Semantic Preserving Bijective Mappings for Representations of Special Functions between Computer Algebra Systems and Word Processors

André Greiner-Petter

Information Science Group, University of Konstanz, Germany
andre.greiner-petter@t-online.de

Abstract

This paper presents a translation tool between representations of semantically enriched mathematical formulae in a Word Processor (WP) and its corresponding representations in a Computer Algebra System (CAS). The representatives for the CAS are Maple and Mathematica, while the representative WP is LaTeX. The National Institute of Standards and Technology (NIST) in the U.S. has developed a set of semantic LaTeX macros for Orthogonal Polynomials and Special Functions (OPSF). The Digital Library of Mathematical Functions (DLMF) use these semantic macros to provide a semantically enhanced mathematical online compendium. This kind of semantic macros provides exclusive access to the semantic information of the functions. However, even if the semantics of a representation of one formula is unique, it does not need to match the semantics in another representation for the same formula. While some distinctions are obvious, such as syntactical characteristics, others are more difficult to examine, such as differences in definitions, which leads to error-prone manual translations. This paper presents an automatic translation system and discusses the problems and suggests solutions. Furthermore, this paper introduces new evaluation approaches for the developed translation tool. With the help of the automatic translation tool, the evaluation experiments were able to discover errors in the DLMF and in the CAS Maple.

Keywords: LaTeX, Computer Algebra Systems (CAS), Translation, Presentation to Computation (P2C)

1 Introduction

A typical workflow of a scientist who writes a scientific publication is to use a Word Processor (WP) to write the paper and one or more Computer Algebra System (CAS) for verification, analysis and visualization. Especially in Science, Technology, Engineering and Mathematics

(STEM) literature, LaTeX[1] has become the de facto standard for writing scientific publications over the past 30 years (Alex, 2007; Knuth, 1998, p. 559; Knuth, 1997). LaTeX enables printing of mathematical formulae in a structure similar to handwritten style. For example, consider the Jacobi polynomial (Olver et al., 2017, 18.3 in table 1)

$$P_n^{(\alpha,\beta)}(\cos(a\Theta)), \tag{1}$$

where $\alpha, \beta > -1$, and n a nonnegative integer. This formula is written in LaTeX as

```
P_n^{(\alpha,\beta)}(\cos(a\Theta)).
```

While LaTeX focuses on displaying mathematics, a CAS concentrates on computations and user friendly syntax. Especially important for a CAS is to embed unambiguous semantic information within the input. Each system uses different representations and syntax in consequence. Hence, a writer needs to constantly translate mathematical expressions from one representation to another and back again. Table 1 shows four different representations for the same formula (1).

Systems	Representations
Generic LaTeX	`P_n^{(\alpha,\beta)}(\cos(a\Theta))`
Semantic LaTeX	`\JacobiP{\alpha}{\beta}{n}@{\cos@{a\Theta}}`
Maple	`JacobiP(n,alpha,beta,cos(a*Theta))`
Mathematica	`JacobiP[n,\[Alpha],\[Beta],Cos[a \[CapitalTheta]]]`

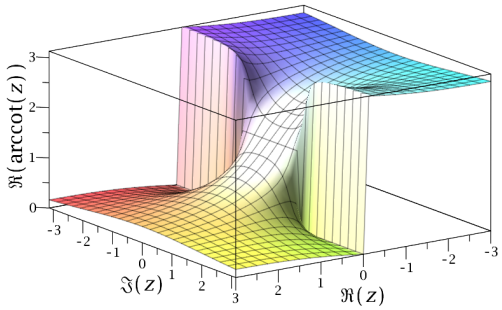
Table 1: Different representations for the same mathematical formula (1). Generic LaTeX is the default LaTeX expression. Semantic LaTeX uses special semantic macros to embed semantic information.

Translations from generic LaTeX to CAS are difficult to realize since the semantic information are not explicitely given in the input. The National Institute of Standards and Technology (NIST) has created a set of semantic LaTeX macros Miller and Youssef, 2003. Each macro ties specific character sequences to a well-defined mathematical object and is linked with the corresponding definition in the Digital Library of Mathematical Functions (DLMF) or Digital Repository of Mathematical Formulae (DRMF). These macros embeds necessary semantic information into LaTeX expressions. One example of such a macro is given table 1 for the semantic LaTeX representation for the Jacobi polynomial. The macros provide isolated access to important parts of the mathematical function, such as the arguments.

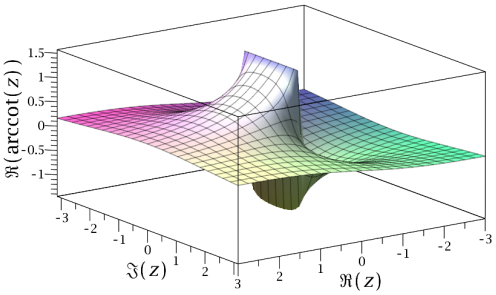
Even with embedded semantic information, a translation between the systems can be difficult. A typical example of complex problems occurs for multivalued functions (Davenport, 2010). A CAS usually defines *branch cuts* to compute principal values of multivalued functions (England et al., 2014), which makes the implementation of a theoretically continues function to a discontinued presentation of it. In general, positioning branch cuts follows conventions, but

[1]Note that technically LaTeX is not a WP (`https://www.latex-project.org/about/`, seen 07/2017) like *Microsoft Word*. However, since LaTeX (an extension of TeX) is the standard to write articles in STEM and we only focus on mathematical writings in this paper, we categorize LaTeX as a WP as well.

can be positioned arbitrarily in many cases. Communicate and explain the decision of defined branch cuts is a critical point for CAS and can vary between the systems (Corless et al., 2000). Figure 1 illustrates two examples of different branch cut positioning for the inverse trigonometric arccotangent function. While Maple defines the branch cut at $[-\infty i, -i]$, $[i, \infty i]$ (figure 1a), Mathematica defines the branch cut at $[-i, i]$ (figure 1b).



(a) The real part of arccotangent with a branch cut at $[-\infty i, -i]$, $[i, \infty i]$.



(b) The real part of arccotangent with a branch cut at $[-i, i]$.

Figure 1: Two plots for the real part of the arccotangent function with a branch cut at $[-\infty i, -i]$, $[i, \infty i]$ in figure (a) and at $[-i, i]$ in figure (b), respectively. (Plotted with Maple 2016)

Hence, a CAS user needs to fully understand the properties and special definitions (such as the position of branch cuts) in the CAS to avoid mistakes during a translation (England et al., 2014). In consequence, a manual translation process is not only laborious, but also prone to errors. Note that this general problem has been named to automatic Presentation-To-Computation (P2C) conversion (Youssef, 2017).

This article presents a new approach for automatic P2C and vice versa conversions. Translations from presentational to computational (computational to presentational) systems are called forward (backward) translations. A forward translation is denoted with an arrow with the target system language above the arrow. For example,

$$t \overset{\mathfrak{M}_{aple}}{\mapsto} c,$$

where t is an expression in the LaTeX language and c is an element of the Maple language $\mathfrak{M}_{aple}$. As we will see later in this article, we need to compare mathematical concepts between systems. This is impossible from a mathematical point of view. Consider the irrational mathematical constant e, known as Euler's number. The theoretical construct for this symbol cannot be mathematically equivalent to the value `exp(0)` in Maple, caused by computational and implementational limitations. Instead of using the term *equivalent*, we introduce a *appropriate* and *inappropriate* translations. We call a translation such as

$$\texttt{\textbackslash cos@\{z\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{cos(z)} \tag{2}$$

as *appropriate*, while a translation such as

$$\texttt{\textbackslash cos@\{z\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{sin(z)} \tag{3}$$

is called *inappropriate*. Note that it is not always as easy as in this example to decide if a translation is appropriate or not. Therefore, this article also presents several validation techniques to automatically verify if a translation is appropriate or inappropriate. In addition to this terminology, we introduce *direct translations*. Later in the paper, we will explain that a translation from one specific mathematical object to its *appropriate* counterpart in the other system is not always possible. We call a translation to the *appropriate* counterpart *direct*. For example, the translation (2) is *direct*, while a translation to the definition of the cosine function

$$\texttt{\textbackslash cos@\{z\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{(exp(I*z)+exp(-I*z))/2}$$

is not a *direct* translation.

Note that partial results of this paper have been published in (Cohl et al., 2017).

2 Related Work

Since LaTeX became the de facto standard for writing papers in mathematics, most of the CAS provide simple functions to import and export mathematical LaTeX expressions[2]. Those tools have two essential problems. First of all, they cannot import mathematical expressions where the semantic information is absent, which is the usual case for generic LaTeX expressions. Therefore, they are only able to import simple mathematical expressions, where the semantics are unique. For example, the internal LaTeX macro `\frac` always indicates a fraction. These import tools fail for more complex expressions, for example the Jacobi polynomial in table 1. The second problem appears in the export tools. Mathematical expressions in CAS are fully semantic, otherwise the CAS wouldn't be able to compute or evaluate the expressions. During the export process, the semantic information gets lost, because generic LaTeX is not able to carry semantic information. In consequence of these two problems, an exported expression cannot be imported to the same system again in most cases (except those simple expressions described above). Our tool should solve these problems and provide round trip translations between LaTeX and CAS.

Obviously, the semantics must be well known before an expression can be translated. There are two approaches to solve that problem: someone could specify the semantic information during the writing process (pre-define semantics) or the translator can find out the right semantic information in general mathematical expressions before it translates the expression. The second approach is unreasonably difficult compared to the pre-definition approach. We will talk about those difficulties in the following sections. We will also introduce a new approach in **??** to achieve the second idea.

[2]The selected CAS Maple, Mathematica, Matlab, and SageMath provide import and/or export functions for LaTeX:
Maple, `http://www.maplesoft.com/support/help/Maple/view.aspx?path=latex` seen 06/2017;
Mathematica, `https://reference.wolfram.com/language/tutorial/GeneratingAndImportingTeX.html` seen 06/2017;
Matlab, `https://www.mathworks.com/help/symbolic/latex.html` seen 06/2017;
SageMath, `http://doc.sagemath.org/html/en/tutorial/latex.html` seen 06/2017

However, the pre-definition approach is simple and easy to realize, as long as the system can carry semantic information. A typical, not mathematical example are Wikipedia articles (Foundation, 2001). An author of such an article can specify further information about symbols, words and sentences by creating a hyperlink to a more detailed explanation. Usually, the author defines the hyperlink, when he is writing the article or he needs to manually add the hyperlink later on. Similar to this approach are *interactive documents*[3], such as the Computable Document Format (CDF) (Research, 2011) by Wolfram Research or the *worksheets* of Maple, which are also sometimes referred to as interactive documents. Those documents pre-define the semantics to allow computations. Those complex document formats require specialized tools to show and work with the documents (Wolfram CDF Player, or Maple for the *worksheets*). The JOBAD architecture (Giceva, Lange, and Rabe, 2009) is able to create web based interactive documents and uses Open Mathematical Documents (OMDoc) (Kohlhase, 2006) to carry semantics. The documents can be viewed and edited in the browser. Those JOBAD-documents also allow to perform computations during CAS. This gives the opportunity to calculate, compute and change mathematical expressions directly in the document. The translation performs in the background, invisible for the user. Similar to the JOBAD architecture other interactive web documents exist, such as *MathDox* (Cuypers et al., 2008) and *The Planetary System* (Kohlhase et al., 2011). All of them demonstrate the potential for the education system.

All of these systems carry semantic information. Most of them create their own way to make this possible. The web based documents mostly use standardized ways, such as the content Mathematical Markup Language (MathML), a specialization of MathML (Ausbrooks et al., 2014), that focuses on the semantics of an expression rather than any particular rendering for the expression. Since LaTeX is the de facto standard to write mathematical expressions in documents there are projects to extend LaTeX in a way that enable to carry the semantic information directly in LaTeX. There are two known projects that created a semantic version of LaTeX: StEX (Kohlhase, 2008) developed by Michael Kohlhase and the DLMF/DRMF LaTeX macros developed by Bruce Miller. Our translator uses the DLMF/DRMF LaTeX macros rather than StEX. A full explanation of this decision will be given in **??**.

However, *interactive documents*, web based or not, have an essential problem: all of them create an entire new document format. We want to create an independent lite translation tool for LaTeX. Once we achieve this goal, an integration into *interactive documents* might be possible, besides many other possible applications.

Another approach that tries to avoid the translation problem are LaTeX packages that allow computations directly through the LaTeX compiler (for example *LaTeXCalc* (Churchill and Boyd, 2010)) or allow CAS commands for computations directly in TeX-files. For example, *sagetex* (Drake, 2009) is an interface LaTeX package for the open source CAS *sage*[4]. This package allows *sage* commands in TeX-files and uses *sage* in the background to compute the commands. Obviously, the first example is not as powerful as a CAS would be. However, *sagetex* does not really solve our problem of the scientific workflow. The writer still needs to translate expres-

[3]There is no adequate definition what interactive documents are. However, this name is widely used to describe electronic document formats that allow interactivity to change the content in real time.

[4]An abbreviation for *SageMath*.

sions manually. The only difference is that the author can write computations directly into the TEX-file. However, these approaches have the potential for a powerful combination with our translation tool.

In summary, today's translation tools are either specialized for one specific CAS (such as the import and export functions of the CAS) or they completely avoid the translation process by creating a new document format (such as *interactive documents*). Our goal is to provide a lite translation tool for mathematical LATEX expressions and multiple CAS.

3 Translation Problems

Most of the existing translation tools cannot preserve the semantic information during the translation process (see section 2). Therefore, our program should not lose semantic information during the process. However, the semantics can be changed during a translation. This happens when a definition or property of a mathematical object varies from system to system. Our reference systems for all mathematical LATEX expressions are the DLMF and the DRMF.

Let us focus on translations of mathematical objects that are not defined in one of the languages, first. Obviously, we cannot translate these objects *directly*. Our workaround is to translate not the mathematical object itself, but the definition of the object. For example, the *Gudermannian* (Olver et al., 2017, (4.23.10)) $\mathrm{gd}(x)$ function can be defined by

$$\mathrm{gd}(x) := \arctan(\sinh x) \quad x \in \mathbb{R}. \tag{4}$$

This function is not implemented in Maple. Therefore, we translate its definition (4)

$$\texttt{\textbackslash Gudermannian\{x\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{arctan(sinh(x))}. \tag{5}$$

We want to avoid such translations, because they are not intuitive and therefore can confuse a user. However, providing such translations is still better than providing no translation.

Instead of none possible translation, how can we handle multiple alternative translations? We already discussed the problem of the inverse trigonometric function $\mathrm{arccot}(x)$. Because of a different branch cut in Maple (compared to the DLMF definition), there are alternative translations available (see (**??**)). In such cases, we define the most intuitive, the direct translation. In addition, we inform the user about the available alternative translations and the branch cut issues. Hence,

$$\texttt{\textbackslash acot\{x\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{arccot(x)}. \tag{6}$$

Since we have these alternative translations, we also use the same technique for the previous Gudermannian example. We provide one direct translation and possible alternative translations for a mathematical object.

These two problems already explained, why our translation process is in general not bijective. However, we will explain later in a more detailed way that the translation is only bijective for a subset of possible expressions. Thereby, a bigger goal is to increase this subset as much as possible.

3.1 Libraries

The first step for providing an automatic translation tool is to defining the translation for each function. We manually create a table of translations for each DLMF/DRMF LaTeX macro. Translations for generic LaTeX functions, mathematical symbols, Greek letters and constants are organized in JavaScript Object Notation (JSON) files for a better performance.

For each translation, we design a translation pattern with placeholders to define the correct position of each argument. The placeholders are defined as `$i`, where `i` is a non-negative integer specifying the position, starting at 0. Table 2 illustrates the translation patterns for the trigonometric sine function in semantic LaTeX, Maple and Mathematica.

Semantic LaTeX	`\sin@{$0}`
Maple	`sin($0)`
Mathematica	`Sin[$0]`

Table 2: Translation patterns for the sine function in the DLMF, Maple and Mathematica.

Placeholders gives us the opportunity to define the order of the arguments and handle nested function calls. The order of arguments can differ between the systems. For example, if a function possesses parameters written in sub- and superscripts, Maple and Mathematica usually put the parameters in the subscript before the parameters in the superscripts. The notation in the DLMF/DRMF LaTeX macros uses the reversed order. Table 1 in the introduction shows an example of such different orders. The translation patterns for the Jacobi polynomial are illustrated in table 3.

Forward Translation:	
Maple	`JacobiP($2, $0, $1, $3)`
Mathematica	`JacobiP[$2, $0, $1, $3]`
Backward Translation from Maple/Mathematica:	
Semantic LaTeX	`\JacobiP{$1}{$2}{$0}@{$3}`

Table 3: Forward and backward translation patterns of the Jacobi polynomial. The pattern for the backward translation is the same for Maple and Mathematica.

Note that Maple also uses the symbol `$` in few functions. This can cause trouble in the replacement process. However, we can handle all of the few cases by adding additional parenthesis.

Consider the differentiation function in Maple

$$\texttt{diff(f, [x\$n])}, \tag{7}$$

with f as an algebraic expression or an equation, x the name of the differentiation variable and n for the n-th order differentiation. Therefore

$$\frac{d^2x^2}{dx^2} = \texttt{\textbackslash deriv[2]\{x\^{}2\}\{x\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{diff(x\^{}2, [x\$2])}. \tag{8}$$

But the translation pattern in Maple would be

$$\texttt{diff(\$1, [\$2\$\$0])}. \tag{9}$$

Replace the placeholders sequentially by the arguments ends in

$$\begin{aligned} & \texttt{diff(\$1, [\$2\$\$0])} \\ \text{Replace index 0 by 2} \quad &\rightarrow \quad \texttt{diff(\$1, [\$2\$2])} \\ \text{Replace index 1 by } x^2 \quad &\rightarrow \quad \texttt{diff(x\^{}2, [\$2\$2])} \\ \text{Replace index 2 by } x \quad &\rightarrow \quad \texttt{diff(x\^{}2, [xx])}. \end{aligned}$$

We use parenthesis to solve this issue and define the translation pattern with

$$\texttt{diff(\$1, [\$2\$(\$0)])}. \tag{10}$$

A special problem can be binary operators. A good example is the multiplication sign. In LaTeX most scientist would use white spaces or `\cdot` to indicate a multiplication (using none of them can cause ambiguities, see 4.2.2). Similar to that approach, Mathematica also understands white spaces and asterisks as multiplications. When we translate semantic LaTeX expressions to Maple, we use the 1D Maple representation (see subsection **??**), where an asterisk becomes mandatory for multiplications. White spaces would produce a syntactical error in this input format. On the other hand, white spaces in LaTeX do not necessarily indicate a multiplication. It could also be used to improve the readability. Whether a white space indicates a multiplication or not is a semantic information in the expression. We introduce a new DLMF macro that indicates a multiplication, but not gets rendered. This macro is `\idot`, for an invisible dot.

As already mentioned, the information is stored in three different file types. Together they form the backend library.

3.1.1 CSV Tables

The Comma-Separated Values (CSV) file represents the translation tables generated by MS Excel. Although CSV is an abbreviation for *comma-seperated*, one can specify the split symbol. Most common are semicolons. In our case we use semicolons, because commas are frequently used in string representations of mathematical functions to separate multiple parameters and

Semantic Macro	Entry in the CSV file
`\LegendreP{\nu}@{x}`	`\LegendreP{\nu}@{x}`
`\LegendreP[\mu]{\nu}@{x}`	`X1:\LegendrePX\LegendreP[\mu]{\nu}@{x}`

Table 4: CSV entry example of the Legendre and associated Legendre function of the first kind.

variables. Note that semicolons are also sometimes used, but CSV files are able to use quotation marks to separate a value with semicolon from the split symbols.

Note that CSV files are databases. While we do not want to introduce the theory of databases in this thesis, note that our databases contain a *primary key*. A primary key uniquely identifies and provides access to information in the database. The main CSV file is called `DLMFMacro.csv`. It provides information about each macro. The primary key of this database is the macro itself. Further information consists of the name of the mathematical function, the hyperlink to the DLMF definition, the numbers of parameters, variables and the maximum number of @ symbols. The maximum number of @ symbols is mostly used for backward translations. Additionally, the CSV file can contain information about the constraints, branch cuts and a role. The additional role can be used to ignore the macro in the translation process, or to specify that this macro is a mathematical constant. For example, mathematical constants are ignored, because they are organized in the JSON files. This avoid multiple definitions in the database.

Since a translation is not necessarily bijective, the translations are defined in two separate CSV files per CAS. For Maple those files are `DLMF_Maple.csv` for the forward translation, and `CAS_Maple.csv` for the backward translation. Both are organized in the same already described logic.

In 5 we defined the definition for the Gudermannian function. The translated expression contains two function calls: the inverse tangent function and the hyperbolic sine function. Instead of providing a hyperlink to the definitions for both functions on the Maple help page, we only provide one hyperlink. We specify the hyperlink by a special prefix in the translation pattern. To explain this prefix, we need to take a look to optional parameters before. While the primary key in the CSV files is only the macro name, we need to specify the number of optional parameters in front of the macro name. Otherwise, for example, the Legendre and the associated Legendre function of the first kind would be linked to the same definition. Our prefix notation solve both problems. It links to one specific function for the translation and specify the correct version of the macro, if there are optional parameters included. The prefix starts and ends with an *X* and contains two information. The number of optional parameters (if none, than this number will be 0) and the name of the used function. Table 4 illustrates the lexicon entry of the Legendre and the associated Legendre function of the first kind.

In Maple the number of optional arguments is not stored in the dictionaries. Therefore, we link to the total number of arguments. Consider Olver's associated Legendre function. A forward translation can be defined with

$$\texttt{\LegendreBlackQ\{\nu\}@\{z\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{LegendreQ(\nu,z)/GAMMA(\nu+1)}. \tag{11}$$

In this case, we would prefer to provide the hyperlink to the definition of Maple's `LegendreQ` function rather than to the definition of the `GAMMA` function. The used function has two arguments in Maple. Hence, the lexicon entry for the translation pattern is

```
X2:LegendreQXLegendreQ($0,$1)/GAMMA($1+1).        (12)
```

All of the CSV files just store the information to provide an easy access for the developer. Nevertheless, the program itself works only with the *lexicon files*. Therefore, each CSV file needs to get converted into a lexicon file. Subsection 3.2 explains how to do that and how to update the database.

3.1.2 JSON Libraries

Some translations are defined in separate JSON files. The reason is to separate translations for Orthogonal Polynomials and Special Functions (OPSF) from translation for single symbols and generic LaTeX macros. These files therefore define translations for the following types expressions.

- **Generic LaTeX macros for functions**:
  For example, the macros `\frac{a}{b}`, `\binom{a}{b}` and `\sqrt{a}`. This section also defines translations for special symbols for mathematical functions such as the exclamation mark '`!`' to indicate the factorial function or `\mod` for the modulo function.
- **Mathematical Symbols**:
  For example, the symbol to indicate multiplications, sums, inequalities.
- **Greek Letters**:
  For example, `\alpha`, `\beta` or `\Theta`.
- **Mathematical Constants**:
  For example `\cpi` for the constant π or `\EulerConstant` for the Euler-Mascheroni constant γ.

3.1.3 Lexicon Files

The knowledge of the Mathematical Language Parser (MLP) is based on lexicon files, as explained in subsection **??**. Since the MLP project was not a part of the DLMF/DRMF LaTeX macros project, the Backus-Naur Form (BNF) grammar defines no rules about semantic macros. Furthermore, the knowledge database has no information about the semantic macros. Therefore, we created a new lexicon file that defines each semantic LaTeX macro and provides further information about it. This information is stored in the previously mentioned CSV files.

Table 5 presents the entry for the sine function in the `DLMF-macros-lexicon.txt`. The `global-lexicon.txt` file is the main dictionary of the MLP and combines the whole knowledge of the parser in one

Symbol: \sin
Feature Set: dlmf-macro
DLMF: \sin@@{z}
DLMF-Link: dlmf.nist.gov/4.14# E1
Meanings: Sine
Number of Parameters: 0
Number of Variables: 1
Number of Ats: 2
Maple: sin($0)
Maple-Link: www.maplesoft.com/support/ help/maple/view.aspx?path=sin
help/maple/view.aspx?path=sin
Mathematica: Sin[$0]
Mathematica-Link: reference.wolfram.com/ language/ref/Sin

Table 5: The entry of the sine function in the lexicon file.

file. This lexicon files also help to translate ambiguous expressions. We will discuss some of these decision in the following sections. However, it is important to mention that a successful translation strongly depends on the comprehensive database of the lexicon files.

The concept of the lexicon files is also adapted to store the backward translations from Maple. Since the MLP project is developed to parse LaTeX expressions, it could be confusing to use the original lexicon files to also store names of Maple functions. Therefore, we create another type of lexicon files, which are very similar to the original lexicon files. In Maple a function can have multiple entries differing in the number of arguments. Therefore, the primary key of the backward translation database is the combination of the function name and the number of arguments. For example, the name `GAMMA` in Maple is reserved for the Euler Gamma function in case of one argument, but also for the incomplete Gamma function in the case of two arguments. Therefore the lexicon file for the backward translation also contains the number of arguments to specify a translation.

3.2 Extending the Forward Translator

The whole project is still work in progress. Therefore, it is an important goal to keep it easy to update the backend libraries. This section will explain the typical workflow to add new translations for semantic LaTeX macros and Maple functions or how to support a forward translation to another CAS.

3.2.1 Adding Translations to Existing CAS

Since CSV files are not as easy to maintain as MS Excel[5] worksheets, the workflow starts with MS Excel. Once a user updates the existing tables, the MS Excel file needs to get exported as a CSV file. The translator project contains the `lexicon-creator.jar` file to convert the CSV files to the backend lexicon files. Of course, these steps are not necessary if the user just changes a translation in the JSON files.

Once the `lexicon-creator.jar` finished the conversion process, the translator is aware of all changes and ready to use.

3.2.2 Supporting New CAS

To support a complete new CAS for the forward translation, the user needs to add the CAS in the JSON files and create two new CSV files. Even if there is no backward translation implemented yet, the CSV file for the backward translation is needed.

Once all of the files are updated, the `lexicon-creator.jar` can be used to update the lexicon files.

4 The Forward Translation

The translation process starts with semantic LaTeX expressions. These expressions get analyzed and parsed by the MLP, which produces a MLP-Parse Tree (PT) of the input expression with additional information about each symbol. An abstract translator class analyzes each node and delegates them to specialized subtranslators. In this process an object called Translated Expression Object (TEO) is build. This TEO is needed to rebuild a string representation of the mathematical expression given by the PT.

The overall forward translation process is explained in figure 2. All translation patterns and related information are stored in the DLMF/DRMF tables. These tables are converted by the `lexicon-creator.jar` to the `DLMF-macros-lexicon.txt` lexicon file. Together with the `global-lexicon.txt` file, the PT will be created by the MLP. The `latex-converter.jar` takes a string representation of a semantic LaTeX expression and uses the MLP as well as our `Translator` to create an appropriate string representation for a specified CAS. The following sections 4.1 to 4.4 will focus on the `Translator`.

4.1 Analyzing the MLP-Parse Tree

The main task is to analyze the PT. Remember that the PT is different to expression trees as described in section **??**.

[5]Microsoft Excel is widely used to work with tables and is able to import, export and manipulate CSV files.

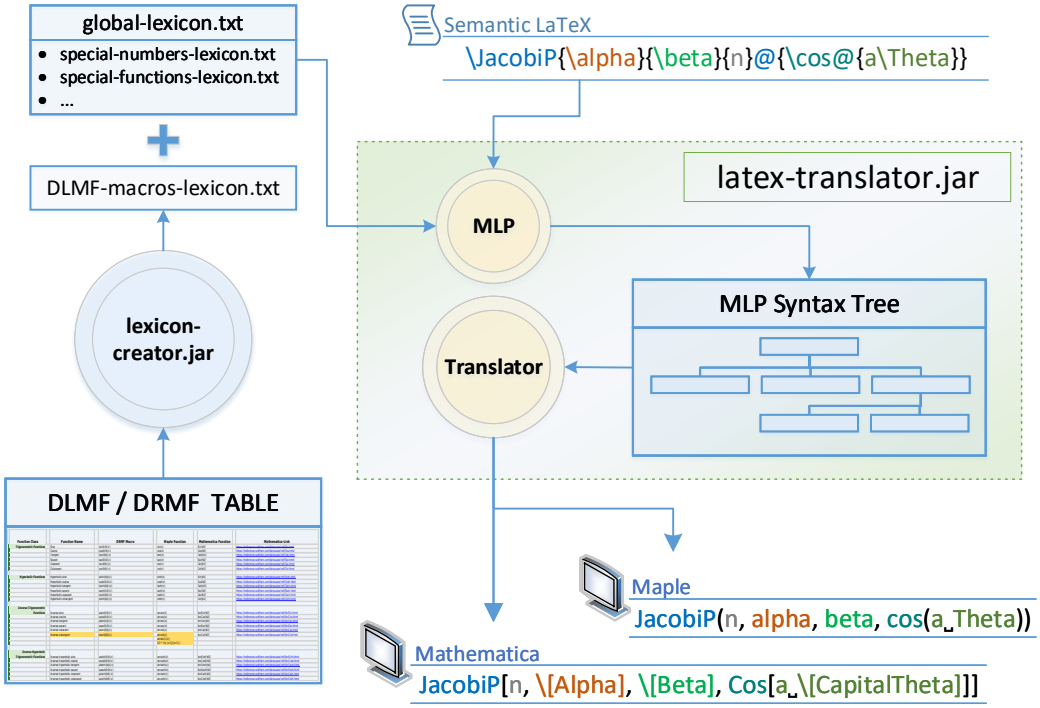


Figure 2: Process diagram of a forward translation process. The MLP generates the PT based on lexicon and JSON files. The PT will be translated to different CAS.

Since the BNF does not define rules for semantic macros, each argument of the semantic macro and each @ symbol are following siblings of the semantic macro node. That is the reason, why we stored the number of parameters, variables and @ symbols in the lexicon files. Otherwise, the translator could not find the end of a semantic macro in the PT.

Figure 3 visualizes the PT of the Jacobi polynomial example from table 1. Because of these differences to expression trees, a backward conversion of the PT to a string representation can be difficult, especially for finding necessary or unnecessary parentheses. Therefore we create the TEO. Of course, the TEO is deeply incorporated into the translation process, but in this subsection we will focus on the general idea and process of the translation. We will focus on the usage of TEO in subsection 4.3.

The general idea of a translation of a PT is a simple recursive algorithm, such as algorithm 1. Whenever the algorithm finds a leaf, it can translate this single term. If the node is not a leaf, it starts to translate all children of the node recursively.

But this approach is not completely feasible, because sometimes the algorithm needs to look ahead and check the following siblings for a valid translation, for example in case of a semantic macro with arguments (see

Algorithm 1 Simple translation algorithm for the MLP-Parse Tree

Input: Root r of the parse tree T

```
1: procedure TRANSLATE(r)
2:     if r is leaf then
3:         TRANSLATE_LEAF(r);
4:     else
5:         for all children v_n of r do
6:             TRANSLATE(v_n);
7:         end for
8:     end if
9: end procedure
```

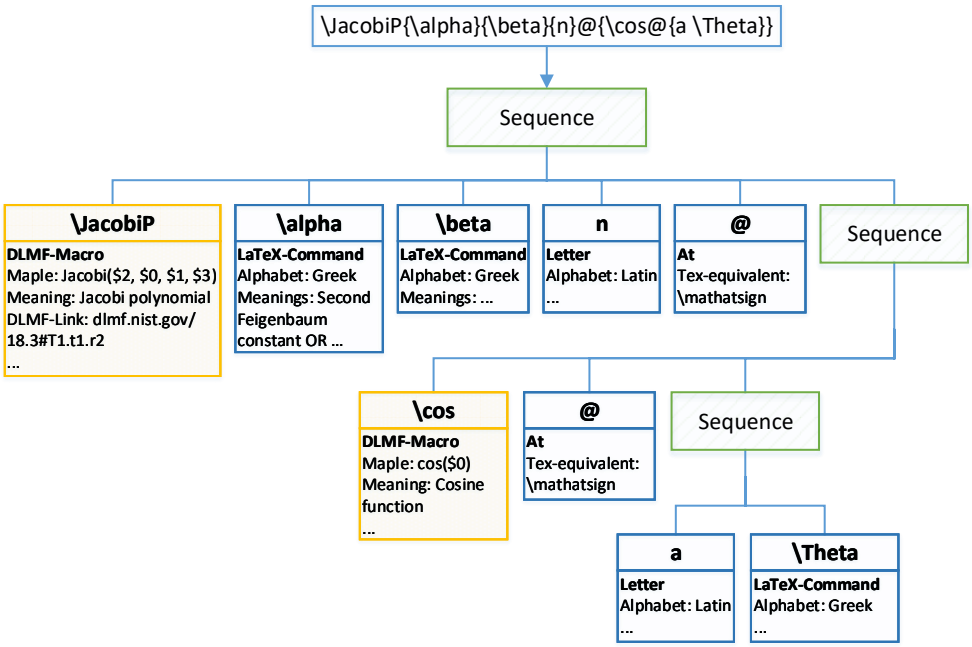


Figure 3: MLP-PT for a Jacobi polynomial using the DLMF/DRMF LaTeX macro. Each leaf contains information from the lexicon files.

figure 3). Therefore, the `TRANSLATE_LEAF` function also needs to know the following siblings of the current node. To avoid multiple translations of the same node, a list representation would be a better solution. Algorithm 2 explains an abstract version of the final algorithm for our translation tool.

If the root r is a leaf, it can be translated as a leaf. Eventually, some of the following siblings are needed in order to translate r correctly. If r is not a leaf, it contains one or more children. Therefore, we can call the `ABSTRACT_TRANSLATOR` recursively for the children. Once we have translated r, we can go a step further and translate the next node. Line 8 checks if there are following siblings left and calls the `ABSTRACT_TRANSLATOR` recursively in that case.

With this approach we are able to translate most of the semantic LaTeX expressions. To take a look at the problems with this approach, we need to think about the operators and their standard notations in mathematics. There are many types of notation used to represent formulae. For example, the Normal Polish Notation (NPN)[6] (hereafter called prefix notation) places the operator

[6]Also known as *Warsaw Notation* or *prefix notation*

Algorithm 2 Abstract translation algorithm to translate MLP-Parse trees.

Input: Root r of a MLP-Parse tree T. List *following_siblings* with the following siblings of r. The list can be empty.

```
1:  procedure ABSTRACT_TRANSLATOR(r, following_siblings)
2:      if r is leaf then
3:          TRANSLATE_LEAF(r, following_siblings_of_r);
4:      else
5:          siblings = r.getChildren();                ▷ siblings is a list of children
6:          ABSTRACT_TRANSLATOR(siblings.removeFirst(), siblings);
7:      end if
8:      if following_siblings is not empty then
9:          r = following_siblings.removeFirst();
10:         ABSTRACT_TRANSLATOR(r, following_siblings);
11:     end if
12: end procedure
```

to the left of/before its operands. The Reverse Polish Notation (RPN)[7] (hereafter called postfix notation) does the opposite and places the operator to the right of/after its operands. The infix notation is commonly used in arithmetic and places the operator between their operands, which only makes sense as long as the operator is a binary operator. For example, table 6 shows an expression in different notations.

Notation	Expression
Infix	$(a + b) \cdot x$
Prefix	$\cdot + a\ b\ x$
Postfix	$a\ b + x\ \cdot$
Functional	$\cdot(+(a, b), x)$

Table 6: The mathematical expression '$(a+b)\cdot x$' in infix, prefix, postfix and functional notation.

In mathematical expressions, notations are mostly mixed, depending on the case and number of operands. For example, infix notation is common for binary operators ($+, -, \cdot,$ mod etc.), while functional notations are obviously used for any kind of functions (*sin*, *cos*, etc.). Sometimes the same symbol is used in different notations to identify a different meaning. For example, the '$-$' as an unary operator is used in prefix notation to indicate the negative value of its operand, such as in '-2'. Of course, $-$ can also be the binary operator for subtraction, which is commonly used in infix notation. An example for the postfix notation is factorial, such as '2!'.

Most programming languages (and CAS as well) internally use prefix or postfix notation and do not mix the notations in one expression, because it is easier to parse those notations. However, the common practice in science is to use mixed notations in expressions. Since the MLP has barely implemented mathematical grammatical rules, it takes the input as it is and does not build

[7] Also known as *postfix notation*

an expression tree. Therefore, it parses all four examples from table 6 to four different PTs rather than to one unique expression tree. In general, this is not a problem for our translation process because most CAS are familiar with the most common notations. Therefore, the translator does not need to know that a and b are the operands of the binary operator '+' in '$a + b$'. We could just translate the symbols in '$a + b$' in the same order as they appear in the expression and the CAS would understand it. However, this simple approach generates two problems.

1. The translated expression is only syntactically correct if the input expression was syntactically correct.
2. We cannot translate expressions to a CAS which uses a different notation.

Problem 1 should be obvious. Since we want to develop a translation tool and not a verification tool for mathematical LaTeX expressions, we can assume syntactically correct input expressions and produce errors otherwise. Problem 2 is more difficult to solve. If a user wants to support a CAS that uses prefix or postfix notation by default, the whole translator would fail in its current state. Supporting CAS with another notation would be a part of future work and will be mentioned in chapter **??**.

Nonetheless, changing a notation could also solve ambiguities in some situations. Consider the two ambiguous examples in table 7. While a scientist would probably just ask for the right interpretation of the first example, Maple automatically computes the first interpretation. On the other hand, LaTeX automatically disambiguate the first example by only recognizing the very next element (single symbols or sequence in curly brackets) for the superscript and therefore displays the second interpretation. The second example is already interpreted as the double factorial function of n, since this notation is the standard interpretation in science. We wrote the second interpretation as the standard way in science to make it even more obvious. However, surprisingly, Maple computes the first interpretation again rather than the common standard interpretation.

	Text Format Expression	First Interpretation	Second Interpretation
1:	4 ^ 2!	$4^{2!}$	$4^2!$
2:	n!!	$(n!)!$	$(n)!!$

Table 7: Ambiguous examples of the factorial and double factorial function. One expression in a text format can be interpreted in different ways.

In most cases, parentheses can be used to disambiguate expressions. We used them in table 7 to clarify the different interpretations in example 2. But sometimes, even parentheses cannot solve a mistaken computation. For example, there is no way to add parentheses to force Maple to compute $n!!$ as the double factorial function. Even $(n)!!$ will be interpreted as $(n!)!$. Rather than using the exclamation mark in Maple, one could also use the functional notation. For example, the interpretations (2!)! and (2)!! can be distinguished in Maple by using `factorial(factorial(2))` and `doublefactorial(2)` respectively.

As already mentioned, the structure of the PT makes it difficult to change the notation for all kind

of operators. Therefore, and especially because of the examples above, we change the notation during the forward translation process to the functional notation only for the factorial and double factorial function. Thus,

$$\texttt{n!} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{factorial(n)},$$
$$\texttt{n!!} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{doublefactorial(n)}.$$

The translator needs to presume some properties for the functions similar to LaTeX, which only recognizes the very next element right after the caret for the superscript.

The biggest problem about translating the factorial (or double factorial) function is the common postfix notation for them. In this stage, algorithm 2 only translates the current node and if necessary the following siblings, but not the preceding siblings. Consider an expression such as '4!'. When the variable *r* reaches the exclamation mark, the argument '4' of the function has already been translated. The current version of the algorithm has no access to this previously translated expression. Even more complicated, would be the case of the double factorial '4!!'. The following two approaches would solve this problem.

1. The translator always checks if the next two following siblings are one or two exclamation marks.
2. The current state of program has access to the previously translated expressions and can modify them retrospectively.

Approach 1 would be unnecessarily inefficient. Therefore we implement approach 2. That is another reason why we implement an extra object to organize translated expressions, the TEO. This object stores all parts of the previously translated expressions. The translator has access to these parts and is able to modify them afterwards. A more detailed explanation about the TEO follows in subsection 4.3.

	Node type	Explanation	Example
***r* has children**	Sequence	Contains a list of expressions.	$a + b$ is a sequence with three children (a, + and b).
	Balanced Expression	Similar to a sequence. But in this case the sequence is wrapped by `\left` and `\right` delimiters.	`\left(`$a + b$`\right)` is a balanced expression with three children (a, + and b).
	Fraction	All kinds of fractions, such as `\frac`, `\ifrac`, etc.	`\ifrac{a}{b}` is a fraction with two children (a and b).
	Binomial	Binomials	`\binom{a}{b}` has two children (a and b).
	Square Root	The square root with one child.	`\sqrt{a}` has one child (a).
	Radical with a specified index	n-th root with two children.	`\sqrt[a]{b}` has two children (a and b).
	Underscore	The underscore '_' for subscripts.	The sequence a_b has two children (a and '_'). The underscore itself '_' has one child (b).
	Caret	The caret '^' to for superscripts or exponents. Similar to the underscore.	The sequence $a\hat{}b$ has two children (a and '^'). The caret itself '^' has one child (b).
***r* is a leaf**	DLMF/DRMF LaTeX macro	A semantic LaTeX macro	`\JacobiP`, etc.
	Generic LaTeX macro	All kinds of LaTeX macros	`\Rightarrow`, `\alpha`, etc.
	Alphanumerical Expressions	Letters, numbers and general strings.	Depends on the order of symbols. $ab3$ is alphanumerical, while $4b$ are two nodes (4 and b).
	Symbols	All kind of symbols	'@', '$*$', '+', '!', etc.

Table 8: A table of all kinds of nodes in a MLP syntax tree. Note that this table groups some kinds for a better overview. For a complete list and a more detailed version see Youssef, 2017.

4.2 Delegation of Translations

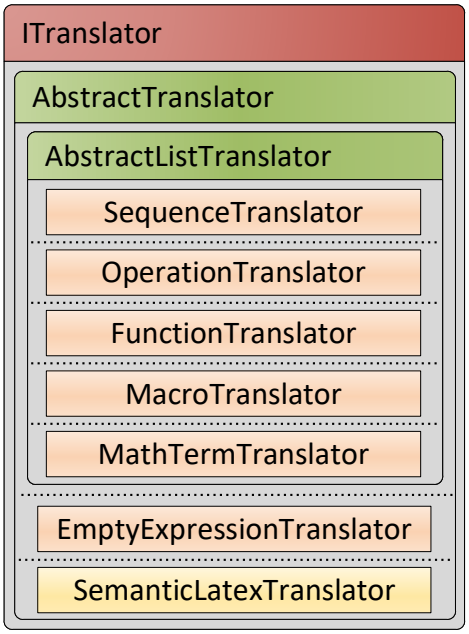


Figure 4: A scheme of the forward translator and its specialized subtranslators.

Algorithm 2 is a simplified version of the translator process. The main progress happens in lines 3 and 6. There are several cases what r can be. Table 8 on page 18 gives an overview of all different types in a PT. A more detailed explanation of the types can be found in (Youssef, 2017).

The BNF grammar defines some basic grammatical rules for generic LaTeX macros, such as for `\frac`, `\sqrt`. Therefore, there is a hierarchical structure for those symbols, similar to the structure in expression trees. As already mentioned, some of these types can be translated directly, such as Greek letters, while others are more complex, such as semantic LaTeX macros. Therefore, the translators delegates the translation to specialized subtranslators. This delegation process is implemented in lines 3 and 6 of algorithm 2.

Figure 4 is a scheme of the forward translator and its subtranslators. The entry point for each translation process of the program is the `SemanticLatexTranslator`. It delegates the translations for the expression through the `AbstractTranslator` to the other subtranslators. The `AbstractTranslator` is the interface of all other translators and coordinates the translation process by delegating translations for specific nodes to specialized classes. For example, the `MacroTranslator` only translates semantic LaTeX macros. The `EmptyExpressionTranslator` is used to delegate the so called *empty expressions* to other classes. An *empty expression* is a node with children, but without a symbol itself, such as *sequences* and *balanced expressions*.

4.2.1 Subtranslators

The most interesting translators are the `MacroTranslator` (translating semantic LaTeX macros) and the `SequenceTranslator` (translating all kinds of *sequences*).

The `SequenceTranslator` translates the *sequence* and *balanced expressions* in the PT. If a node n is a leaf and the represented symbol an open bracket (parentheses, square brackets and so on) the following nodes are also taken as a *sequence*. Hence, combined with the recursive translation approach, the `SequenceTranslator` also checks balances of parentheses in expressions. An expression such as '$(a]$' is producing a mismatched parentheses error. On the other hand, this is a problem for interval expressions such as '$[a, b)$'. In the current version, the program cannot distinguish between mismatched parentheses and an half-opened, half-closed interval. Whether

an expression is an interval or another expression is difficult to decide and can depend on the context. In addition, the parenthesis checker could simply be deactivated to allow mismatched parentheses in an expression. But this functionality is not implemented yet.

Another problem that the `SequenceTranslator` solves is the position of multiplication signs in an expression. There are a couple of obvious choices to translate multiplication signs. For example, using `\cdot` or `\idot` will be obviously translated as a multiplication symbol. However, `\idot` is fairly new and therefore not frequently used yet. Furthermore, the most common symbol for multiplications is still the white space or none symbol at all, as explained above. Consider the simple expression $2n\pi$. The PT generates a sequence node with three children, namely 2, n and π. This sequence should be interpreted as a multiplication of the three elements. Therefore, the `SequenceTranslator` checks the types of the current and next nodes in the tree to decide if there should be a multiplication symbol or not. For example, if the current or next node is an operator, a relation symbol or an ellipsis there will be no multiplication symbol added. However, this approach implies an important property. The translator interprets all sequences of nodes as multiplications as long as it is not defined otherwise. This potentially produces strange effects. Consider an expression such as $f(x)$. Translate this to Maple will be `f*(x)`. But we do not consider this translation to be wrong, because there is a semantic macro to represent functions. In this case, the user should use `\f{f}@{x}` instead of `f(x)`.

Besides the translation of sequences, the translation process of the DLMF/DRMF LaTeX macros can be complex too; not only because of the structure of the PT, but also because of the complex definition of the semantic macros. As explained in **??**, they can contain optional parameters, exponents are allowed before and after the arguments and the number of @ symbols varies. All these different cases will be solved by the `MacroTranslator`.

Algorithm 3 is the translation function of the `MacroTranslator` without error handling. It analyzes the next element right after the macro first. As mentioned, this can be one of the following:

- an exponent such as '`^2`'.
- an optional parameter in square brackets.
- a parameter in curly brackets (a *sequence* node in the PT) if none of the above.
- an @ symbols if none of the above.
- a variable in curly brackets (a *sequence* node) if none of the above.

An exponent will be translated and shifted to the end of the translated semantic macro, since it is more common to write the exponent after the arguments of a function in CAS. Therefore, the function translates and stores the exponent in line 5. One could ask what happens when there is an exponent given before and after the arguments. The `MacroTranslator` only translates the following siblings until each argument is translated. The first exponent will be shifted to the end. If right after the translated macro (with all arguments) follows another exponent, we interpret it as another exponent for the whole previous expression. In that case, it would be the macro with

Algorithm 3 The translate function of the MacroTranslator. This code ignores error handling.

Input:

macro - node of the semantic macro.
args - list of the following siblings of *macro*.
lexicon - lexicon file

Output:

Translated semantic macro.

```
1:  procedure TRANSLATE_MACRO(macro, args, lexicon)
2:      info = lexicon.getInfo(macro);
3:      argList = new List();              ▷ create a sorted list for the translated arguments.
4:      next = args.getNextElement();
5:      if next is caret then
6:          power = translateCaret(next);
7:          next = args.getNextElement();
8:      end if
9:      while next is [ do                 ▷ square brackets indicate optional arguments.
10:         optional = TRANSLATE_UNTIL_CLOSED_BRACKET(args);
11:         argList.add(optional);
12:         next = args.getNextElement();
13:     end while
14:     argList.add( TRANSLATE_PARAMETERS(args, info) );
15:     SKIP_AT_SIGNS( args, info );
16:     argList.add( TRANSLATE_VARIABLES(args, info) );
17:     pattern = info.getTranslationPattern();
18:     translatedMacro = pattern.fillPlaceHolders(argList);
19:     if power is not null then
20:         translatedMacro.add(power);
21:     end if
22:     return translatedMacro;
23: end procedure
```

the first translated exponent. Table 9 shows an example for the trigonometric cosine function with multiple exponents.

As you can see, the input expression in semantic LaTeX is interpreted as (13).

$$\left(\cos(x)^2\right)^2 \tag{13}$$

An open square bracket right after the semantic macro (after the exponent respectively) is an optional parameter. Since expressions in square brackets are not considered to be a *sequence* in the PT, the optional parameter is not grouped in a single node. Therefore, every node after the opening square bracket forms an expression for just one optional parameter until the algorithm reaches the closing square bracket. Such expressions in square brackets will be also translated by

Displayed As	$\cos^2(x)^2$
Semantic LaTeX	`\cos^2@{x}^2`
Translated Maple Expression	`((cos(x))^(2))^2`

Table 9: A trigonometric cosine function example with exponents before and after the argument.

the `SequenceTranslator` as described above. A semantic macro could have multiple optional parameters. Therefore, the function translates all optional parameters and stores them in line 9.

The lines 14 to 16 translates the parameters and variables. The @ symbols will be skipped. Nevertheless, the current program has an error handling implemented in line 15 if the maximum number of @ symbols is exceeded. In the following lines, the function takes the translation pattern from the lexicon and removes each placeholder by the corresponding argument. In addition, if there is an exponent right after the semantic macro, it will be attached to the translated semantic macro (see table 9). The parameters and variables are always grouped by curly brackets in *sequence*-nodes in the PT. Therefore, every argument after the optional parameters and potential exponent cannot be longer than one node in the PT.

4.2.2 Handle Ambiguities

Our semantic LaTeX expressions are not full semantic with respect to possible ambiguities in the expression. We already mentioned some techniques to solve ambiguities in semantic LaTeX expressions, for example how to handle double factorials. In table 10 are four examples of ambiguous expressions. LaTeX implemented the rules that the superscript or subscript is only one symbol (or a balanced expression in curly brackets). Therefore, each of these input expressions are not ambiguous in LaTeX. Since we talking about the forward translation, we should follow the rules of LaTeX to interpret ambiguous expressions.

Ambiguous Input	LaTeX Output
`n^m!`	$n^m!$
`a^bc^d`	$a^b c^d$
`x^y^z`	Double superscript error
`x_y_z`	Double subscript error

Table 10: Ambiguous LaTeX expressions and how LaTeX displays them.

Another more questionable translation decision are alphanumerical expressions. As explained in table 8, the MLP handles strings of letters and numbers differently, depending on the order of the symbols. The reason is, that an expression such as '4b' is usually considered to be a multiplication of 4 and 'b', while 'b4' looks like indexing 'b' by 4. While the first example produces two nodes, namely 4 and 'b', the second example 'b4' produces just a single alphanumerical node in the PT. The first expression would be translated as a multiplication of 4 and 'b'. Because '4b' looks like '$4 \cdot b$' and the multiplication is commutative, we would assume that '4b'

and '$b4$' are equivalent. This is one reason why we interpret such alphanumerical expressions as multiplications of the symbols. On the other hand, an expression such as '*energy*', would be an alphanumerical node as well and could be interpreted as a single variable named 'energy'. In that case, someone could wonder why the variable '*energy*' was translated as '$e \cdot n \cdot e \cdot r \cdot g \cdot y$'. While in physics or engineering variables possibly appear with longer names, such as the 'energy' example, it is more common to use single symbols for variables in mathematics (Cajori, 1994). Therefore, we interpret such alphanumerical expressions as multiplications of variables rather than one variable with an alphanumerical name.

Another, more elegant way to solve this problem would be to use the newly created semantic macro `\idot` (see subsection **??**) to define an invisible multiplication symbol. However, because `\idot` is such a new semantic macro, it is not used in the DLMF or DRMF quite often yet. Therefore, our program still interprets alphanumerical expressions as multiplications.

Other ambiguities can appear, when the input expression contains a symbol, which is typically used to represent a specific mathematical object, but the user did not use the semantic macro to represent it. For example, the input expressions '$3i$' contains an 'i', which is usually associated with the imaginary unit. Since there is a semantic macro for the imaginary unit, namely `\iunit`, we do not translate the letter to the imaginary symbol in Maple and use the identity translation instead. Furthermore, we inform the user about the potential misusage of `i` (see 4.4).

In general the translator is drafted to solve ambiguous expressions or automatically find a workaround to disambiguate the expression. Only if there is no way to solve the ambiguity with the defined rules, the translation process stops.

4.3 Translated Expression Objects

It is sometimes necessary to provide access to already translated symbols or subexpressions. A typical example is the (double) factorial function, where the argument was translated before the function was registered by the translator (postfix notation). To realize this, the Translated Expression Object (TEO) is implemented. It is mainly a list object of translated expressions. It also groups subexpressions when it becomes clear that multiple symbols belong together. This typically happens when something is wrapped in parentheses but also when a function is completely translated.

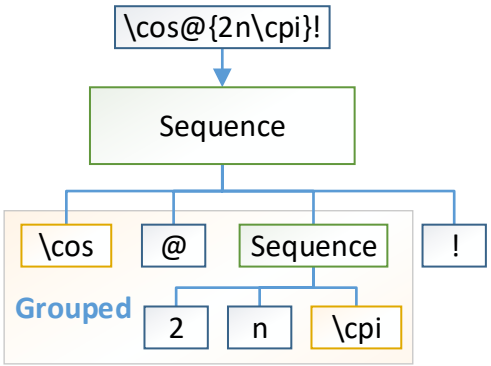


Figure 5: The MLP-PT for '$\cos(2n\pi)!$' and the grouped argument of the factorial function.

Figure 5 shows an example for the expression '$\cos(2n\pi)!$'. If we just analyze the PT, the previous node of the exclamation mark (which indicates the factorial function) is the sequence. Now it

could be possible to misinterpret the sequence as the argument of the factorial function. But the translator will first translate the cosine function, which has one argument. In that case the argument is the sequence node. Once the cosine function has been completely translated (which means the whole argument has been translated as well) the TEO groups the translated cosine and the sequence. Therefore, when the translator reaches the exclamation mark and asks for the last translated object in the list, it is not the translated sequence but the whole cosine function.

Table 11 shows some examples of the TEO list after the translation process was finished. Note that a translation of '$a + b$' contains three translated objects, while '$(a + b)$' contains just one.

Input Expression	TEO List
$a + b$	`[a, +, b]`
$(a + b)$	`[(a+b)]`
$\frac{a}{b} - 2$	`[(a)/(b), -, 2]`

Table 11: How the TEO-list groups subexpressions.

4.3.1 Local & Global Translated Expressions

During the translation process multiple local TEO are created by the subtranslators. Besides the local TEO there is just one global TEO. The reason to create local objects is originally motivated by the recursive structure of the translator. A subtranslator should only handle the current subexpression. However, the global TEO becomes necessary because of the functions in postfix notation such as the factorial function. For example, when a subtranslator tries to translate the exclamation mark as the factorial function, the local TEO is empty, because the argument of the function has been previously translated by another subtranslator and stored in another local TEO. Therefore, we implement a global TEO, which contains all of the previous translated subexpressions in each state of the translation process. With this implementation the local TEO becomes more and more obsolete. In the current version, the program stores redundant translated expressions. For debugging reasons, a translation process still handles multiple local and one global TEO. Starting the forward translation with the flag `--debug` (for the debugging mode) shows the elements of the global TEO after the translation has been finished.

4.4 Additional Information

As already mentioned, a translation is not always straight forward. In those cases, the user should get informed about the translation and the reason for it. Consider the translation

$$\texttt{3i} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{3*i}, \tag{14}$$

Since the imaginary unit in Maple is associated with '`I`' instead of '`i`', a user might be confused by 14. Therefore, we created the `InformationLogger` class to organize further information

about the translation process. In the case of 'i', the user is informed about the semantic macro \iunit and the potential misusage of 'i'.

Essentially, the InformationLogger provides further information about a translation process. This information is defined in the lexicon files. Consider the example of the arccotangent function, which is defined with a different branch cut in DLMF and in Maple. As already mentioned, we provide alternative translation patterns to solve these problems. These alternative translations, explanations, branch cuts, domains etc. will be automatically stored in the InformationLogger when an expression contains a mathematical object with further information.

5 Maple to Semantic LaTeX Translator

The backward translation is based on the internal data structure of Maple. Therefore, a license of Maple is mandatory to perform backward translations. The following subsections will explain why this approach was necessary and how Maple works internally. Using the internal data structure also requires to handle the internal properties of Maple. This can causes some problems. Those problems and our solutions will be explained in subsection 5.3. In the last subsection 5.4, we will focus on the implementation of the backward translation.

Note that all explanations in this section is based on Maple's programming guide (Bernardin et al., 2016) for Maple 2016.

5.1 Internal Data Structure

While our forward translation is realized for the 1D input representation in Maple, this representation is not suitable for a backward translation. Since Maple uses it's own syntax and programming language, we would need a new parser for Maple expressions to realize a backward translation tool. Of course, Maple has this parser implemented. But it is not possible to use the parser without using Maple itself. Therefore, an installed version of Maple is mandatory for our backward translation.

As already mentioned, Maple has several ways to represent an expression. The 1D and 2D representations are used as input representations. Besides those, there are internal data structures to perform all kinds of computations on the expression. Internally, each expression is handled and stored as a Directed Acyclic Graph (DAG). The Maple DAG is very similar to an expression tree. But for efficiency, it does not store copies of an object. Consider the integral from (**??**)

$$\int_0^\infty \frac{\pi + \sin(2x)}{x^2} dx. \tag{15}$$

As already mentioned, the Maple DAG has no copies of objects. Therefore, the 'x' in (15) only appears once in the DAG.

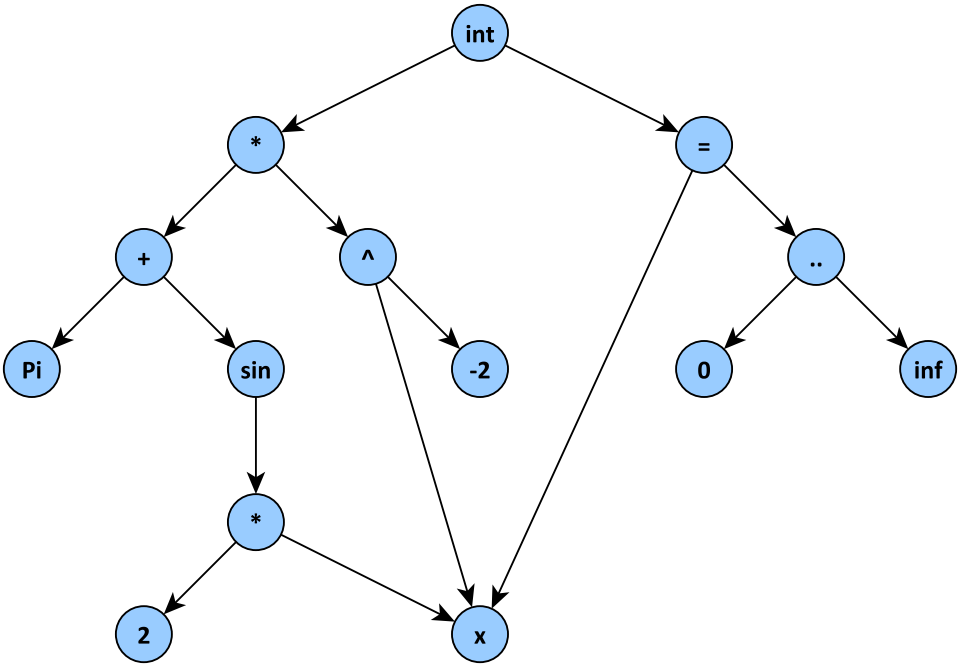


Figure 6: The Maple DAG of equation (**??**)

The Maple DAG can be visualized in a graph, such as in figure 6. But the internal representation looks a bit different. Each node in the DAG stores its children and has a header, which defines the type and the length of the node. Consider the polynomial $x^2 + x$. Figure 7 illustrates the internal DAG representation with headers and arguments.

Maple has also another representation, similar to the DAG, that allows multiple copies of the same object in its representation. This representation is called `InertForm` and is more similar to an expression tree than Maple's DAG. The `InertForm` represents the DAG as a tree by splitting object copies and displays the tree in a list structure. Each node (or element in the list) has the prefix `_Inert_XXX`, that indicates the node is in the `InertForm`. The suffix `XXX` is the type of the node. Some of the important types are specified in table 12. Note that this is only a subset of all possible types.

5.2 Maple's Open Maple API

Maple provides an Application Programming Interface (API), called `OpenMaple` (Bernardin et al., 2016, §14.3) for interaction with the Maple kernel. The `OpenMaple` API is implemented for different programming languages such as `C`, `Java` and `Visual Basic`. Since our forward translation and the MLP is implemented in `Java`, we use the `Java` API for the backward translation. The `Java` API has some limitations to interact with Maple's DAG.

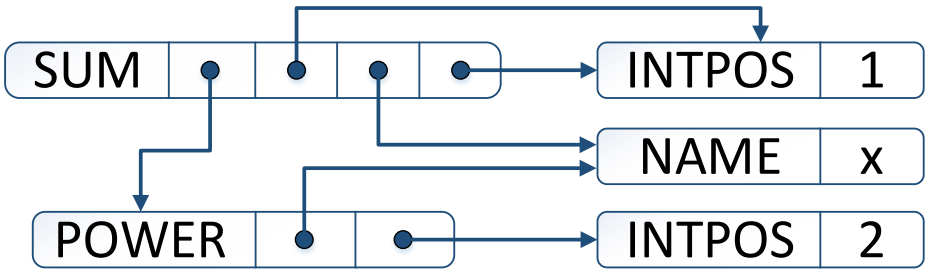


Figure 7: The internal Maple DAG representation of $x^2 + x$.

For the backward translation, we use the `InertForm`. This representation can be converted to a nested list version and the `OpenMaple` Java API can handle lists. Therefore, the backward translation firstly converts the `InertForm` to a nested list structure before any translation process starts.

The following figure 8 shows the `InertForm` and the nested list of the polynomial $x^2 + x$.

```
_Inert_SUM(
    _Inert_POWER(
        _Inert_NAME("x"),
        _Inert_INTPOS(2)
    ),
    _Inert_NAME("x")
)
```

```
[_Inert_SUM,
    [_Inert_POWER,
        [_Inert_NAME, "x"],
        [_Inert_INTPOS, 2]
    ],
    [_Inert_NAME, "x"]
]
```

Figure 8: `InertForm` and the nested list representation of $x^2 + x$.

In the nested list, the first element specifies the type of the current list while the following arguments store the arguments and optional attributes.

5.3 Workarounds for Problems

Note that the backward translations usually do not have problems with ambiguous expressions. Every CAS needs to solve ambiguities in its own representation, otherwise it could not compute the expression. Our backward translation from Maple back to semantic LaTeX is therefore based on already disambiguated expressions.

The general walkthrough of our backward translation starts with a string of a `1D` Maple input expression. As described above, this string is entered in Maple to parse the expression and work on the nested list version of the `InertForm` afterwards. This causes some problems. Mostly because Maple automatically evaluates input expressions immediately - even before we have a chance to take a look at the internal data structure of the input. For example, an input such as

Type	Explanation
SUM	Sums.
PROD	Products.
EXPSEQ	Expression sequence is a kind of list. The arguments of functions are stored in such sequences.
INTPOS	Positive integers.
INTNEG	Negative integers.
COMPLEX	Complex numbers with real and imaginary part.
FLOAT	Float numbers are stored in the scientific notation with integer values for the exponent n and the significand m in $m \cdot 10^n$.
RATIONAL	Rational numbers are fractions stored in integer values for the numerator and positive integers for the denominator.
POWER	Exponentiation with expressions as base and exponent.
FUNCTION	Function invocation with the name, arguments and attributes of the function.

Table 12: A subset of important internal Maple structures. See (Bernardin et al., 2016) for a complete list.

$\sin\left(\frac{\pi}{2}\right)$ is immediately evaluated to 1 and the internal data structure just shows us the positive integer value 1 instead of the trigonometric function.

Furthermore, Maple also changes input expressions slightly, mostly because of missing data types for a specific expression. For example, the internal data structure has no type to represent fractions except the `RATIONAL` type, which only allows integer values for the numerator and denominator. Thereby, an expression like $\frac{x}{2}$ is automatically changed to $x\frac{1}{2}$. There is no way no avoid such simple arithmetic changes in the expression, but we can avoid the automatic evaluations.

Firstly, we summarize all known changes and evaluations Maple automatically performs on input expressions.

- Maple evaluates input expressions immediately.
- There is no data type to represents square roots such as $\sqrt{x}$ (or n-th roots). Therefore, Maple stores roots as an exponentiation with a fractional exponent. For example, $\sqrt{x}$ is stored as $x^{\frac{1}{2}}$.
- There is no data type for subtractions, only for sums. Negative terms are changed to absolute values times '-1'. For example, $x - y$ is stored as $x + y \cdot (-1)$.
- Floating point numbers are stored in the scientific notation with a mantissa and an exponent in the base 10. For example, 3.1 is internally represented as $31 \cdot 10^{-1}$.
- There is only a data type for rational numbers (fractions with integer numerator and positive denominator), but not for general fractions, such as $\frac{x+y}{z}$. This will be automatically

changed to $(x + y) \cdot z^{-1}$.

There are unevaluation quotes implemented to avoid evaluations on input expression. Table 13 gives an example how those unevaluation quotes work.

	Without unevaluation quotes	With unevalation quotes
Input expression:	`sin(Pi)+2-1`	`'sin(Pi)+2-1'`
Stored expression:	1	`sin(Pi)+1`

Table 13: Example of unevaluation quotes for `1D` Maple input expressions.

Since we want to keep a translated expression similar to the input expression, we use unevaluation quotes during the whole translation process. Unevaluation quotes also solve the problems with square roots and n-th roots. In Maple's `1D` input representation, a square root is a function call (`sqrt(x)`) and the unevaluation quotes prevent evaluations on functions. Therefore, a backward translation simply needs to translate a square root as a normal function. Hence, we do not have problems with the internal representation of roots in Maple.

Because of the absent data type for subtractions, a long sum with negative terms could be difficult to read. We change the order of constants (such as an integer value) to be in front of products. Therefore, an expression such as '$-y$' is not represented by '$y \cdot (-1)$', but by '$(-1) \cdot y$'. The translator now needs to check if there is a leading '-1' and can translate it to '$-y$' rather than to '$(-1) \cdot y$'.

Floating point numbers in scientific notation also causes a problem. Consider the input expression 0.41 as the nested list `InertForm` representation

```
[_Inert_FLOAT, [_Inert_INTPOS, 41], [_Inert_INTNEG, 2]].        (16)
```

The backward translator is implemented in `Java`. To translate such an expression back to 0.41, computations become necessary. Of course, it makes sense to perform those computations in the CAS (in that case, in Maple) rather than in the translation program. Therefore, we implemented a procedure[8] to automatically convert `FLOAT` nodes to a string representation of the floating point number. We created a new type, called `MYFLOAT` to represent such numbers. With this approach, we avoided computations in the translation process.

For fractions, we use a similar approach and introduce a new internal data type `DIVIDE`. The implementation of `DIVIDE` was difficult. On the one hand, some expressions with negative exponent should be displayed as a fraction while others should should be displayed with the negative exponent. We follow the same rule as Maple follows internally to display fractions. When the exponent is a negative numeric element, the expression should be displayed as a fraction. If the exponent is something else, we uses the negative exponent representation. Table 14 shows two examples with negative exponents. One of them is displayed as a fraction (exponent '-2') and the other is not (exponent '$-2y$'). We implement the same rule for our newly developed `DIVIDE`

[8] *Procedures* in Maple are small programs similar to methods and functions in Object-Oriented Programming (OOP) languages.

data type. Note that with the new created data type, we are still not able to avoid changes from $\frac{x}{2}$ to $x\frac{1}{2}$, but we are able to work with more general fraction expressions.

	Negative numeric exponent	General negative exponent
1D Maple input expression	`(x-1)^(-2)`	`(x-1)^(-2*y)`
How Maple displays the input	$\frac{1}{(x-1)^2}$	$(x-1)^{-2y}$

Table 14: Different styles to display negative exponents depending on the type of the exponent. Maple displays expressions with negative numeric exponents as fractions, while other negative exponents are not displayed as fractions.

We implement all of these changes before the actual translation process starts. This helps keeping the translated expression similar to the input expression.

5.4 Implementation Details

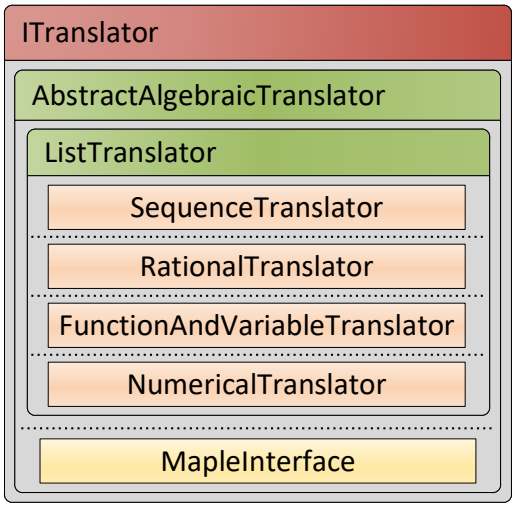


Figure 9: A scheme of the backward translator and its specialized subtranslators.

The general translation process runs through different states. Each translation process starts with the string representation of a `1D` Maple input. This string will be converted to the internal nested list representation of our custom `InertForm` (see the previous subsection 5.3). Since the `InertForm` is similar to an expression tree, it can be translated as the PT in the forward translation. Hence, we use the same approach of multiple subtranslators for the backward translation process.

Figure 9 shows the subtranslators of the backward translation process. The structure is the same as for the forward translation, seen in figure 4. The `MapleInterface` represents the entry point of a translation process. Before the translation starts, the Maple kernel needs to be initialized and prepared for the translation process by loading all custom procedures. Once everything is loaded, an expression is delegated to specialized subtranslators depending on the internal data type, defined by custom procedures and Maple's `InertForm`.

The following figure 10 illustrates the backward translation process for the Jacobi polynomial example $P_n^{(\alpha,\beta)}(\cos(a\Theta))$ from table 1. The input expression is converted to the nested list representation of the `MyInertForm` (the customized `InertForm` of Maple). Afterwards, each node

of the tree is translated separately by specialized subtranslators (visualized by blue arrows). The translation of functions (bold blue arrows) is again realized by translation patterns, which define the correct position for each argument. The last step is used to put all translated arguments to the right position (visualized by red arrows).

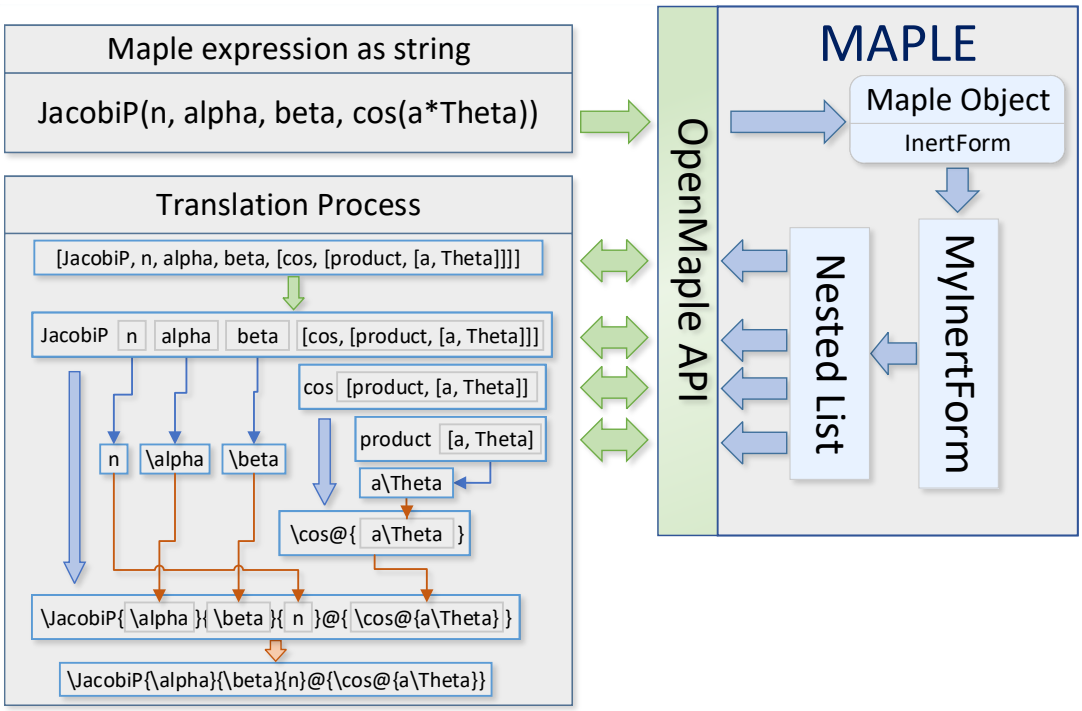


Figure 10: A scheme of the backward translation process from Maple for the Jacobi polynomial $P_n^{(\alpha,\beta)}(\cos(a\Theta))$. The input string is converted by the Maple kernel into the nested list representation. This list is translated by subtranslators (blue and red arrows). A function translation (bold blue arrows) is again realized by translation patterns to define the position of the arguments (red arrows).

This project aims for a translation of mathematical expressions from LaTeX sources to CAS and back again. Essentially, the translator translates only string representations. We need to verify that such translations are *appropriate*. Therefore, we develop verification and validation techniques for translations and present them in this chapter.

First of all, each semantic LaTeX macro ties a specific character sequence to a well-defined mathematical object. This definition can be found at the DLMF or DRMF. Therefore, our reference source for semantic LaTeX expressions is always the DLMF or DRMF. This gives us the opportunity to compare more than the string representations of the mathematical objects in a translation process. For example, consider the simple translation of the cosine function

$$\texttt{\textbackslash cos\{x\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{cos(x)}. \tag{17}$$

Our verification and validation techniques, to proof for example that (17) is an *appropriate* translation, can be grouped into three parts. We present in the following sections these three parts and conclusively, we use all these techniques to actually verify translations for a test suite directly extracted from the DLMF.

6 Round Trip Tests

Round trip tests are our first approach to verify translations. The idea is to translate an expression back and forth again and check if the representation changes during this process. The translator is able to translate expressions from and to Maple. Therefore, we implement the following round trip tests only for Maple.

Furthermore, we take advantage of Maple's symbolic simplification function to compare a Maple expression in the beginning of an round trip test with the Maple expression after the translations. If Maple's *simplify* function returns zero for the difference between two representations, both expressions are symbolically equivalent. Note that this relies on the correctness of the simplification, which is not proven (see section **??**). On the other hand, the equivalence is not disproven if the simplification returns something different to zero.

A round trip test always starts with a valid expression either in semantic LaTeX or in Maple. A translation from one system to another is called **a step**. A complete round trip translation (two steps) is called **one cycle**.

Definition 6.1: *(Fixed Point Representation)*
*A **fixed point representation** (or short fixed point) in a round trip translation process is a string representation that is identical to all string representations in the following cycles.*

Obviously, there may appear two fixed point representations in round trip tests, one for each system. We will see that most of the round trip translations reach fixed points after a certain amount of steps.

Theorem 6.1: *(Fixed Point Properties)*
Consider two languages $A, B \in \mathcal{L}$. A fixed point representation is reached if the current representation is just identical to the next representation in the same language (after one cycle). Furthermore, if $a \in A$ is a fixed point representation, the translation of a to $b \in B$ is also a fixed point representation.

Proof. Let $A, B \in \mathcal{L}$ be languages. Consider a round trip translation test between the languages A and B. Without loss of generality, let $a \in A$ be a fixed point representation and let a be the *first* fixed point representation that appears in this test. The MLP as well as Maple are deterministic, because a test case works in only one Maple-session and we do not compute the inputs in Maple. Therefore, the translation process is not able to call any probabilistic algorithms. Hence, our translator is also deterministic.

Let $b \in B$ with

$$\mathrm{tr}_B^A(a) = b. \tag{18}$$

Since the translation process is deterministic, a will be always translated to b. Consider $a' \in A$ as the backward translation from b

$$\mathrm{tr}_A^B(b) = a'. \tag{19}$$

Now a was defined to be a fixed point representation in A. Therefore, a is identical to all string representation in the following cycles. Thus $a = a'$.

Furthermore, because the translations are deterministic, b will be always backward translated to $a' = a$ and a will be always forward translated to b. Therefore, b is identical to the representation after one cycle. Per definition, b is therefore a fixed point in B.

Remember, a was chosen arbitrarily as the first fixed point representation that appears in the round trip test. Therefore, whenever a fixed point appears in a round trip test, the next translation to the respective other language will be a fixed point as well. Consequently, a fixed point is already reached, when just the next representation (after one cycle) is identical to the current representation. □

With theorem 6.1, we are able to find fixed points in round trip tests by only testing the equality to the next representation (after one cycle). Furthermore, we only need to find the first fixed point representation in either system, because the next step will be a fixed point representation for the respective other system. When a round trip test reaches such a fixed point in Maple, the *simplify* function tries to simplify the difference between the first expression in Maple and the fixed point expression. It is not feasible to simplify or compute LaTeX expressions yet (neither generic nor semantic LaTeX) - actually, our translation tool just makes this possible. Therefore, we check the equivalence between a fixed point and its first expression in semantic LaTeX manually.

Table 15 illustrates an example of a round trip test which reaches a fixed point. The test formula is

$$\frac{\cos(a\Theta)}{2}. \tag{20}$$

Steps	semantic LaTeX/Maple representations
0	`\frac{\cos@{a\Theta}}{2}`
1	`(cos(a*Theta))/(2)`
2	`\frac{1}{2}\idot\cos@{a\idot\Theta}`
3	`(1)/(2)*cos(a*Theta)`
4	`\frac{1}{2}\idot\cos@{a\idot\Theta}`

Table 15: A round trip test reaching a fixed point.

Step 4 is identical to step 2. Theorem 6.1 implies that step 2 is the fixed point representation for semantic LaTeX and step 3 is the fixed point representation for Maple. The structure of the expression changes between the steps 1 and 2. The reason are internal changes made by Maple. These problems have been discussed in the previous section 5.3.

Almost all test cases are reaching a fixed point representation. Starting from semantic LaTeX, the first fixed point is reached at step 2 in semantic LaTeX, because of Maple's internal changes. If the structure of the semantic LaTeX input is already identical to the internal structure of Maple's representation and the expression only contains functions and symbols with an available direct translation from and to each system, the input expression is already the fixed point representation. For example, consider the semantic LaTeX expression in step 2 in table 15 as the input expression. In that case the input is already a fixed point. Starting a round trip test from Maple, the first fixed point representation is reached in step 1 in semantic LaTeX, because Maple's changes has already been performed after the first translation. Again, this is only true if the expression only contains functions and symbols with a direct translation from and to each system. Furthermore, if a test case reaches a fixed point, it is equivalent (checked by Maple's *simplify* function or manually) to the input expression.

There is currently only one exception known where a round trip test does not reach a fixed point representation at all: Legendre's incomplete elliptic integrals (Olver et al., 2017, (19.2.4-7)) are defined with the amplitude ϕ in the first argument in the DLMF, while Maple takes the trigonometric sine of the amplitude as the first argument. Therefore, the forward and backward translations are defined as

$$\texttt{\textbackslash EllIntF@\{\textbackslash phi\}\{k\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{EllipticF(sin(phi),k)}, \tag{21}$$

$$\texttt{\textbackslash EllIntF@\{\textbackslash asin@\{\textbackslash phi\}\}\{k\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{EllipticF(phi,k)}. \tag{22}$$

A round trip translation for this function produces an infinite chain of sine and inverse sine calls. The problem is that the round trip tests prevent simplifications in Maple during the translation process (see section 5.3). This problem could be resolved by allowing automatic evaluations in the round trip tests. However, currently we do not allow those evaluations because the changes can be serious, which makes it hard to follow possible errors.

The round trip tests are very successful, but they only detect errors in string representations. However, because of the simplification techniques of fixed points, we are able to at least detect

logical errors in one system: Maple. On the other hand, these tests cannot determine logical errors in the translations between the two systems - but this is the critical part we want to verify. Consider defining a forward and backward translation for the sine function obviously wrong as

$$\texttt{\textbackslash sin@\{\textbackslash phi\}} \overset{\mathfrak{M}_{aple}}{\leftrightarrow} \texttt{cos(phi)}, \tag{23}$$

$$\texttt{\textbackslash cos@\{\textbackslash phi\}} \overset{\mathfrak{M}_{aple}}{\leftrightarrow} \texttt{sin(phi)}. \tag{24}$$

In that case the round trip test would not detect any errors and reach a fixed point representation, because the simplification techniques only simplify two representations in the same system but cannot compare the representation in one system to those in the other. Therefore, we implemented tests for equations rather than general formulae.

7 Function Relation Tests

In addition to the correctness of string representations, it is more important to validate whether a translation is *appropriate*. Special functions are highly related to each other. The theory of special functions researches these relations and other properties of the functions. The DLMF is a compendium of special functions and lists a lot of those relations. The translation of such relations to another system and the following analysis of the relations with the translated formulae would form a very effective test.

With this technique we can detect translation errors such as in (23 and 24). Consider the DLMF equation for the sine and cosine function (Olver et al., 2017, (4.21.2))

$$\sin(u+v) = \sin u \cos v + \cos u \sin v. \tag{25}$$

Assume the translator would forward translate the expression based on (23, 24). Than

$$\texttt{\textbackslash sin@\{u + v\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{cos(u + v)}, \tag{26}$$

$$\texttt{\textbackslash sin@@\{u\}\textbackslash cos@@\{v\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{cos(u)*sin(v)}, \tag{27}$$

$$\texttt{\textbackslash cos@@\{u\}\textbackslash sin@@\{v\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{sin(u)*cos(v)}. \tag{28}$$

This produces the equation in Maple

$$\cos(u+v) = \cos u \sin v + \sin u \cos v, \tag{29}$$

which is wrong. Since the expression is correct before the translation, we conclude an error during the translation process.

However, there are two essential problems with this approach. Testing the mathematical equivalence of expressions is hard to solve and CAS have trouble to test even simple equations. Furthermore, this approach only checks forward translations because there is no way to check

equivalence of expressions in LaTeX automatically (again this could become feasible with our translator). We use Maple's *simplify* function to check if the difference of the left-hand side and the right-hand side of the equation is equal to zero. In addition, we use *simplify* and check if the division of the right-hand side by the left-hand side returns a numerical value or not. This simplification function is the most powerful function to check the equivalence in Maple. However, there are several cases where the simplification fails. We implement some tricks to help *simplify*: pre-conversion of the functions of the expression to other special functions is able to help *simplify*, because the algorithms in the background can use different simplification methods for different functions. For example, the user can use the Maple function *convert* to convert the expression

$$\sinh x + \sin x \tag{30}$$

to an equivalent representation using the exponential function:

$$\frac{1}{2}e^{x} - \frac{1}{2}e^{-x} - \frac{1}{2}i\left(e^{ix} - e^{-ix}\right). \tag{31}$$

With such pre-conversions we are able to improve the simplification process in Maple. However, the limitations of the *simplify* function are still the weakest part of this verification approach. Consider the equation (**??**) from **??**.

$$U(0,z) = \sqrt{\frac{z}{2\pi}}K_{\frac{1}{4}}\left(\frac{1}{4}z^{2}\right) \tag{32}$$

As already discussed, the modified Bessel function of the second kind (Olver et al., 2017, (12.7.10)) has its branch cut in Maple at $z < 0$. However, the argument of K contains z^2. For now, just focus on

$$K_{\frac{1}{4}}\left(\frac{1}{4}z^{2}\right) \tag{33}$$

Because of z^2, if $|\mathrm{ph}(z)| \in \left(\frac{\pi}{2}, \pi\right)$ the value of (33) would be no longer on the principal branch. However, Maple will still compute (33) on the principal branch independently of the value of z. Hence, a translation

$$\texttt{\textbackslash BesselK\{\textbackslash frac\{1\}\{4\}\}@\{\textbackslash frac\{1\}\{4\}z\^{}2\}} \overset{\mathfrak{M}_{aple}}{\mapsto} \texttt{BesselK(1/4,(1/4)*z\^{}2)} \tag{34}$$

is incorrect if $|\mathrm{ph}(z)| \in \left(\frac{\pi}{2}, \pi\right)$ and one has to use analytic continuation for the right-hand side of equation (32). In subsection **??** we also shown that the complete right-hand side of equation (32) has three branch cuts, caused by the square root function and the z^2 argument of the modified Bessel function of the second kind. Such complex examples cannot be simplified yet.

However, this validation approach is powerful and runs automatically. We use this approach to verify a large set of test cases extracted directly from the DLMF, see section 9.

8 Numerical Tests

Another approach of verification are numerical tests. In principle, we follow the same approach as the relation tests in section 7, but rather than using the simplification functions we compare the left-hand side and right-hand side for actual values.

For example, reconsider equation (32) from (Olver et al., 2017, (12.7.10)). Define the difference of right-hand side and left-hand side

$$D(z) := U(0, z) - \sqrt{\frac{z}{2\pi}} K_{\frac{1}{4}}\left(\frac{1}{4}z^2\right), \tag{35}$$

and compute $D(z)$ for some values of z. Table 16 presents four computations for $D(z)$, one value for each quadrant in the complex plane.

z	$D(z)$
$1 + i$	$2 \cdot 10^{-10} - 2 \cdot 10^{-10} i$
$-1 + i$	$2.222121916 - 1.116719816 i$
$-1 - i$	$2.222121916 + 1.116719816 i$
$1 - i$	$2 \cdot 10^{-10} + 2 \cdot 10^{-10} i$

Table 16: Four computations of $D(z)$ in Maple.

The values are computed with the default precision of 10 significant digits. Caused by the machine accuracy, the first and last values are different from zero. In such cases, we can increase the precision to figure out if it is an accuracy issue or $D(z)$ is really different from zero at those points. Increasing the precision of the computation for the first value $z = 1 + i$ up to 100 significant digits returns $3 \cdot 10^{-100} i$ and can therefore be considered as zero (we get the same results for $z = 1 - i$). Hence, table 16 shows us what we already know. The second and third values are computed on the principal branch of the modified Bessel function of the second kind and therefore produces a difference between the left-hand side and right-hand side of equation (32).

This approach of verification is very powerful, because it can verify not only our translations but also equations in a mathematical compendium. This has been demonstrated by discovering an overall sign error of the DLMF equation (36), see section 9.

However, this approach has also a significant problem. It is easy to see that something went wrong, but very hard to proof equivalence. Obviously, only four tested values are not sufficient to proof the equivalence. How many and which values do we need for a comprehensive result? Reformulating this question to a general problem brings us to:

> *How can we discretize equations down to a finite set of test points such that the probability of equivalence for the continues equation is high if the equation is valid for the finite set of test points?*

As far as we know, there is no answer to this question yet. This is one reason why we do not implement an automatic process of numerical tests yet. Another reason are constraints. Constraints are not as easy to translate as general formulae. They often contain quantifiers or logical operators which are difficult to translate in the current state of the program. However constraints are necessary for numerical tests.

9 DLMF Test Suite

To verify our translations with the approaches described above we create test suites by extracting equations and formulae from the sources of the DLMF. We start with a small test suite of 110[9] equations and provide a more detailed look to those tests. We are able to translate 108 of those 110 cases.

Two of the test cases (lines 42 and 43) contain macros we are not able to translate yet. Those macros are `\subplus` and `\CFK`, which are used for continued fractions. They have been created for another translation project (Cohl et al., 2017, §5) for continued fractions defined in the *Handbook of Continued Fractions for Special Functions* (2008) (A. A. M. Cuyt et al., 2008). The conversion process for the Continued Fractions for Special Functions (CFSF) (Backeljauw and A. Cuyt, 2009) dataset from Maple and the Encoding Continued Fraction Knowledge (eCF) (Trott and E. W. Weisstein, 2013; Trott, E. Weisstein, et al., 2013) dataset from Wolfram uses special file types. Therefore, our translator is not able to handle those macros yet.

However, we translate 108 test cases and for 84 of them the simplification function of Maple returns zero as the difference of the left-hand side and right-hand side. The 24 remaining cases are translated, but the simplification function is not able to simplify the difference to zero. Therefore, we try to improve this part. Line 46 (Olver et al., 2017, (18.5.10)) for example, can be simplified to zero, if n is previously set to an arbitrary integer value. For all those 24 test cases, we perform also manual numerical tests, which reveal an overall sign error in equation (ibid., (14.5.14))[10]

$$\mathsf{Q}_{\nu}^{-1/2}(\cos\theta) = -\left(\frac{\pi}{2\sin\theta}\right)^{1/2} \frac{\cos\left(\left(\nu+\frac{1}{2}\right)\theta\right)}{\nu+\frac{1}{2}}. \tag{36}$$

Besides this wrong equation, all of these manual numerical tests return zero for the difference between the left-hand side and right-hand side. Of course it is still not possible to conclude the equivalence based on our manual numerical tests, as explained above.

Those 110 test cases are not sufficiently comprehensive for all of the functions of the DLMF. Therefore, we create a larger dataset[11] of 4,165 semantic LaTeX formulae, which is extracted from all chapters of the DLMF.

[9] The set of test cases is available at
`https://gist.github.com/AndreG-P/f48121d70fbd2a57b2cf6d88019ddadf`

[10] The equation had originally been stated as shown in equation (36). The DLMF has fixed the wrong equation with version 1.0.16.

[11] We are planning to publish this dataset at `http://drmf.wmflabs.org`.

We are able to translate 2,232 (approx. 53.59%) of those test cases. The *simplify* function of Maple returns zero for the difference for 490 of the translated 2,232 test cases. We increase the number to 713 with automatic pre-conversions helping the simplification process. With simplification over the division of the left-hand side by the right-hand side (rather than simplify the difference) we have three more verified translations and 716 in total. Therefore, 1,516 test cases were translated but the translation cannot be verified automatically yet.

The translator cannot translate 1,933 test cases. First of all, 32 lines contain neither an equation nor a relation. Because we have created this dataset for our verification approaches, we do not translate those 32 lines at all. Most of the other failures are produced because of missing semantic information in the LaTeX expression or the formulae contains a macro that cannot be translated in the current state of the program, such as in 978 lines that contain a macro that cannot be translated. For example, in 171 lines one of the symmetric elliptic integrals (Olver et al., 2017, (19.16.1-6)) appears and we do not provide translation patterns for those integrals, because there is no direct translation possible yet. A workaround would be to allow translations based on the definitions, see (4).

Other errors appear in 914 cases, where the semantic information is missing or is not unique. An example are 284 lines that use a potentially ambigious single quote (or prime) notation ’’ to identify derivatives of functions.

References

Alex, G. (June 2007). “Do Open Source Developers Respond to Competition? The (La)TeX Case Study”. In: *Review of Network Economics* 6.2, pp. 1–25.

Ausbrooks, R., Buswell, S., Carlisle, D., Chavchanidze, G., Dalmas, S., Devitt, S., Diaz, A., Dooley, S., Hunter, R., Ion, P., Kohlhase, M., Lazrek, A., Libbrecht, P., Miller, B., (deceased), R. M., Rowley, C., Sargent, M., Smith, B., Soiffer, N., Sutor, R., and Watt, S. (2014). *Mathematical Markup Language (MathML) 3*. Tech. rep. Version 3.0 2nd Edition. International Organization for Standardization (ISO).

Backeljauw, F. and Cuyt, A. (July 2009). “Algorithm 895: A Continued Fractions Package for Special Functions”. In: *ACM Trans. Math. Softw.* 36.3, 15:1–15:20.

Bernardin, L., Chin, P., DeMarco, P., Geddes, K. O., Hare, D. E. G., Heal, K. M., Labahn, G., May, J. P., McCarron, J., Monagan, M. B., Ohashi, D., and Vorkoetter, S. M. (2016). *Maple 2016 Programming Guide*. Maplesoft, a division of Waterloo Maple Inc.

Cajori, F. (Mar. 1, 1994). *A History of Mathematical Notations*. Dover Publications Inc. 848 pp.

Churchill, B. and Boyd, S. (2010). *LaTeXCalc*. `https://sourceforge.net/projects/latexcalc/`. Seen 06/2017.

Cohl, H. S., Schubotz, M., Youssef, A., Greiner-Petter, A., Gerhard, J., Saunders, B. V., McClain, M. A., Bang, J., and Chen, K. (2017). “Semantic Preserving Bijective Mappings of Mathematical Formulae Between Document Preparation Systems and Computer Algebra Systems”. In: *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*. Ed. by H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke. Vol. 10383. Lecture Notes in Computer Science. Springer, pp. 115–131.

Corless, R. M., Jeffrey, D. J., Watt, S. M., and Davenport, J. H. (2000). “"According to Abramowitz and Stegun" or arccoth needn’t be uncouth”. In: *ACM SIGSAM Bulletin* 34.2, pp. 58–65.

Cuypers, H., Cohen, A. M., Knopper, J. W., Verrijzer, R., and Spanbroek, M. (June 2008). “MathDox, a system for interactive Mathematics”. In: *Proceedings of EdMedia: World Conference on Educational Media and Technology 2008*. Ed. by J. Luca and E. R. Weippl. Vienna, Austria: Association for the Advancement of Computing in Education (AACE), pp. 5177–5182.

Cuyt, A. A. M., Petersen, V. B., Verdonk, B., Waadeland, H., and Jones, W. B. (2008). *Handbook of Continued Fractions for Special Functions*. Springer.

Davenport, J. H. (2010). “The Challenges of Multivalued "Functions"”. In: *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*. Ed. by S. Autexier, J. Calmet, D. Delahaye, P. D. F. Ion, L. Rideau, R. Rioboo, and A. P. Sexton. Vol. 6167. Lecture Notes in Computer Science. Springer, pp. 1–12.

Olver, F., Olde Daalhuis, A., Lozier, D., Schneider, B., Boisvert, R., Clark, C., Miller, B., and Saunders, B., eds. (2017). *NIST Digital Library of Mathematical Functions*. `http://dlmf.nist.gov/`, Release 1.0.14 of 2017-12-21.

Drake, D. (June 2009). *sagetex*. `https://ctan.org/tex-archive/macros/latex/contrib/sagetex/`. Seen 06/2017. Comprehensive.

England, M., Cheb-Terrab, E. S., Bradford, R. J., Davenport, J. H., and Wilson, D. J. (2014). "Branch cuts in Maple 17". In: *ACM Comm. Computer Algebra* 48.1/2, pp. 24–27.

Foundation, W. (Jan. 2001). *Wikipedia*. `https://en.wikipedia.org`.

Geuvers, H., England, M., Hasan, O., Rabe, F., and Teschke, O., eds. (2017). *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*. Vol. 10383. Lecture Notes in Computer Science. Springer.

Giceva, J., Lange, C., and Rabe, F. (2009). "Integrating Web Services into Active Mathematical Documents". In: *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference, MKM 2009, Held as Part of CICM 2009, Grand Bend, Canada, July 6-12, 2009. Proceedings*. Ed. by J. Carette, L. Dixon, C. S. Coen, and S. M. Watt. Vol. 5625. Lecture Notes in Computer Science. Springer, pp. 279–293.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley.

– (June 11, 1998). *Digital Typography*. Reissue. Lecture Notes (Book 78). Center for the Study of Language and Information (CSLI). 685 pp.

Kohlhase, M. (2006). *OMDoc - An Open Markup Format for Mathematical Documents [version 1.2]*. Lecture Notes in Computer Science. Springer.

– (2008). "Using LaTeX as a Semantic Markup Format". In: *Mathematics in Computer Science* 2.2, pp. 279–304.

Kohlhase, M., Corneli, J., David, C., Ginev, D., Jucovschi, C., Kohlhase, A., Lange, C., Matican, B., Mirea, S., and Zholudev, V. (2011). "The Planetary System: Web 3.0 & Active Documents for STEM". In: *Proceedings of the International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June, 2011*. Ed. by M. Sato, S. Matsuoka, P. M. A. Sloot, G. D. van Albada, and J. Dongarra. Vol. 4. Procedia Computer Science. Elsevier, pp. 598–607.

Miller, B. R. and Youssef, A. (2003). "Technical Aspects of the Digital Library of Mathematical Functions". In: *Ann. Math. Artif. Intell.* 38.1-3, pp. 121–136.

Research, W. (July 2011). *Computable Document Format (CDF)*. `http://www.wolfram.com/cdf/`.

Trott, M. and Weisstein, E. W. (May 2013). *Computational Knowledge of Continued Fractions*. `http://blog.wolframalpha.com/2013/05/16/computational-knowledge-of-continued-fractions`. seen 3/2017.

Trott, M., Weisstein, E., Marichev, O., and Rowland, T. (Oct. 2013). *The eCF-Project – Final Report*. Tech. rep. Wolfram Foundation.

Youssef, A. (2017). "Part-of-Math Tagging and Applications". In: *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*. Ed. by H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke. Vol. 10383. Lecture Notes in Computer Science. Springer, pp. 356–374.