effectiveness of semantic tree generation. Next, the interoperability of the system with other possible applications is examined, which mostly depends on the back-end APIs.

5.1 Semantic Tree Generation

Testing **MathSemantifier** through normal operation is not a trivial task, because the results need a human expert to check whether the results are indeed correct. Fortunately, the **MathHub Glossary [KWA]** contains a sufficient number (about 3000) of examples of Presentation MathML with Content MathML annotations that result from applying the **Presentation Algorithm** discussed in the introduction. Since **MathSemantifier** is the partial inverse of the **Presentation Algorithm,** checking whether the results are indeed correct boils down to comparing two Content MathML trees.

For the purpose of testing **MathSemantifier** on the **MathHub Glossary,** the **Evaluation** option mentioned in the **Web UI** section is used. It processes examples from the **MathHub Glossary.**

5.1.1 Results

A visual control of approximately 500 examples from the **MathHub Glossary** revealed the following results**.** About 40% of the examples are semantified correctly. However, this is largely because of reasons that do not depend on **MathSemantifier** itself.

Contrary to what the numerical result suggests, for expressions with less than 1000 parse trees that are correctly rendered, **MathSemantifier** produces correct results with a very high probability.

Let us look into the reasons that make **MathSemantifier** fail. First of all, the reasons that do not depend on the system itself.

1. The Presentation Algorithm fails to render CMML into PMML properly in about 30-40% of the cases.

2. About 10% of the examples include symbols from archives that were not included in the CFG

Up to this point, if we exclude the above mentioned examples, the effective success rate of **MathSemantifier** is about 80-90%.

1. Some notations are omitted because they create cycles in the CFG

2. Input data that is too long or complex having more than 1000 parse trees

3. Assuming the input is more knowledge-rich than it actually is

While the effective success rate of 80-90% is inspiring for a proof of concept system, the main issue of **MathSemantifier** is simply poor performance on long or complex input. Therefore, **MathSemantifier** is a successful proof of concept, but not yet a practical tool.

6 Conclusions

The conclusion of this study is the proof of concept architecture and implementation of a system capable of converting **Presentation MathML** to all the possible meanings, which is a list of **Content MathML**. The testing revealed that the system is able to recognize correctly single top level symbols, as well as the whole set of readings of expressions with less than 1000 parse trees (this is not the limit, but no testing in section 8 is done beyond that). Naturally, **MathSemantifier** can only recognize the symbols that were used when building the Context Free Grammar it uses. This shows that it is certainly possible to aggregate the knowledge from the MMT Notations and to use it for parsing purposes. However, both the Notations and the Parsing Framework need significant improvements in order for the system to be scalable beyond what is presented in the previous section. The most important part of this study is, therefore, the optimizations and heuristics used, and other techniques presented below that further research could benefit from.

6.1 Further work

The study presented in this paper reached certain results, however there is a long way yet to a fully automatic and scalable system that could handle large collections of papers without any supervision. Below there are some suggestions that further research could make use of in order to get closer to this goal.

6.1.1 Suggestions for further optimizations

1. The CFG could be checked for unused rules, and, more importantly, converted to BNF, for instance

2. Theory based optimization - either let the user specify which theories to create the grammar from, or take into consideration that notations from the same theory are likely to be close in the input (for instance, $-$ and $+$ are more likely to both be used as arithmetic symbols in an expression, than only one of them)

3. Sequence arguments (for instance, $2 + 3 + 4 + 5 + 6$) currently generate a very high number of possible parses. If more control over parsing were possible, parsing of sequence arguments should be greedy - attempting to take in as many delimiter argument pairs as there are.

6.1.2 Requirements for a more suitable Notation Database

1. The most important optimization would be adding types to the notation input arguments and output. This would allow for type based parsing, which would certainly be more efficient and generate more relevant results.

2. Grouping similar notations within a theory. For example, arithmetic plus on natural, integer or real number should be possible to connect somehow. If the notations have types, the it will naturally occur by grouping the notations with the same types into single rules.

3. For the **csymbol** used in an **apply** tag, the notation should cross-reference it with the part of the representation it corresponds to, since it is not otherwise clearly specified.

6.1.3 Requirements for a more suitable Parsing Framework

1. One of the biggest problems with the current implementation is that it greedily matches the argument renderings, which are free form subterms. If a custom parsing framework would be used, lazy matching of such subterms would greatly increase the performance.

2. The ability to handle types, precedences and associativity.

3. The ability to handle greedy parsing - most CFG parsing frameworks have the counted rule *Expression* ::= *Expression*+, which has a similar meaning to the + found in a **regex.** However, **regexes** actually will do a greedy match, the more appropriate match being then *Expression*+?, the non-greedy version. This is critical to reduce the number of useless parse trees when dealing with Sequence Arguments

References

[arX] arXiv : Monthly Submissions Statistics. https://arxiv.org/stats/monthly_submissions. [Online; accessed 8-May-2016].

[arX15] arXiv Online. *URL, as seen November 2015, http://arxiv.org*, 2015.

[GIJ+] Deyan Ginev, Mihnea Iancu, Constantin Jucovschi, Andrea Kohlhase, Michael Kohlhase, Heinz Kroger, Jurgen Schefter et Wolfram Sperber : The SMGloM Project and System.

[HPK14] Radu Hambasan, Corneliu C. Prodescu et Michael Kohlhase : MathWebSearch at NTCIR-11. 2014.

[Keg] Jeffrey Kegler : The Marpa Parser. https://jeffreykegler.github.io/Marpa-web-site/. [Online; accessed 19-April-2016].

[Koha] Michael Kohlhase : OMDoc - An open markup format for mathematical documents (Version 1.2).

[Kohb] Michael Kohlhase : The SMGLOM Archive. https://mathhub.info/smglom. [Online; accessed 4-May-2016].

[KT] KWARC et Ion Toloaca : MathSemantifier Parsing Backend. https://svn.kwarc.info/repos/MMT/. [Online; accessed 4-May-2016].

[KWA] KWARC : MathHub Glossary. https://mathhub.info/mh/glossary. [Online; accessed 1-May-2016].

[Rab] Florian Rabe : The MMT API: A Generic MKM System.

[Tola] Ion Toloaca : MathSemantifier Parsing Backend. https://github.com/itoloaca/Notation-Based-Parsing. [Online; accessed 4-May-2016].

[Tolb] Ion Toloaca : MathSemantifier Web UI. https://github.com/itoloaca/MathSemantifier. [Online; accessed 4-May-2016].