Theorema Project: Document Processing
Entwurfsversion vom 3. Dezember 2024
(mit alternativem Deckblatt)

Jack C. Heseltine, BA

B A C H E L O R A R B E I T

eingereicht am

Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg, Fachhochschule Oberösterreich, in Kooperation mit RISC,
dem Research Institute Symbolic Computation (Johannes Kepler Universität Linz)
zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

im September 2024

Advisor:

Assoc. Univ.-Prof. DI Dr. Wolfgang Windsteiger

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, Fachhochschule Oberösterreich, in Kooperation mit RISC,
dem Research Institute Symbolic Computation (Johannes Kepler Universität Linz)
zur Erlangung des akademischen Grades
Bachelor of Science in Engineering, September 30, 2024

Jack C. Heseltine, BA

Contents

Preface

I would like to thank, in chronological order as pertains to this thesis, the following persons who enabled this course of study and thesis in Hagenberg, both in the professional and the educational context:

Abstract

This work explores the Wolfram Language as a Software Engineering tool, with a particular focus on the Theorema mathematical software package, in combination with the LaTeXtypesetting system. It delves into the advanced functionalities and paradigms of Wolfram Language, including high-level programming, functional programming, and pattern matching, to showcase these capabilities beyond object oriented programming languages in particular, as applied to mathematical document transformation.

Through Theorema, package development using Wolfram Language is demonstrated from conception through execution to the point that the new package can be easily integrated with the existing Theorema system: the associated analysis touches on the workings of Theorema but the focus is on an implementational bridge between computational mathematics and document preparation, aiming to provide easy extensibility and delivering on further Software Engineering principles to make for a rounded Wolfram Language and Theorema package, as the final project output.

The thesis also addresses the challenges and methodologies associated with the LaTeXtypesetting of mathematical content, emphasizing the transformation of Wolfram Language/Theorema notebooks using a Wolfram-Language-native approach. This includes an examination of first-order predicate logic symbols, to ensure coverage at the output side, and the role of (mathematical) expressions in Wolfram Language, the input side, showcasing back-and-forth between typesetting and (symbolic) computational languages, and particularly, recursive parsing of entire notebook expressions as the basic working principle in this approach.

Kurzfassung

Diese Arbeit untersucht die Wolfram Language als ein Software Engineering Werkzeug, mit einem besonderen Fokus auf das mathematische Softwarepaket Theorema in Kombination mit dem LaTeX-Textsatzsystem. Sie vertieft sich in die fortgeschrittenen Funktionalitäten und Paradigmen der Wolfram Language, einschließlich der High Level Programmierung, der funktionalen Programmierung und des Pattern Matchings, um diese Fähigkeiten auch über die objektorientierte Programmierung hinaus zu demonstrieren, besonders im Hinblick auf die Transformation mathematischer Dokumente.

Mittels Theorema wird die Entwicklung von Packages in der Wolfram Language von der Konzeption über die Ausführung bis zu dem Punkt, an dem das neue Paket problemlos in das bestehende Theorema-System integriert werden kann, demonstriert. Die zugehörige Analyse befasst sich mit der Funktionsweise von Theorema, wobei der Schwerpunkt auf einer implementierungstechnischen Brücke zwischen computergestützter Mathematik und Dokumentenvorbereitung liegt. Das Ziel ist es, einfache Erweiterbarkeit zu ermöglichen und weitere Prinzipien des Software Engineerings zu realisieren, um ein umfassendes Wolfram und Theorema Language Package als Endprodukt des Projekts zu liefern.

Die Arbeit thematisiert auch die Herausforderungen und Methodologien, die mit dem LaTeX-Textsatz von mathematischem Inhalt verbunden sind, und beleuchtet besonders die Transformation von Wolfram/Theorema-Notebooks nativ in der Wolfram Language. Dies beinhaltet eine Untersuchung von Symbolen der Prädikatenlogik erster Ordnung, um die Abdeckung auf der Ausgabeseite sicherzustellen, und die Rolle von (mathematischen) Ausdrücken (Expressions) in der Wolfram Language, der Eingabeseite, die die gegenseitige Kommunikation von zwischen Textsatz- und (symbolischer) Programmiersprache aufzeigt, insbesondere das rekursive Parsen von ganzen Notebook Expressions als grundlegendes Arbeitsprinzip des gewählten Ansatzes.

Chapter 1

Introduction

1.1 Mathematica as Both a Tool and an Object of Study, at RISC: The Theorema Project

RISC has a notable relationship to Mathematica: a quick glance at the list of software packages produced by RISC ("Symbolic computation can be seen as the automation and algorithmization of mathematics. Therefore, most of what we do results in concrete software." [44]) demonstrates this: across different branches of Mathematics, many packages are *Mathematica* packages.

The Theorema Project is a long-standing effort originated by Bruno Buchberger and continued by Wolfgang Windsteiger to this day. It will be explored in some detail and with a view to extending it as a collection of Mathematica packages in the following chapter on Theory 2, where this thesis accompanies a practical work, using Wolfram Language (WL) as an engineering tool. Therefore this thesis also has an applications slant, focusing on various aspects of the development of such a package..

1.1.1 The Mathematical Document Problem: Syntax and Semantics

The publication format standard for mathematical and technical documents today is LaTeX, but there is a problem with this standard from a mathematics point of view: this system does not allow for mathematical evaluation of mathematical statements, in a world where such systems already exist: to actually work with a mathematical statement the user has to translate the LaTeXformat into another system's format and do evaluation there. In this sense, LaTeXdocuments are not live, they look good on paper, but do not *do* anything.

The Theorema Project and in a wider sense, the Wolfram Language ecosystem today (see appendix D), addresses this problem at the root by representing even the document itself as an expression that can be worked with in the same manner as any other expression contained in said document. Compound expressions of this sort are well-suited for representing especially First Order Predicate Logic (FOPL), or in Software Engineering Terms, they are suitable data structure. This is ultimately the reason a WL-native approach was chosen in this thesis' practical work, rather than say, text-based parsing into another representation format or programming language, for example XML or Java.

From this, this thesis will pay particular attention to WL as an engineering language and the process of designing and developing complex systems in a WL-environment.

1.1.2 Starting from Rewriting: Mathematica and Wolfram Language as a Programming Language

Mathematica is not Wolfram Language, where in "a first approximation, the Wolfram Language = Mathematica + Wolfram|Alpha [a knowledge-based web service] + Cloud [a storage-oriented web service] + more" [59], but this disambiguation will be made again in D.0.1: at the time Bruno Buchberger was writing, it was the same thing, and it is particularly in "Mathematica as a Rewrite Language" [7] that crucial analysis, here from the perspective of the field of rewriting, is made. Under the assumption that it is worth looking at the set of features that make the language stand apart, I would like to follow Buchberger's thoughts on Mathematica as a language for rewriting to begin.

Remarking on the stability of Mathematica, Buchberger observes that "Wolfram's pattern matching is essentially the natural concept of conditional rewriting." [7, p. 2] Writing for an intended audience of both mathematicians in rewriting and Mathematica developers (the groups he poses would benefit from mutual exchange, in [7]), Buchberger mostly offers a definition of the rewriting concept in terms of Mathematica syntax, also useful here. The foundational language constructs according to Buchberger are "constants, ordinary variables, sequence variables, expressions and conditional equalities (rewrite rules)" [7, p. 3]: in order, constants in Mathematica are user-defined and may have "built-in" meaning (like "Sin"); identifiers can be written as variables by following them immediately with an underscore as in "x_"; sequence variables specify one, no, or multiple occurrence and are introduced with three underscores after the identifier as in "x___" (BlankNullSequence is the Wolfram Language term [4] nowadays).

Mathematica expressions are described like this by Buchberger:

> Any constant and any ordinary variable is a Mathematica expression. If F is an expression that is not a variable and $T_1, \ldots, T_n$ are expressions or sequence variables then
> $$F[T_1, \ldots, T_n]$$
> is also an expression. (Such an expression is called a "compound expression" or "application expression".) [7, p. 4]

For example, integers, or "Sin[Plus[2,x_]]" are expressions. For an expression like Plus, or any generic F[x, y] (in standard form [17]), taking two or more arguments, there also exists the infix notation x ~ f ~ y [17], and for single-argument expressions, prefix and postfix using symbols, f @ x and x \\f [17] respectively - these shorthands taken together with the multi-decade process of adding new functions across mathematical and other domains led to modern WL syntax, along with the "Wolfram" re-branding of Mathematica, as a product.

Buchberger considers Mathematica function definitions (conditional) rewrite rules [7, p. 5] of the form:

$$lhsExpr := rhsExpr(/; condition)$$

The bracketed condition need not occur and *lhsExpr* may not be a variable, as in a simple assignment. Then, "Mathematica programs are just finite sequences of such (conditional) rewrite rules separated by semicolons." [7, p. 5] (The semicolon in Wolfram Language is a built-in symbol symbolizing a CompoundExpression [12], it is also used to suppress output of evaluation if placed immediately after an expression [46].) These ideas will be revisited in Chapter 2: the main point here is that the evaluation mechanism in Mathematica works such, that the rewrite rules, or definitions, are applied in a standardized order until no candidate definitions are left.

1.1.3   Wolfram Language Expressions, Comparison to Object Oriented Programming

Buchberger points out that 'the mechanism for associating rules with identifiers opens an immediate possibility for realizing "object oriented programming" in Mathematica' in [7, p. 7] by referencing The Mathematica Book of 1996 in turn, particularly up-values ("UpValues" in current Wolfram Language syntax) in Mathematica (at the time, Version 3.0): object oriented programming can be practised directly, beyond simply thinking of expressions as objects, in this way.

The `quat` object in question is an instance of "a class of abstract mathematical objects of type `quat`" [59, At the end of section 2.4.10, Associating Definitions with Different Symbols] Wolfram introduces to fulfill certain properties that "overload" arithmetic operations as an example. To "tag" Mathematica data (expressions) as quat objects would entail defining their heads like `quat[`*data*`]`. Here tagging is taken to mean specifying the type of an expression.

Upvalues are then used to specify the form arithmetic operations like addition take on when it comes to `quat` objects, see the following code example taken from [59, At the end of section 2.4.10, Associating Definitions with Different Symbols], where the following expression defines an upvalue for `quat` with respect to Plus (Quaternions [34]).

```
quat[x_] + quat[y_] ^:= quat[x + y]
```

That is, delayed assignments are associated with `quat`-objects (the "gs" of the relevant Wolfram TechDoc [71, in section Associating Definitions with Different Symbols]), so that at least since Mathematica Version 3.0 an evaluation of the following form would take place:

```
quat[a] + quat[b] + quat[c] = quat[a + b + c]
```

Or, "when you define an upvalue for quat with respect to an operation like Plus, what you are effectively doing is to extend the domain of the Plus operation to include quat objects. You are telling Mathematica to use special rules for addition in the case where the things to be added together are quat objects." [59, At the end of section 2.4.10, Associating Definitions with Different Symbols]

This becomes yet another interpretation of Wolfram Language expressions [63, The Meaning of Expressions].

1.1.4   Wolfram Language as a Document Representation Format

Since everything is an expression in Wolfram Language ("everything you type into the Wolfram Language is treated as an expression," [17]) so too is a complete notebook: A

quick look at the Test Notebook listing in Section A shows some comment-lines with metadata such as Mathematica version and internal caching information, but mostly one big `Notebook[]` expression, the "the low-level construct that represents a notebook manipulated by the Wolfram System front end." [38]

It in turn consists of `Cell[]`s, "the low-level representation of a cell inside a Wolfram System notebook." [10] Again, the structure is defined in terms of its front-end purpose, which is to organize input and output: examples can be found in [59].

The front-end allows switching between rendered cells and their expression format, using the keyboard shortcuts documented in [9], or the "Cell" menu item "Show Expression."

As in the FirstTour test notebook, the cells that make up the `Notebook[]` expression (collected in a list), mostly contain in `BoxData[]` and related structures (again, "low-level" representations [6]), used for typesetting: 'When Wolfram Language expressions are displayed in notebooks, they are represented by two-dimensional typesetting structures of "boxes"' [19] - thus, the typesetting mechanism for Wolfram Language notebooks is the combination of these box structures, also expressions of course, with cell- and notebook-expressions. Any parsing on a notebook therefore takes place in relation to these basic structural elements.

1.2   Motivation, Project Shape and Development Environment

The motivation for this project is two-fold: First, LaTeX is a standard widely used in technical and scientific communities, including journals of relevance to RISC, for example. Second, the native export functionality provided in Mathematica out of the box is not tailored to the Theorema background evaluation of formula representations despite their relevance to Theorema-publications.

The status quo this project seeks to address is the labor-intensive, manual LaTeX-preparation of formulae from Theorema notebooks for publication purposes.

1.2.1   Need: The LaTeX-Standard for Academic Publishing in Mathematical Disciplines

The LaTeX-typesetting software system is maintained by The Latex Project [29]: In their words, "LaTeX is the de facto standard for the communication and publication of scientific documents." [30]

The current thesis is not so much about the format as it is about WL as an engineering tool for the Theorema context and transforming WL/Theorema-notebooks to this format, but the short form history and overview feature list of the system shall be included here: It is based on Donald E. Knuth's TeX typesetting language, where LaTeX was first developed in 1985 by Leslie Lamport (the "La" in LaTeX), and currently by The LaTeX Project. [29]

The Latex Project lists the system's current set of features as follows [28].

- Typesetting journal articles, technical reports, books, and slide presentations.
- Control over large documents containing sectioning, cross-references, tables and figures.
- Typesetting of complex mathematical formulas.

- Advanced typesetting of mathematics with AMS (American Mathematical Society) LaTeX.
- Automatic generation of bibliographies and indexes.
- Multi-lingual typesetting.
- Inclusion of artwork, and process or spot colour.
- Using PostScript or Metafont fonts.

1.2.2   Shaping: Comparison to Existing Functionality in Mathematica

Mathematica provides native LaTeX-export functionality, drawing on AMS-LaTeX, already listed in in the features of the system: AMS-LaTeX extensions are included in the standard LaTeX distribution, where the "amsmath part is an extension package for LaTeX that provides various features to facilitate writing math formulas and to improve the typographical quality of their output." [2]

The .TeX-file produced in this way consists some lines implementing related (mathematical and graphics) package imports and document setup commands but for this project, the native Mathematica solution does not work well for Theorema notebooks, as is easily seen when attempting to use this feature for this project's main test object, FirstTour.nb (provided in the project files), the PDF-rendering of the native LaTeX-export (based on FirstTourNativeExport.tex, also provided in the project repository).

1.2.3   Development Environment: Tools Used

In the present work, Mathematica and WolframKernel 14.1 [59] are used throughout: Screenshots of Mathematica for desktop are used to show code evaluation where the frontend is relevant, and IntelliJ IDEA 2022.3.3 [27] is used in conjunction with the IntelliJ plugin Wolfram Language by Hal's Corner [60] for syntax highlighting, as a guide for setting up a modern WL development environment and note on the present work's tooling: to reproduce this setup, simply install Mathematica (may require a license) and IntelliJ, then the plugin inside of the IntelliJ settings.

1.2.4   Note on Platform-(In)dependence

Core concepts employed in this work to answer the challenge outlined in section 1.2, for example pattern matching, related to the symbolic approach already discussed and explored in depth in Sections 3.2 and 4.3, are central to WL and the platform as a whole, making backwards and forwards compatibility within the Mathematica ecosystem highly likely.

Since this work transforms Theorema documents and Theorema extends Wolfram Language, the tool developed here can be applied to vanilla Mathematica notebooks (Wolfram Language under the hood, that is, see section D.0.1 on the relevant terminology) as well: The application requires execution on a compatible WL kernel setup. So, while the package is not, in principle, dependent on Theorema, and will simply transform available patterns in the input data - if these are not present, there is limited transformation - it is entirely dependent on the Wolfram Kernel included with Mathematica distributions.

On the level of the operating system, this implementation is platform-*in*dependent and benefits from the Wolfram Language ecosystem setup (see criticisms in Section D.0.2) the way Theorema does, because (of) "Mathematica programs run without any modifications on essentially all available operating system platforms (Linux, OS X, and Windows), the powerful development group at Wolfram Research that keeps Mathematica being always an up-to-date platform growing into various directions, and the huge group of Mathematica users." [57, p. 72]

1.3 Vision

Figure 1.1 shows the Theorema commander: this project is a prototype that could be contained in the functionality exposed directly to the user, to "export as LaTeX" in addition to any native Mathematica functionality that has already been shown to be ill suited.
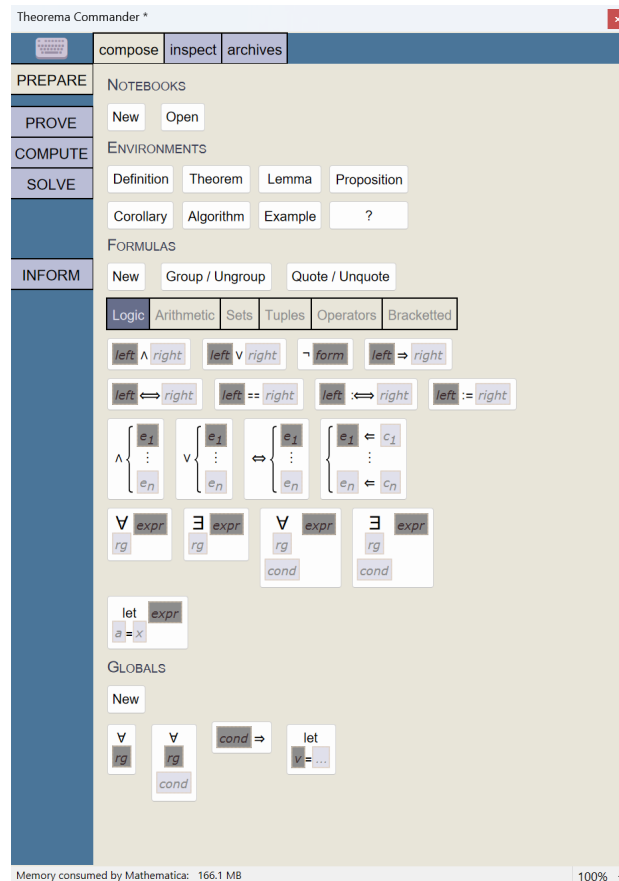
Figure 1.1: The start view in the Theorema Commander window: on Windows, the Commander opens in its own window, emulating an interface for a separate notebook.

Chapter 2

Theoretical Background: Theorema and Software Engineering, and the Project-Perspective on Programming Paradigms in Wolfram Language

2.1 Theorema Overview

Theorema is currently available in Version 2.0, under GPL [26] and inlcuding the full source code on GitHub [24]. A tutorial is available as well. [56]

Relating the original goal of The Theorema Project with the current project, this foreword excerpt contextualizes Theorema in the world of theorem provers (as of 2006) by comparing the system to 16 others in the same class of system:

> We can also see clearly from the examples in this collection that the notations for input and output have to be made more human readable. Several systems do generate LaTeX output for the discovered proofs, but perhaps additional thought about formatting output might be valuable. The Theorema Project (system 12 in the present list) made readability of proofs a prime requirement, and their report shows their success. However, the objective Prof. Bruno Buchberger set originally for the project was to produce a tool for pedagogic use, not research. [22, p. 4]

The proof mechanisms are not the subject of the present thesis so much as a look at the underlying data structure, implemented in Wolfram Language: we do want to situate ourselves in the logic-landscape, however.

2.1.1 Note on the Logic of Theorema

'The logic frame of Theorema is higher order predicate logic, which is extended by the language construct "sequence variables,"' [22, p. 110] already introduced in Section 1.1.2 and allowing for pattern matching capabilities. In this view, "the Theorema system is a (growing) collection of various general purpose and special theorem provers. The general

purpose provers (like the first order predicate logic natural deduction prover) are valid only for special fragments of predicate logic (e.g. first order predicate logic)." [22, p. 110] The more assumptions are made, the more specialized the prover module that is consulted.

2.1.2   Theorema vs Mathematica

Just as we were interested in disambiguating Mathematica and Wolfram Language, to differentiate Theorema (Language) from the former two:

> All Theorema 'reasoners' (provers, solvers, and simplifiers) are written in the programming language of Mathematica. Theorema does not use the Mathematica algorithm library or any implicit mathematical knowledge presupposed in Mathematica algorithms. [22, p. 110]

In current speak, Theorema is implemented in Wolfram Language - but not using in-built algorithms or knowledge, such as the native experimental (in Version 14.0) functions *ProofObject* [42] and *FindEquationalProof* [20].

2.1.3   *Theorema 2.0* - Theorema Commander, Current Project Structure

Theorema 2.0 most prominently introduces an interactive graphical user interface (GUI) to realize a full fledged mathematical assistant system, depicted in Figures **??** (splash screen at application start), 1.1 (start screen under Windows) and 2.1 (again the Theorema Commander, this time under Linux, taken from [57]), profiting from GUI-friendly dynamic expressions [28, p. 76] and cascading stylesheets [45] both introduced with Mathematica Version 6 to realize a native implementation. The mode of interacting with the system fundamentally changes to be more newcomer friendly, because less relient on prior knowledge of the language:

> As an example, giving a definition meant evaluation of a `Definition[...]`-command, stating a theorem meant evaluation of a `Theorem[...]`-command, proving a theorem meant evaluation of a `Prove[...]`-command, and performing a computation meant evaluation of a `Compute[...]`-command. For the new Theorema 2.0 system, we envisage a more 'point-and-click'-like interface as one is used to from modern software tools like a mail user agent or office software. [57, p. 73]

The target user groups are mathematicians and students of mathematics [57, p. 73] Since the Theorema provers are composed of smaller "special prover models" that can be recombined: "In the current status, the access to special prover modules is restricted to the system developers, but a mechanism for users to compose their own provers from available special prover modules is planned for future versions of the system." [22, p. 111]
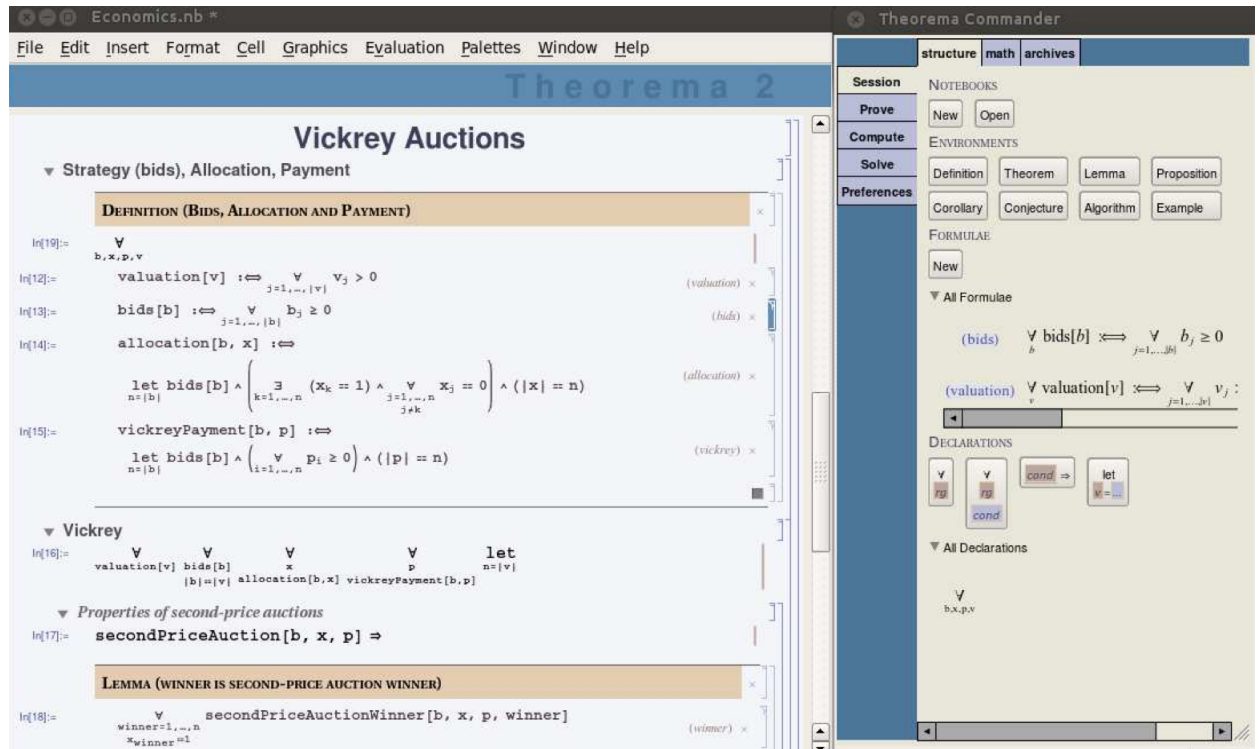
Figure 2.1: Taken from [57], this side-by-side view of a notebook with the Commander also shows off the virtual keyboard, including "keys" for formula elements: comparing to the current version of the Commander in 1.1, some changes have been made to the interface options, e.g. "Session" appears to be "Prepare"

2.1.4   Theorema Environment and Surfacing the Theorema Language: Compound Expression as Datastructure

Theorema data is read from Mathematica notebooks [57, p. 75], but exists beyond the data directly visible in the front-end (this needs to be loaded at the appropriate time ahead of the LaTeX-transformation, so that this authoritative form of the formula is the source of the transformation) - for display purposes, Theorema also defines specific stylesheets, changing the visual appearance from standard Mathematica notebooks, visible in Figure 2.1.

> As soon as the [Theorema] formula is passed to the system through Mathematica's standard Shift-Enter, the formula is stored in an internal datastructure that carries a unique key for each formula in addition to the formula itself and its label. This key consists of the absolute pathname of the notebook file in which it was given, and the unique cell-ID within that notebook, which is provided by the Mathematica front-end. [57, p. 75]

The Theorema formula expression starts like `{Theorema`Common`FML$[{"ID:169304498"`, ... where we actually looking at a list of (denoted by curly braces) containing multiple `Theorema`Common`FML$` expressions, each in turn containing a list, and then some more data,

but inside the list the first element is the unique cell-ID that the front-end provided. But, "the user never sees nor needs the concrete formula key explicitly." [57, p. 75]

There is a hidden complication at this connection between front-end and internal datastructure: To capture the idea of a scope to make definitions in, Theorema allows for *global declarations*, 'which may either contain one or several "orphaned" universal quantifiers (each containing a variable and an optional condition, but missing the formula, to which thery refer) or an "orphaned" implication (missing the right hand side), or an abbreviation indicated by a "let."' [57] names this biimplication:

$$\text{bids}[b] :\Longleftrightarrow \forall_{j=1,\cdots,|b|}\, b_j \geq 0 [57, p.76]$$

This actually translates to:

$$\forall_b bids[b] :\Longleftrightarrow \forall_{j=1,\cdots,|b|}\, b_j \geq 0 [57, p.76]$$

To make the idea specific to WL, an example from this project's main test notebook, FirstTour.nb, tracing such a declaration from its display in the front-end 2.2, to its notebook cell structure 2.1, and finally, to its Theorema formula correlate 2.2, helps clarify this aspect of Theorema, relevant to the current goal of rendering an accurate output in LaTeX: The question for the implementation will be how to obtain the Theorema-formula per relevant notebook cell and decide about a full length output (with global declarations) or a somehow trimmed version that is closer to the localized definition: further, it will likely be this hidden Theorema-representation we want to make visible by outputting the output document, rather than the purely-Wolfram-Language representation of the cell (structure) that holds the code for the display of the formula.
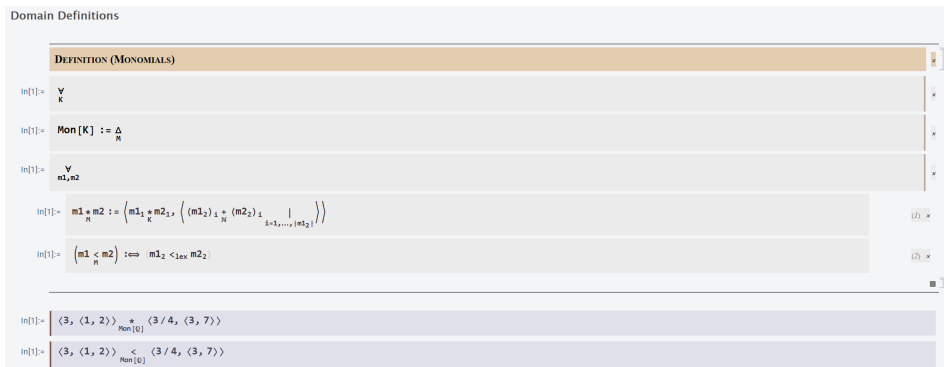
Figure 2.2: A global declaration in Theorema with three different terms.

2.2   Large Systems with Wolfram Language

Wolfram Research advocates for building large systems in WL and cites the WL system itself as "one of the more complex software systems ever constructed. It is built from several million lines of source code, written in C/C++, Java, and the Wolfram Language." [50, The Software Engineering of the Wolfram System] Wolfram Research cites the following general principles and more as they concern building large systems in any language [8]:

Program 2.1: This is an excerpt from the notebook cell expression depicted in the front-end rendering in Figure 2.2 and contains just one such global declaration, `UnderscriptBox ["[ForAll]", "K"]]` as the pertinent line from this cell structure.

```
 1 ...
 2 Cell[BoxData[
 3  UnderscriptBox["\[ForAll]", "K"]], "GlobalDeclaration",
 4  CellFrameLabels->{{None,
 5     Cell[
 6      BoxData[
 7       ButtonBox[
 8       "\[Times]", Evaluator -> Automatic, Appearance -> None, ButtonFunction :>
 9        Theorema`Language`Session`Private`removeGlobal[{
10         "C:\\Users\\jackh\\OneDrive\\Documents\\RISC2023\\prototype-wolfram-\
11 lang\\FirstTour.nb", 2090454223}]]]]}, {None, None}},
12  ShowCellTags->False,
13  CellChangeTimes->{{3.62218621587292*^9, 3.622186218593838*^9}},
14  EmphasizeSyntaxErrors->True,
15  CellTags->"ID:2090454223",
16  CellLabel->"In[1]:=",
17  CellID->2090454223,ExpressionUUID->"809f4e1b-f26c-4265-a5d0-d66f43b5b903"]
18 ...
```

Program 2.2: This is another Theorema formula and contains as part of it the global declaration term encoded in the notebook expression in 2.1.

```
 1 Theorema`Common`FML$[{"ID:2008910260",
 2   "Source:C:\\Users\\jackh\\git\\repository\\tma2tex\\FirstTour.nb"},
 3  Theorema`Language`EqualDef$TM[
 4   Theorema`Language`DomainOperation$TM[Theorema`Knowledge`M$TM,
 5     Theorema`Language`Times$TM][Theorema`Knowledge`m1$TM,
 6    Theorema`Knowledge`m2$TM],
 7   Theorema`Language`Tuple$TM[
 8    Theorema`Language`DomainOperation$TM[Theorema`Language`K$TM,
 9      Theorema`Language`Times$TM][
10     Theorema`Language`Subscript$TM[Theorema`Knowledge`m1$TM, 1],
11     Theorema`Language`Subscript$TM[Theorema`Knowledge`m2$TM, 1]],
12    Theorema`Language`TupleOf$TM[
13     Theorema`Language`RNG$[
14      Theorema`Language`STEPRNG$[
15       Theorema`Language`VAR$[Theorema`Knowledge`VAR$i$TM], 1,
16       Theorema`Language`BracketingBar$TM[
17        Theorema`Language`Subscript$TM[Theorema`Knowledge`m1$TM, 2]],
18       1]], True,
19     Theorema`Language`DomainOperation$TM[
20       Theorema`Language`IntegerInterval$TM[1, \[Infinity], True,
21        False], Theorema`Language`Plus$TM][
22      Theorema`Language`Subscript$TM[
23       Theorema`Language`Subscript$TM[Theorema`Knowledge`m1$TM, 2],
24       Theorema`Language`VAR$[Theorema`Knowledge`VAR$i$TM]],
25      Theorema`Language`Subscript$TM[
26       Theorema`Language`Subscript$TM[Theorema`Knowledge`m2$TM, 2],
27       Theorema`Language`VAR$[Theorema`Knowledge`VAR$i$TM]]]]]], "1"]
```

- Divide the System into Components
- Write and Use Unit Tests
- Think of the Architecture, Not the Code
- Use Source Control
- Write Documentation

There are also WL-specific advantages in Software Engineering, explored in [8, Take Advantage of the Wolfram Language].

2.2.1 Modularity with Packages

In WL, Package development [39], namespace management [36], and further scoping constructs [43] are interrelated and form the extensibility of the system: A typical Wolfram Language package is a .wl or .m file that contains a collection of functions and variables. These packages are structured in a way that separates the implementation from the interface, often using `BeginPackage[]` [15] and `EndPackage[]` [15] to define the public interface and `Begin[]` [3] and `End[]` [14] for the implementation section.

Contexts are used to manage namespaces [36] , preventing name collisions between different packages or within different parts of the same package. By convention, package names serve as contexts, which helps in organizing the functions and variables and avoiding naming conflicts. Packages in the Wolfram Language use `Get[]` (`<<`) [23] for loading, which executes the package code, effectively defining the functions and variables in the specified context. (`Needs[]` is the alternative, only loading loading the package if the specified context is not already in `$Packages`, the relevant context variable in this case. [37])

2.2.2 Theorema as an Extensible Mathematica Package

Theorema can be loaded like any WL-package but is really a collection of Wolfram Language packages, so the proposed format of the functionality implemented with this project is also a WL-package, a file with file ending ".wl" (current) or ".m" (historically) and following the layout given by the Theorema template file ("Theorema /PackageTemplate.m" [52]) - including the copyright statement, which bakes the GNU (a recursive acronym, "GNU's Not Unix" [49]) licensing in right at the point of extensibility and states that anyone is free to redistribute and/or modify the software under the terms of the GPL (General Public License). [48] The software is then provided without any warranty.

2.3 Paradigms: The Project-Perspective on the Multi-Paradigm Approach in Wolfram Language

WL supports multiple programming paradigms, including the procedural one: "The Wolfram Language supports all standard procedural programming constructs, but often extends them through integration into its more general symbolic programming environment." [41] While not a pure OOP (Object Oriented Programming) language, WL can

mimic OOP concepts through associative arrays (Associations) and symbolic structures, as outlined in Section 1.1.3.

2.3.1 Functional vs Procedural Programming in Wolfram Language, and the High-Level Programming Paradigm

Functional programming in WL is introduced in the Wolfram Research documentation center as well [21], and exploring this in the context of high level programming is particularly fruitful, since the combination of abstracted functionality with functional approaches can make for readable code. In a WL context, this is typically the preferred programming approach, combining well with the Wolfram Research dedication to expanding the language with ever more encompassing abstractions in the form of new function definitions, which can be termed the high-level programming: the language itself is not so concerned with code flow commands and data structures as it is with representing expressions and operating on them directly, see appendix D for the company's take on this as symbolic programming.

2.3.2 Symbolic Expressions Lead to Rule-Based Programming and Pattern Matching Approaches

For the work at hand, the representation to find is that of logical connectives (1), predicates (2), and quantifiers (3), so all the elements of First-Order Predicate Logic (FOPL)—implemented in Wolfram Language (WL) and similarly in Theorema—are as follows:

1. **Logical Connective:** `And[p, q]` $\rightarrow p \land q$
2. **Predicate:** `EvenQ[n]` $\rightarrow$ "n is even"
3. **Quantifier:** `ForAll[x, x > 0, x^2 > 0]` $\rightarrow \forall x > 0, \ x^2 > 0$

Beyond what constitutes "symbolic" for Wolfram Research today (a data-centric approach ultimately, see Appendix D), expressions compounded of these elements implement the Theorema language and make it operable within the programming paradigms relevant to the language, discussed in the remainder of this chapter.

2.3.3 Rule-Based Programming

The ρLog package is an example of a package extending WL's rule-based programming capacity [32] - much like Buchberger, the authors identify Mathematica as a powerful system implementing the paradigm they are interested in, rule-based programming, the more general one as compared to (term) rewriting: While both concepts rely on rules, term rewriting is a specific type of rule-based operation focused on the transformation of expressions, whereas rule-based programming is a broader paradigm that can dictate various aspects of a program's behavior based on predefined logical rules.

This project's goals make the rule-based programming paradigm clearer. The author's outline the features lacking in Mathematica in terms of rule-based programming:

1. The possibility to program compositions of reductions, alternative choices, reflexive-transitive closures, etc.

2. A built-in search mechanism to decide the existence of derivations $\text{Expr}_1 \to_{rr} \text{Expr}_2$, where Expr_1, Expr_2 are given expression schemata (patterns), and $\to_{rr}$ is a specification of a sequence of rule reduction steps. Typically, the specification of $\to_{rr}$ is built with the operators mentioned before (composition, choices, etc.)

3. The possibility to generate proofs which justify the existence or non-existence of such reduction derivations.

[32]

To address these shortcomings, they simply implement a package to extend Mathematica according to their needs, achieving the following and demonstrating the flexibility of the system in this way.

1. Concise means to express the basic computational steps of an intended rule application as basic rules. These features are inherited from Mathematica, whose computational engine is based on a higher-order rewrite logic and with advanced support for symbolic and numeric computing.

2. Programming constructs, such as conditional basic rules and rule combinators, which make it possible to structure large specifications of rules.

3. Built-in search mechanism to answer queries of the form $\exists\{R_1, ..., R_k\}\text{Expr}_1 \to_r \text{Expr}_2$ where Expr_1 is a ground expression, r is the identifier of a rule, and Expr_2 is a Mathematica pattern whose variables are named $R_1, \ldots, R_k$ (see Section 2.2) [of [32]].

4. Support for generating proof objects, i.e., certificates that justify the correctness of the answer provided by ρLog to a query.

5. Visualization tools for proof objects, which enable the analysis of the deduction derivations of ρLog in a natural language formulation and at various levels of detail.

[32]

In both contexts, symbolic expressions are not just passive data; they are actively transformed or evaluated as part of the computational process. These expressions provide a versatile and powerful means to represent and manipulate knowledge, logic, and computations in a way that is abstracted from the specific details of the underlying data, allowing for more generalized and flexible rule application and system behavior.

2.3.4 Rule-Based Programming vs Rewriting, via Pattern Matching

Rule-based programming focuses on defining rules that guide the transformation of expressions within a program. It is declarative, meaning it specifies *what* should be done rather than *how* to do it. Rules are applied to expressions iteratively until no further rules can be applied or until a certain condition is met.

Pattern matching is a technique often used within rule-based programming but is more specifically focused on identifying parts of an expression that match a certain form. Pattern matching is integral to the process of identifying where and how rules

should be applied in rule-based programming. Not all pattern matching is necessarily linked to rule-based programming.

As seein in the introduction 1, Rewriting, while similar to rule-based programming, is a specific subset focused on the transformation of expressions through specific rules. It operates under the paradigm of applying these rules to achieve a specific form or outcome. Rewriting is a more specialized operation compared to the broader scope of rule-based programming.

Key Differences

- **Scope:** Rule-based programming can encompass various aspects of program behavior dictated by rules, whereas rewriting is a particular technique within this broader paradigm.
- **Focus:** Rule-based programming is concerned with defining transformations (the *what*), while pattern matching focuses on identifying parts of an expression that need transformation (the *identification*).
- **Application:** Pattern matching serves the rule-based programming paradigm by identifying where rules apply, while rule-based programming defines what transformations to apply.

Chapter 3

Concept: Package Design, Theorema Integration, TeXForm Consideration – In Any Case Pattern-Matching-Based Expression Processing

Having already taken a broad conceptual introductory approach and weighed the relevant theory, it should be efficient to derive the concept for a program package dealing with transformation of the notebook format to LaTeX at this point: The aim is to apply the strengths of WL as a programming language as directly as possible.

3.1 Conceptual Cornerstones for this Project

3.1.1 WL-Native Approach for Direct Integration with Theorema

The design for the package is such that it is intended to be integrated with the Theorema project as a stand-alone package. Figure 3.1 illustrates the relationship of this package to Theorema (Language) and Wolfram Language, as a whole.

The package itself is designed to follow the logic illustrated in Figure 3.2: files involved are colored green, code blue, transformation logic red - these are all the subject of this chapter.

This setup lends itself to an approach where the LaTeX-template is used for customizations, at output: the template is filled by the LaTeX generated by transforming the relevant Theorema Language formulas, particularly, but the Wolfram Language notebook as a whole, generally.

It also turns out the native WL environment, or kernel, has an existing function TeXForm taking a WL expression and transforming it into its typesetting correlate in LaTeX: this was researched as part of the present work, even to the level of the source code, but ultimately discarded as a viable path due to some very central conceptual aspects that will be highlighted in the following. The analysis is contained in appendix E and was ultimately helpful in deriving a specification as well.
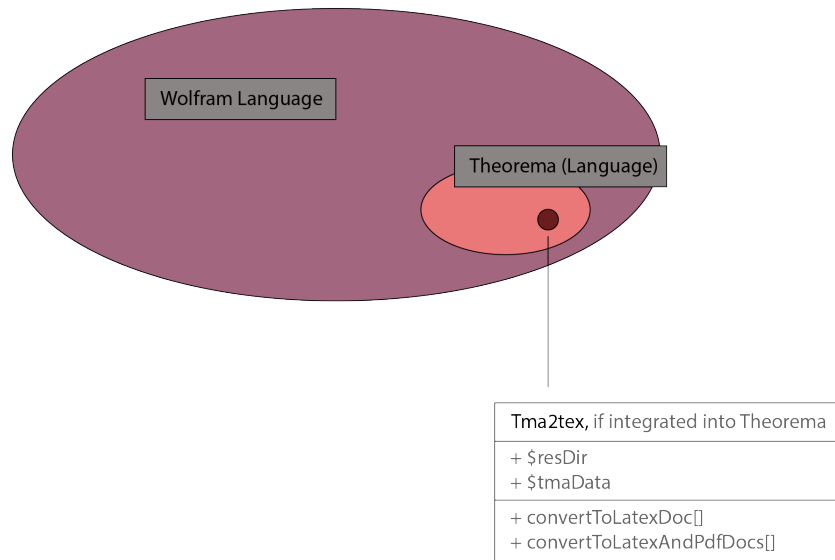
Figure 3.1

3.1.2 Package/MakeTeX-Specification

The following specification was developed in coordination with RISC and developed further as the project matured, especially as concerns extensibility (`MakeTeX[]`):

- **Package Dependencies:**
  - `Theorema` (when not incorporated directly into Theorema, which is not part of this project and subject to user intention/Theorema development plan)
  - If TeXForm-approximation were to be developed further: `Texformdump`

- **Global Variables:**
  - `Tma2tex::$resDir`
    * **Usage:** Defines the directory for LaTeX-templates and any other resources.
    * **Value in project repo:** `C:\Users\jackh\git\repository\tma2tex\res`
  - `Tma2tex::$tmaData`
    * **Usage:** Contains the Theorema-Datastructure that holds formula-expressions and is typically equivalent to `Theorema::Common::$tmaEnv` on the Theorema-side but can be used to show the content according to Tma2Tex (as a separate package).
    * **Value:** Subject to notebook loaded, e.g. FirstTour.nb (gives Theorema formula expressions, as a list, in Theorema language
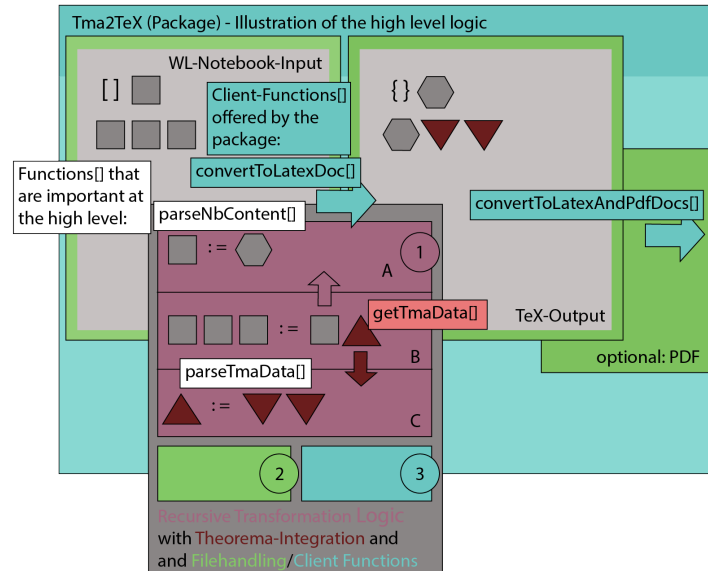
Figure 3.2

- **Client-Functions:**
  - `convertToLatexDoc`
    * **Usage: `convertToLatexDoc[notebookPath]`** transforms a given WL notebook (by file path) to TeX output, creating a new TeX file from a specified resource template.
  - `convertToLatexAndPdfDocs`
    * **Usage: `convertToLatexAndPdfDocs[notebookPath]`** transforms a given WL notebook (by file path) to PDF file as final output, with TeX file as intermediary step, from a specified resource template.
  - `convertToLatexFromString`
    * **Usage: `convertToLatexFromString[nbContentString_, resourceDir_Optional: Tma2tex::$resDir]`** is experimental and intended to be called from the Cloud, simply transforming Wolfram Language String Input to TeX Output (returned directly, not via file). Also uses a template, the resource for which can be passed as the second argument.
- **Lower Level Functions**, provided to the user in the case of TeXForm-approximation - this alternative approach, hinging on a MakeBoxes, a function to create a boxes-representation as an intermediary step, is discussed in section 3.1.4. This approach would benefit from the following specification.

- – `boxesToTeX[boxes_]`, to transform a cell-level boxed expression
- – `expressionToTeX[expr_]`, to transform cell-level expressions (uses Make-Boxes [64] in the background, in Texformdump-package, to then pass to Texformdump'boxesToTeX in the end)
- – `makeTex[boxes_]`, for adding custom box-level transformation rules
- – `makeTex[string_?isTmaCommand]`, for adding custom Theorema-command-level typesetting instructions, in the form of a
  `stringTM`-named LATEXcommand in tmaTemplate, e.g. `\newcommand\ForallTM[2]`
  `\forall_{1} 2` where `string` takes on the value `Forall`.

These would essentially form the head of the package - see the overall notes on package design to conclude this chapter - and all become available to the user, where it is anticipated that mostly the higher level functions, convertToLatexDoc and convertTo-LatexAndPdfDocs, will be of interest, and lowever level MakeTex only secondarily for specific cell transformations as needed, or for customizations of the transformation, and especially for users well-aquaited with the package.

3.1.3 For This Project: No Layout-Information in the LATEX

Having considered the native TeXForm implementation and its reliance on native Make-Boxes, the approach taken here can now be appreciated in its difference: the core idea is to not include layout information of the type that is provided by MakeBoxes in generating the LATEX, but to work directly with the Theorema-Formula and give the user complete control over layout via LATEXand specifically the macros defined in-template.

A comparison will best illustrate the difference, first considering a Theorema formula and the desired output, and then a native WL expression, the relevant MakeBoxes-intermediate step, and the final TeXForm generated LATEX-output.

A Theorema formula:

```
Theorema`Language`Iff$TM[
  Theorema`Language`And$TM[
   Theorema`Language`Forall$TM[
    Theorema`Language`RNG$[
     Theorema`Language`SIMPRNG$[
      Theorema`Language`VAR$[Theorema`Knowledge`VAR$x$TM]]], True,
    Theorema`Language`Or$TM[
     Theorema`Knowledge`P$TM[
      Theorema`Language`VAR$[Theorema`Knowledge`VAR$x$TM]],
     Theorema`Knowledge`Q$TM[
      Theorema`Language`VAR$[Theorema`Knowledge`VAR$x$TM]]]],
   Theorema`Language`Forall$TM[
    Theorema`Language`RNG$[
     Theorema`Language`SIMPRNG$[
      Theorema`Language`VAR$[Theorema`Knowledge`VAR$y$TM]]], True,
    Theorema`Language`Implies$TM[
     Theorema`Knowledge`P$TM[
      Theorema`Language`VAR$[Theorema`Knowledge`VAR$y$TM]],
```

```
     Theorema`Knowledge`Q$TM[
      Theorema`Language`VAR$[Theorema`Knowledge`VAR$y$TM]]]]],
  Theorema`Language`Forall$TM[
   Theorema`Language`RNG$[
    Theorema`Language`SIMPRNG$[
     Theorema`Language`VAR$[Theorema`Knowledge`VAR$x$TM]]], True,
   Theorema`Knowledge`Q$TM[
    Theorema`Language`VAR$[Theorema`Knowledge`VAR$x$TM]]]]]
```

The Theorema-MakeBoxes result:

```
FormBox[
     RowBox[{
        RowBox[{"(",
          RowBox[{
            RowBox[{
              UnderscriptBox["\[ForAll]",
               RowBox[{
                 StyleBox["x", "ExpressionVariable"]}]],
              RowBox[{
                RowBox[{"P", "[",
                  StyleBox["x", "ExpressionVariable"], "]"}], "\[Or]",
                RowBox[{"Q", "[",
                  StyleBox["x", "ExpressionVariable"], "]"}]}]}], "\[And]",
            RowBox[{
              UnderscriptBox["\[ForAll]",
               RowBox[{
                 StyleBox["y", "ExpressionVariable"]}]],
              RowBox[{
                RowBox[{"P", "[",
                  StyleBox["y", "ExpressionVariable"], "]"}], "\[Implies]",
                RowBox[{"Q", "[",
                  StyleBox["y", "ExpressionVariable"], "]"}]}]}]}], ")"}],
       "\[DoubleLeftRightArrow]",
        RowBox[{
          UnderscriptBox["\[ForAll]",
           RowBox[{
             StyleBox["x", "ExpressionVariable"]}]],
          RowBox[{"Q", "[",
            StyleBox["x", "ExpressionVariable"], "]"}]}]}], TheoremaForm]
```

Applying TeXForm (in a modified version available in the file `tma2texV0.wl`) would give:

```
\left(\underset{x}{\forall }P[x]\lor Q[x]\land
\underset{y}{\forall }P[y]\Rightarrow Q[y]\right)\Leftrightarrow
\underset{x}{\forall }Q[x]
```

However, the goal (as defined by the user) actually is:

```
\IffTM{ \AndTM{ \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{x}}}}{ \OrTM{ P[
\VARTM{x}]]}{ Q[ \VARTM{x}]}}}{ \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{y}}}}{
\ImpliesTM{ P[ \VARTM{y}]]}{ Q[\VARTM{y}]}}}}{ \ForallTM{
\RNGTM{ \SIMPRNGTM{ \VARTM{x}}}}{ Q[ \VARTM{x}]}}
```

That is, the formula structure should be translated directly to customizable Theorema-specific commands, and not actually assume any layout information whatsoever, the way TeXForm approaches the problem.

3.1.4   MakeBoxes: An Alternative Typesetting-Pipeline

Both the Theorema and the TeXForm packages provide MakeBoxes-functionality [31], taking care of the Mathematica typesetting: underscripts, delimiters, and the like are typesetting data that is specified in addition to formal and semantic data. This is the reason TeXForm's expressionToTeX actually uses boxesToTeX under the hood, by intercalating a MakeBoxes-step.

For purposes of Theorema it would be paramount to use the MakeBoxes definition supplied in `Theorema`Language`Syntax` applying all the relevant typesetting information at boxes-level, taking into account various Theorema-specific information like Theorema-standard operators, non-standard operator, quantifiers, ranges, and the like.

Such an expression can generally be used for TeXForm parsing, but the parsing mechanism would need to be adapted to allow for customizations, a crucial part of this project and required by the user: registering custom Theorema-LaTeXcommands should be made easy for the user, allowing for custom specification of the command in the template.

Non-standard commands (as specified by TeXForm) would be rendered as text (`\text`), signaling to the user that the command should would need to be registered (on the Mathematica side) and specified (on the LaTeXtemplate side). Conceptually, for this implementation, this mechanism requires a symbol-level adaptation of TeX-Form in the source: `$TeXReplacements` needs to be extended in the case of customizations and MakeTeX implemented to respond to Theorema-expressions dynamically, in terms of the implementation details already discussed. At the level of the specification `makeTex[string_?isTmaCommand]` allows for the required customization.

This style of approach was ultimately not chosen, rejecting layout-processing at WL-level and delegating this to LaTeXentirely, as described. (A draft implementation is available in the project repository as a `V.0` (`tma2texV0.wl` using `texformdump.m`), still, in case of future need.)

3.2   Double Recursive Descent Through Wolfram and Theorema Language Using Pattern Matching and Rule Based Programming

Pattern matching and rule-based programming are core aspects of WL and form the backdrop the approach proposed in this concept outline: While they are closely related and sometimes overlap in their applications, they serve distinct purposes in the language.

The WL engine tries to find occurrences of defined patterns within expressions. When a match is found, various operations, such as replacement, extraction, or modification of the matched part, can be performed.

Rule-based programming involves defining the operative rules that transform expressions in specific ways. It is a declarative programming style focusing on the what rather than the how. Rules are applied to expressions until no more rules are applicable, or a specified condition is met.

To disambiguate pattern matching and rule based programming further, it is helpful to focus on scope: Pattern matching is a technique used within rule-based programming. Rules often use patterns, but not all uses of patterns are in the context of rule-based programming. Pattern matching is about identifying parts of expressions that fit a certain form. Rule-based programming is about defining transformations that should be applied to expressions, potentially utilizing pattern matching to identify the parts to be transformed. Finally, in terms of concern, pattern matching focuses on the "identification" part, while rule-based programming focuses on the "transformation" part.

In modern WL documentation rule-based programming and pattern matching are described as the "core of the Wolfram Language's symbolic programming paradigm [...] for arbitrary symbolic patterns." [**wolfram_research_rules_nodate**]

3.2.1 Pattern Matching to Realize LaTeX-Transformation of Wolfram Language Notebook Code

Fig. 3.3 illustrates the two descents occurring in this project high-level, along with central functions and their overloaded counterparts in the code doing the pattern matching. The result is the LaTeXcode far right on bottom in the graphic, with both document and formula level output.

Section 4.3 in the following chapter elaborates on the specificity rules crucial to the call order. Recursion is simple to see at this point: by adding a call to the same function on the right hand side of the rule, the rule becomes recursive. The core idea of this implementation is the fashioning of many rules that are applied according to the relevant Theorema Notebook pattern, called recursively and descended in the sense that we move from outer expression to inner-most. Indeed, we start from calling the relevant function, parseNotebookContent in the form `parseNotebookContent[Notebook[l_List, ___ ]]`, that is, on the entire Notebook expression that represents a Mathematica notebook.

The same idea is applied twice, to two sets of patterns: once for general text and WL notebook expression syntax, and once for the more specific syntax specified by the Theorema Language. Hence it makes sense to speak of a double-recursion, or a double-descent, in this implementation. Theorema Language is rendered distinctively in the final LaTeX/PDF output, to mark this distinction outwardly.

3.2.2 Pattern Matching to Realize LaTeX-Transformation of the Theorema Language Data Structure

This subsection expands on the notion of the horizontal program dimension, after the live version of the current Theorema formula under consideration (being parsed) has been obtained, via getTmaData[]: now the goal is to generate a TeX-snippet like `\IffTM{ \AndTM{ \ForallTM{ ...` with appropriate closing brackets from a formula

Tma2TeX Recursion Flow

Overview Program Logic   Detail Program Logic with relevant high-level Functions
                          - Arrows indicate Recursion Flow, coloring type of expression

Recursion/Transformation Order
(Top to Bottom)
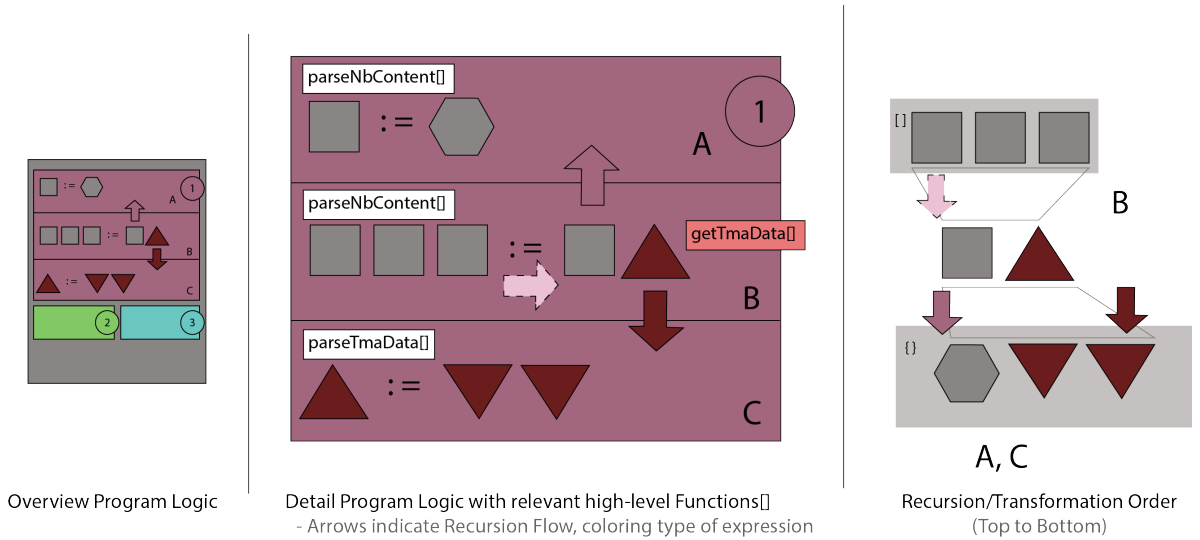
Figure 3.3

like `Theorema`Language`Iff$TM[ Theorema`Language`And$TM[ Theorema`Language`Forall$TM[ ....`
The easy suffix "TM" is chosen for the LATEXoutput, the "$TM" visible in the Theorema
Language code is the original way of keeping separate contexts. In this way Theorema
may specify its own "Iff", "And", "Forall" etc., both in notebooks and the LATEXresult
of this project.

Roughly, there is a Predicate Logic distinction between operator symbols known
to the language, marked by the word "Language" in the symbol name context, and
knowledge outside of the language, functions and predicates with the word "Knowl-
edge" in their context path. [61] The parsing needs to result in square brackets for the
"Knowledge"-symbols, and macro-syntax curly braces for the "Language"-symbols, with
appropriate parameter placement. These macros, or commands, are then fully defined
inside the LATEX-template file.

3.3   Extensibility in Both LATEXand Wolfram Language

3.3.1   A Note on Evaluation Criteria and Stability

The execution of the idea can be evaluated in terms of two core criteria. First, stability
is a key criterion for this work and actually bases on generality of the patterns selected:
WL and Theorema expression patters that will stand the test of time need to be selected
at the development stage, and testing must include a diverse current range of Theorema
notebooks.

3.3.2 WL-Messages and -Tests: Software Design Goals

WL provides modern engineering-oriented functionality for both tests and error-handling, in the form of messages [65] - the mechanism is part of this package such as in the case the file has not been loaded completely to provide the required formula data, see Figure 3.4.


```
In[14]:= << Tma2tex`
         convertToLatexAndPdfDocs[testNotebookPath]

         ••• Tma2tex`Private`tmaDataImport: The Theorema–Formula–Datastructure is empty.
                    Did you evaluate a Theorema notebook before loading the package and calling the
         conversion function?

Out[15]= $Failed
```

Figure 3.4: The basic idea is that every message has a definite name, of the form symbol::tag. [70]

Testing will make up a large part of Chapter 5.

3.3.3 Extensibility

Extensibility at both the WL and LaTeXlevels is the other priority, with the understanding that Theorema Language might change, Wolfram Language might be updated, and additional LaTeXcommands or alternative rendering imperatives might be required.

The main extension points go as follows, where the sectioning of the code follows the final version submitted with this thesis for easier finding of the relevant code. At the WL-level:

- Package Parts 1.B and 1.C, where B covers the first recursive descent through the overall notebook structure and C the second descent into the Theorema-formula:
- formatTmaData (1.C.3): wrapper function around the Formula-LaTeX-output that maybe used for string replacements, for instance, at the level of the formula-LaTeXsnippets.
- Client-Functionality (Part 3): both top level functions convertToLatexDoc and convertToLatexAndPdfDocs may be edited easily to suit the needs of higher-level project, while more rigid configuration details are hidden away in Part 2, especially as concerns filehandling details.
- Overall Package: This project was developed as a package ¨tma2tex¨but the code Parts 1 - 3 can easily be moved to another package, as long as Theorema, the overall package, is in some way available, and the global variables `Tma2tex`\$resDir` and `Tma2tex`\$tmaData` are readied.

The main extension procedure on the level of LaTeXis to add macro-definitions in the template provided in `Tma2tex`\$resDir`, in a file called `tmaTemplate.tex` in the default configuration. A sample set of macros is provided for reference in `sampleTemplateMacros.tex`.

Chapter 4

Implementation, Wolfram Language Programming Paradigms and Guidelines, Integration and Deployment/Cloud

4.1   Overview of the Implementation

The implementation followed the specification introduced in Section 3.1.2, forgoing the approach reliant on TeXForm explored at concept stage, for a simpler, direct translation mechanism, detailed in this chapter. The package specification looks like this at a high level and at the time of completing of the project:

- **Package Dependencies:**
    - `Theorema`
- (Package-public) **Global Variables/User Settings:** The dollar sign is in keeping with the Theorema (and other package) convention for global variables.
    - `$resDir`: expects a file `tmaTemplate.tex`, and any other resources should be added here to.
    - `$tmaData`: set to the value of `Theorema`Common`$tmaEnv`, callable to make background data visible to the user at any time.
- **Client-Functions:** the camelcase (rathern than Pascal-case) is to distinguish the function name from WL-internal function names.
    - `convertToLatexDoc`, input: a Theorema notebook/output: a .tex-file in the current directory/option: DocumentProcessingLevel goes to empty string, None or Full.
    - `convertToLatexAndPdfDocs`: a Theorema notebook/outputs: a .tex-file and a .pdf file in the current directory/option: DocumentProcessingLevel goes to empty string, None or Full.

4.1.1   Note on Modular Programming in Wolfram Language

Related WL encapsulation constructs include `CompoundExpression[]` [12], `Module[]` [35], and `Block[]` [5]], but concretely for this package, in standalone form, the package loads Theorema via a `Needs[]` call, realizing module dependency: if the package were to be integrated to Theorema directly in future work, Theorema functionality would either be available directly or loaded in a more directed fashion in the form of relevant sub-packages.

4.1.2   Overall Structure of the Package

For organization of the code the following hierarchy and division of concerns was followed throughout, to make analysis of this academic project more structured

- **Part 0: Setup** - global variables and the like.
    - `Part 0.A: Public Package Variables`: `Tma2tex`$resDir`, set by the user in the Package Header, and `Tma2tex`$tmaData`, set to `Theorema`Common`$tmaEnv` initially.
    - `Part 0.B: Private Package Variables` like `Tma2tex`$documentProcessingLevel` and `tmaDataAssoc`, concerned with holding further settings and formats of the data.
- **Part 1: Parsing with parseNbContent, getTmaData, parseTmaData**: the main recursive functionality, with the following subdivisions.
    - `Part 1.A: parseNbContent` - in this section, the concern is appropriate presentation of the output of a notebook parsing run.
    - `Part 1.B: parseNbContent at higher level`, concerned with bridging to the Theorema Language.
    - `Part 1.C: getTmaData/parseTmaData`, concerned with processing the Theorema Language.
- **Part 2: Filehandling**
- **Part 3: Client Functions**

4.2   High Level Programming in Practice

High level programming was touched on in Section 2.3.1: In high-level programming within WL, the focus is primarily on manipulating (symbolic, in Wolfram Reseach speak) expressions and applying transformation rules rather than managing low-level implementation details. This abstraction allows leveraging the language's built-in functions for data manipulation, pattern matching, and rule-based transformations as well as several key features like filehandling, reflected in the client functions in this project.

4.2.1   Client Functions

The main client function in this package, relying heavily on high level programming, is `convertToLatexDoc`. This function is responsible for converting a given Theorema

notebook to a LaTeX document. The function takes a notebook path and optional
settings for the document processing level. It retrieves the notebook content, parses it,
and fills a LaTeX template with the extracted data such as the title, author, and date.
The function then exports the filled content to a `.tex` file.

   This function is also called by `convertToLatexAndPdfDocs`, which extends its func-
tionality by converting the LaTeX file to a PDF. After calling `convertToLatexDoc`, it
checks for successful conversion and then uses the `pdflatex` command to compile the
LaTeX file into a PDF document.

   These two functions, `convertToLatexDoc` and `convertToLatexAndPdfDocs`, are the
core client functions intended to be called by Theorema, the user. They encapsulate
the primary operations necessary for converting Theorema notebooks into LaTeX and
PDF documents, providing a seamless integration with the Theorema environment for
document processing.

```
Options[convertToLatexDoc] = {DocumentProcessingLevel -> ""};
convertToLatexDoc[notebookPath_, OptionsPattern[]] := Module[{nb, content,
    latexPath, latexTemplatePath, resourceDir = $resDir, texResult, sownData,
    filledContent, closeFlag = False, documentProcessingLevel},
  If[Length[$tmaData] == 0,
    Message[tmaDataImport::empty, "The Theorema-Formula-Datastructure is empty.
    Did you evaluate a Theorema notebook before loading the package and calling
        the conversion function?"];
    Return[$Failed]
  ];

  documentProcessingLevel = OptionValue[DocumentProcessingLevel];
  If[documentContentProcessingLevel =!= "",
    SetDocumentProcessingLevel[documentProcessingLevel]
  ];

  nb = If[isNotebookOpen[notebookPath],
    NotebookOpen[notebookPath],
    NotebookOpen[notebookPath, Visible->False]; closeFlag = True];

  content = NotebookGet[nb];
  NotebookEvaluate[content];
  latexPath = getLatexPath[notebookPath];
  latexTemplatePath = getLatexTemplatePath[notebookPath];

  {texResult, sownData} = Reap[parseNbContent[content], {"title", "author", "date"}];
  filledContent = fillLatexTemplate[resourceDir,
    <|
      "nbContent" -> texResult,
      "nbTitle" -> First[sownData[[1, 1]]],
      "nbAuthor" -> First[sownData[[2, 1]]],
      "nbDate" -> First[sownData[[3, 1]]]
```

```
      |>];
  Export[latexPath, filledContent, "Text"];
  If[closeFlag === True, NotebookClose[notebookPath]];
]


Options[convertToLatexAndPdfDocs] = {DocumentProcessingLevel -> ""};
convertToLatexAndPdfDocs[notebookPath_, OptionsPattern[]] := Module[{latexPath,
    pdfPath, compileCmd, conversionResult},
  conversionResult = convertToLatexDoc[notebookPath,
    DocumentProcessingLevel->OptionValue[DocumentProcessingLevel]];
  If[conversionResult === $Failed,
    Return[$Failed]
  ];
  latexPath = getLatexPath[notebookPath];
  pdfPath = StringReplace[latexPath, ".tex" -> ".pdf"];
  compileCmd =
   "pdflatex -interaction=nonstopmode -output-directory=" <>
    DirectoryName[latexPath] <> " " <> latexPath;
  RunProcess[{"cmd", "/c", compileCmd}];
]
```

4.2.2   File-handling and LaTeXDetails

Key Functionalities: The primary functionalities of the file-handling code can be summarized as follows:

- **Writing to LaTeX Files:** The function `writeToLatexDoc` is responsible for writing the parsed content of a Theorema notebook to a new LaTeX file. This function opens a file stream using the provided path (`latexPath`), writes the parsed content (`nbContent`) to it, and then closes the stream. This process ensures that the data is efficiently transferred from the internal representation to a format suitable for LaTeX processing.

- **Generating LaTeX File Paths:** The function `getLatexPath` constructs the full path for the new LaTeX file by appending the `.tex` extension to the base name of the original Theorema notebook file. This method ensures that the LaTeX file is stored in the same directory as the source notebook, maintaining a logical and consistent file organization.

- **Locating LaTeX Templates:** The function `getLatexTemplatePath` similarly constructs the path for a LaTeX template file. This template file serves as a wrapper or framework for the main content LaTeX file, ensuring proper formatting and structure in the final output.

- **Filling the LaTeX Template:** The function `fillLatexTemplate` handles the process of importing a predefined LaTeX template (`tmaTemplate.tex`) from the specified directory and dynamically filling it with content from an association (`data`). This template-based approach allows for the flexible customization of the LaTeX document according to different requirements or styles.

Directory Structure and File Organization: The implementation works with a well-defined directory structure where the following components are needed:

- **Notebook Directory:** The primary directory where the original Theorema notebook files are located. The LaTeX files generated by `getLatexPath` and `getLatexTemplatePath` are stored in the same directory, preserving the association between the source notebooks and their corresponding LaTeX outputs.
- **Template Directory:** A subdirectory containing the LaTeX template file (`tmaTemplate.tex`). This template is imported by the `fillLatexTemplate` function and filled with content to create a complete LaTeX document. The template file is assumed to be reusable across multiple notebook conversions, providing a consistent document structure.
- **Resulting Files:** The resulting files from this process include:
  - A LaTeX content file named according to the original notebook (`.tex` extension).
  - The final PDF document generated from the filled LaTeX template, if calling the appropriate client function.

4.3   Implementation of (Double) Recursive Descent with Pattern Matching

4.3.1   General Remarks on Pattern Matching, and Execution Order, in Wolfram Language

There are various nuances when it comes to pattern matching in Wolfram Language. One example is this rendering of a "DisplayFormula," that is, a formula written closely to frontend rendering (box structure first, rather than formula first), without Wolfram or Theorema Language logic in mind.

Notebook WL code is given by the following code snippet, illustrating the front-end-orientation of the code.

```
{Cell[BoxData[
 FormBox[
  RowBox[{
   RowBox[{"(",
    RowBox[{
     UnderscriptBox["\\[ForAll]", "x"],
     RowBox[{"(",
      RowBox[{
       RowBox[{"P", "[", "x", "]"}], "\\[Or]",
       RowBox[{"Q", "[", "x", "]"}]}], ")"}]}], ")"}], "\\[And]",
   ... }]},
  TraditionalForm]], "DisplayFormula", ...
 CellID->2121253064,ExpressionUUID->"384e1c8f-1530-48b6-9503-bea644c47327"],
 ...}
```

The functions BoxData, FormBox, RowBox and UnderscriptBox take care of minimal formatting required for a readable rendering.

These expressions can be parsed recursively with the following WL code to test execution order.

```
parseNbContent[l_List] := If[$documentProcessingLevel == "Full",
    "\\colordiamond{blue}", ""]
parseNbContent[l_List] /; MemberQ[l, _Cell] :=
    StringJoin[If[$documentProcessingLevel == "Full",
        "\\colordiamond{purple}", ""], ToString /@ parseNbContent /@ l]
...
parseNbContent[Cell[BoxData[FormBox[content_,
    TraditionalForm]], "DisplayFormula", ___]] :=
        If[$documentProcessingLevel != "None", StringJoin["\\begin{center}",
            parseNbContent[content], "\\end{center}\n"], ""]
```

While the latter pattern is highly specific, there is only a small difference (involving a condition) between the first two rules, concerned with parsing lists, like the one in the previous example, marked by curly braces.

When multiple rules are applicable to a given expression, WL uses specificity to determine which rule to apply. The specificity rule in pattern matching operates on the principle that more specific patterns are chosen over more general ones when multiple patterns match an expression. Here's how specificity is determined in WL:

- Literal Patterns Over Pattern Objects: A pattern that exactly matches an expression is considered more specific than a pattern involving pattern objects (like `_`, `__`, `___`, or named patterns using `_type`, etc.). For example, a rule for `f[1]` is more specific than a rule for `f[x_]`.

- Constrained Patterns Over Unconstrained: Patterns with conditions (`/;`) or pattern tests are more specific than those without. For example, `f[x_ /; x > 0]` is more specific than `f[x_]`.

- Constrained Patterns Over Unconstrained: Patterns with conditions (`/;`) or pattern tests are more specific than those without. For example, `f[x_ /; x > 0]` is more specific than `f[x_]`.

- Length and Structure: Patterns that match expressions with more specific structure or length are preferred. For example, `f[{x_, y_}]` is more specific than `f[_List]`, and `f[x_, y_]` is more specific than `f[___]`.

- Head Specificity: Patterns specifying a head are more specific than patterns without a head specification. For example, `f[x_Integer]` is more specific than `f[x_]`.

- Order of Definition: When patterns have the same specificity, the rule that was defined first is chosen. This is relevant for user-defined rules where the specificity might appear equal.

- Nested Patterns: In nested patterns, specificity is considered at each level of nesting. A pattern that is more specific at any level of nesting is considered more specific overall.

4.3.2   Limited Approach of Specific Pattern Matching Rules

Part 1.A in the code defines patterns as basic cell structures to extract certain informa-
tion, as in this case:

```
(* -- Part 1.A.1 -- Text Expressions (at the Cell Level):
    Not processed if DocumentProcessingLevel = "None", otherwise yes. *)
parseNbContent[Cell[text_String, "Text", ___]] :=
    If[$documentProcessingLevel != "None", "\\begingroup \\section*{} "
        <> text <> "\\endgroup \n\n", ""]
parseNbContent[Cell[text_String, "Section", ___]] :=
    If[$documentProcessingLevel != "None", "\\section{"
        <> text <> "}\n\n", ""]
```

There are also rules for front-end displayed formulas used in the main text, to display
the main content relevant for the reader:

```
parseNbContent[UnderscriptBox["\[Exists]", cond_]] :=
    If[$documentProcessingLevel != "None",
    "\\underset{" <> parseNbContent[cond] <> "}{\\exists}", ""]
parseNbContent[UnderscriptBox["\[ForAll]", cond_]] :=
    If[$documentProcessingLevel != "None",
    "\\underset{" <> parseNbContent[cond] <> "}{\\forall}", ""]
```

To accomplish coherent recursive pattern matching as it is initiated in the preceeding
example, the patterns need to be defined down to the symbol level.

```
parseNbContent[RowBox[{left_, "\[And]", right_}]] :=
    If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left],
    " \\land ", parseNbContent[right]], ""]
parseNbContent[RowBox[{left_, "\[Or]", right_}]] :=
    If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left],
    " \\lor ", parseNbContent[right]], ""]
parseNbContent[RowBox[{left_, "\[DoubleLeftRightArrow]", right_}]] :=
    If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left],
    " \\Leftrightarrow ", parseNbContent[right]], ""]
```

The approach is naturally limited: foreseeing every possible symbol used in mathe-
matics, even a subset, requires an extensive set of rules, the like is implemented under
the hood in the TeXForm, as is explored in the chapter on the concept: this approach
is only implemented so far, instead making use of `parseNbContent[]` to essentially lo-
cate the major features like titles and sections in the Theorema notebook, and find the
appropriate jumping off point (for parsing) into the Theorema formula.

This happens in Part 1.B: the pattern

```
Cell[CellGroupData[{Cell[headertext_, "EnvironmentHeader", headeroptions___],
                    Cell[formulaboxdata_, "FormalTextInputFormula", options___],
                    furtherNotebookEnvCells___},
                    envOptions___]]
```

is captured and one line in the relevant function call reads:

```
formatTmaData@parseTmaData[getTmaData[cellID]]
```

The `cellID`, extracted from the `options` in the preceding pattern, is used to `getTmaData` from the environment variable: now this is parsed in a second recursive descent (with a slightly different approach), and finally formatted as needed on string- rather than expression-level.

4.3.3 Generalized Parsing Approach for Theorema Data

Core Parsing Function: The function `parseTmaData` uses pattern matching to handle various types of expressions. The first definition,

```
parseTmaData[op\_?isTmaLanguageSymbol[args\_\_\_]] :=
  Module[{nextOp, argList, parsedArgs},
  nextOp = prepareSymbolName[op];
  argList = {args};
  parsedArgs = Switch[
    Length[argList],
    1, "{" <> parseTmaData[argList[[1]]] <> "}",
    2, "{" <> parseTmaData[argList[[1]]] <> "}
       {" <> parseTmaData[argList[[2]]] <> "}",
    3, "{" <> parseTmaData[argList[[1]]] <> "}
       {" <> parseTmaData[argList[[3]]] <> "}",
    _, ""
  ];
  " \\" <> ToString[nextOp] <> parsedArgs
  ]
```

handles expressions where the operator is recognized as a language symbol using the predicate `isTmaLanguageSymbol`. The function utilizes a `Module` to locally define variables and processes the arguments using a `Switch` statement, which handles different numbers of arguments. The main thing here: this covers the Theorema Language case of an epxression that needs to be converted to something like

```
\RNGTM{ \SIMPRNGTM{ \VARTM{a}}}
```

for example: The ranges and variable expressions are transformed to parameterizable LaTeXmacros that can be defined exactly the way the user wishes, LaTeX-side: it is a generalized approach.

Alternative Parsing Cases: Several alternative cases are defined to handle different types of expressions:
- **Non-Language Operators:** The second alternative matches expressions where the operator is not a recognized language symbol:

```
parseTmaData[op\_[args\_\_\_]] :=
  Module[{nextOp, argList, parsedArgs},
  nextOp = prepareSymbolName[op];
  argList = {args};
  parsedArgs = Switch[
    Length[argList],
    1, "[" <> parseTmaData[argList[[1]]] <> "]",
    2, "[" <> parseTmaData[argList[[1]]] <> ", "
        <> parseTmaData[argList[[2]]] <> "]",
    _, ""
  ];
  " " <> ToString[nextOp] <> parsedArgs
  ]
```

This alternative similarly handles the parsing, but formats the output using square brackets instead of curly braces, reflecting a different LaTeX macro style reserved for predicate expressions like `P[x]`, for example. This is considered Theorema Knowledge and marked as such in the typical context paths found in Theorema language expressions.

- **Special Two-Argument Sets:** Another special case handles expressions where two sets of arguments are present:

```
parseTmaData[op\_?isTmaLanguageSymbol[args\_\_\_][args2\_\_\_]] :=
  Module[{nextOp, argList, argList2, parsedArgs, parsedArgs2},
  nextOp = prepareSymbolName[op];
  argList = {args};
  parsedArgs = Switch[...];
  argList2 = {args2};
  parsedArgs2 = Switch[...];
    " \\" <> ToString[nextOp] <> parsedArgs <> parsedArgs2
  ]
```

Here, the function handles two levels of arguments, which are independently parsed and concatenated.

- **Literal Numbers and Terminal Expressions:** Additional cases handle literal integers and terminal expressions that do not require further parsing:

```
parseTmaData[i\_Integer] := ToString[i]
```

```
parseTmaData[ax\_] := prepareSymbolName[ax]
```

These cases ensure that numbers and final symbolic expressions are converted directly to their string representations or appropriately prepared names.

Auxiliary Functions: Two auxiliary functions, `isTmaLanguageSymbol` and `prepareSymbolName`, are defined to assist with parsing:

```
isTmaLanguageSymbol[f\_Symbol] :=
    With[{n = SymbolName[f], c = Context[f]}, c === "Theorema`Language`"]
isTmaLanguageSymbol[f\_] := False
```

The function `isTmaLanguageSymbol` determines whether a given symbol belongs to the `Theorema`Language`` context.

```
prepareSymbolName[op\_Symbol] :=
    With[{n = SymbolName[op]},
    If[StringTake[n, -3] == "$TM",
        If[StringTake[n, 4] == "VAR$", StringDrop[StringDrop[n, 4], -3],
            If[isTmaLanguageSymbol[op], StringDrop[n, -3] <> "TM", StringDrop[n, -3]]],
        If[StringTake[n, -1] == "$", StringDrop[n, -1] <> "TM", n <> "TM"]]
    ]
```

The `prepareSymbolName` function is responsible for converting Theorema symbols into valid LaTeX macro names, ensuring that symbols conform to the required formatting conventions for their respective contexts.

Chapter 5

Testing, Analysis and Bench-marking; Closing Remarks, Potential Future Work

5.1 Messages, Failures, and Testing in WL

If WL is to a be a systems engineering language, it makes sense for error cases, failures, to be readily machine readable and conceived of as input for processing in an automated manner, rather than messages that direct a human user of Mathematica Desktop. This shift is apparent in the recommendation for usage of Failure[] rather than the symbol, `$Failed`, simply "a special symbol returned by certain functions when they cannot do what they were asked to do." [1]

Opposite this, Failure[], introduces more functionality and information, especially the failure type: "Failure[*tag*,*assoc*] represents a failure of a type indicated by *tag*, with details given by the association assoc." [18] This introduces the potential for an abstract way to handle a failure based on type, programmatically, rather than a specific directive.

A basic example of a Failure object is `Failure["InvalidInput", <||>]`, without an association even. Inside the Mathematica frontend it is rendered as in figure 5.1.

Figure 5.1: A simple Failure object rendered in Mathematica [18]

A more complicated example in terms of metadata and parameterized messaging is generated by the WL code `Failure["ExternalOperation", <\|`"MessageTemplate" -> "External operation '1' failed.", "MessageParameters" -> "file upload"‖ >]` and rendered as in figure 5.2.

Failure can also contain indexable metadata that might be helpful for programmatic retrieval in the failure case. [18] In any case, Failure objects can be returned by functions to indicate that an operation did not succeed as expected. This approach allows for more structured error handling, where the calling code can inspect the Failure object to determine the cause of the error and decide how to proceed: They are particularly
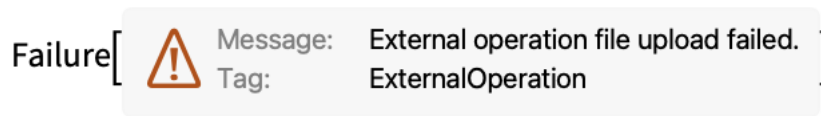
Figure 5.2: A Failure object using a template with positional parameters [18]

useful in functional and symbolic programming patterns prevalent in WL, enabling a more nuanced handling of errors compared to traditional imperative error handling mechanisms like throwing exceptions.

Similarly, Message [33] is about signaling that something unusual has occurred during the evaluation of an expression - where Failure is a way to encapsulatively represent the occurrence of an error and carry forward information about that error in a structured form, a function might generate a Message to inform the user of an error and then return a Failure object to programmatically indicate the error condition to the calling code.

When a Message is generated, it does not by itself halt execution; the computation continues unless the message is associated with an error severe enough to trigger a termination of the evaluation. [33] Messages can be controlled and manipulated using functions like Off, On, and Quiet, allowing to suppress or enable specific messages. This is useful for managing the verbosity of output during computation, especially in cases where certain warnings or errors are expected and do not necessarily indicate a critical failure.

5.1.1   Working with Messages in WL

In WL, messages provide a mechanism for displaying errors, warnings, or other informational text to the user. A message definition typically consists of three parts:

```
CloudConnect::creds = "Incorrect username or password.";
```

The components of a message definition are:

- **Message Head:** The head of a message, such as CloudConnect in the example, represents the function or symbol associated with the message. Some messages are tied to a specific function, while others can be issued by multiple functions. For messages that are generally applicable, the head is set to `General`. For example:

  ```
  General::settf = "Cannot set `1` to `2`; value must be True or False."
  ```

  When issuing a message with a `General` head, the specific function symbol that triggers the message is still used:

  ```
  ...code...
  Message[CloudExpression::settf, dest, expr];
  ...code...
  ```

- **Message Tag:** The message tag is a short, lowercase string that often uses abbreviations. Unlike most naming conventions in the Wolfram Language, message

tags do not use camel case or full words. The tag typically alludes to the expected input or operation that was not received or performed correctly.

- **Message Template:** The template is a full sentence that provides the text to be displayed when the message is issued. It can include template slots, which are placeholders (such as '1' or '2') that will be replaced by data supplied to the Message function at runtime.

By using these components, WL allows for standardized, flexible, and informative messaging that enhances user interaction and debugging capabilities.

Usage messages, or message templates, are a way to document the purpose and usage of symbols (typically functions) directly within the WL environment. These messages provide brief descriptions of what a function does, its arguments, and sometimes examples of how to use it. Usage messages are helpful for both package developers and users, as they offer immediate, inline documentation accessible through the WL interface. They are also used in the present package, see near the top of the source code in Appendix A.

Testing in WL involves evaluation strategies that check for Messages, or Failures, in the modern case. To this end, the language offers a suite of testing functionality: for example, VerificationTest fails by default if any Message is issued (unless it is told to expect a message, e.g. of a certain type). [73] More recently, TestCreate ([67]) and TestEvaluate ([68]) were introduced to the language, to create and run TestObjects, ([47]) respectively.

5.1.2   Testing in the Wolfram Language

WL provides various built-in functions and tools to facilitate different types of testing, including unit tests, integration tests, and performance tests.

Key Features of Testing in WL:
- **Unit Testing:** WL supports unit testing through the VerificationTest function, which allows developers to specify expected outcomes for individual functions or code blocks. This function compares the actual output against the expected result, making it easy to identify discrepancies.
- **Test Files and Test Framework:** The Wolfram Language has a dedicated testing framework that supports the creation of test files. Test files can contain multiple VerificationTest expressions and are typically stored with a `.wlt` extension. These test files can be run individually or as part of a suite using the RunTest function.
- **Automatic Test Evaluation:** WL provides tools for automated test evaluation and reporting. TestReport [69] generates detailed test results, including information on passed, failed, and errored tests. This feature allows developers to quickly assess the overall health of the codebase.
- **Performance Testing:** The Timing and AbsoluteTimingfunctions are used to measure the performance of code in WL. In addition, BenchmarkReport can provide detailed performance analysis, comparing the execution time of different functions or code segments. [72]

- **Continuous Integration and Testing:** WL integrates well with continuous integration (CI) systems, allowing for automated testing in a CI pipeline - using native WL functionality and, for example, WolframScript [75] for commandline support - though so-called Testing Notebooks provide an interface for writing and running tests as well [73]

The Wolfram Language's testing framework is designed to provide a robust and flexible environment for developers to ensure the correctness and performance of their code, making it suitable for both small scripts and large-scale projects.

5.1.3   Testing Approach for this Project

For this project, a structured and multi-layered testing approach has been adopted to ensure the reliability and correctness of the code base, which primarily involves converting Theorema notebooks into LaTeX and PDF documents.

By Testing Category

The focus from the start was unit testing, however - tests were added towards the end of the project due to an evolving specification (and the work of unearthing the details and the important questions to ask) rather than taking a test-driven approach.

Unit Tests: The core functionalities, such as parsing Theorema notebook content, generating LaTeXfiles, and handling file operations, are tested in an outside view (irrespective of LaTeXcode quality). These tests leverage the VerificationTest function to confirm that individual components behave as expected. For instance, the conversion functions like convertToLatexDoc and convertToLatexAndPdfDocs will be tested with various input scenarios to validate their correctness.

The second step is a straightforward unit testing block verifying the expected LaTeXcode of the core parseTmaData function, with a focus on correct rendering of both Theorema Language and Theorema Knowledge.

Integration Tests: Integration tests are not performed as part of this project: these would otherwise ensure that the different modules interact correctly. This includes verifying that the file-handling functions correctly create and manipulate LaTeX files, and that the templates are properly filled with data from the Theorema notebooks. Integration testing would also cover the entire conversion workflow, from input notebook to final PDF output: while some coverage through unit testing is provided, these kinds of tests would be left to the Theorema-integration stage.

Performance Tests: Performance testing is also not performed as part of this project, due to lack of relevance (time is not considered critical, within rational boundaries). Performance tests would otherwise evaluate the efficiency of the conversion process, particularly for large and complex Theorema notebooks, for instane. Functions like parseTmaData and file generation methods would be profiled using Timing and AbsoluteTiming to identify any potential bottlenecks and ensure optimal performance. [73]

Regression Tests: Regression tests will be included to verify that new code changes do not introduce any bugs or negatively affect existing functionality. This will involve re-running all unit and integration tests after any significant code modifications.

Continuous Testing and Automation: No continuous integration pipeline or automatic test triggers were created, relying on Mathematica UI solutions, such as Testing Notebooks, for manual testing instead.

Unknown Pattern Failures

This failure was initially anticipated for the following case: The central parseTmaData[] recursive function was called with an expression that did not fit any of the expected patterns, falling into the most generic case:

```
parseNotebookContent[other_] := StringJoin["\\textcolor{red}{", "Pattern not found! ", T
```

The manually verifiable output follows the screenshot presented in Image 5.3. Also visible is the run-on output of the string representation of the unmatched pattern printed to the underlying LaTeX. (The grayed out "Cell reached" and similar directives are used for cross-referencing navigation in the output document back with the origin-document in development.) This approach was finally abandoned in favor of the previously described unit tests comparing the input with a specified output, rather than hinging testing functionality upon the pattern matching process itself.

In the final versions, errors and failures are not actually passed to the output document, and error states hindering evaluation of the main functions simply trigger messages to the user, as described in the opening of this chapter. The generalized approach chosen for parseTmaData and described in the previous chapter relies on the unchaning Theorema Language specification.
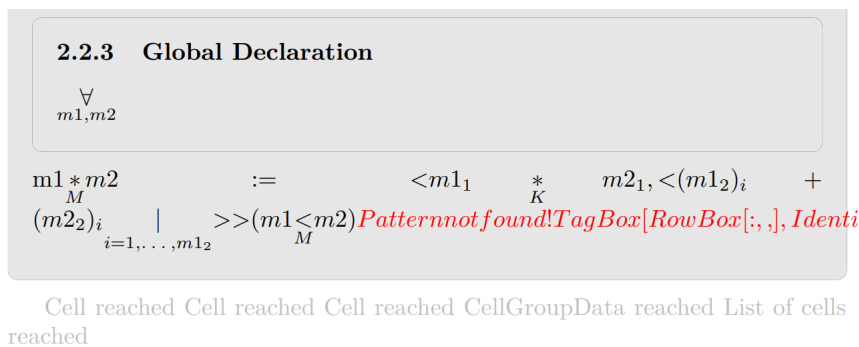
Figure 5.3: parseNotebookContent[other_] output in document with LaTeXcommand formatting

5.2   Analysis and Review

The goal of the project was to develop a robust mechanism for transforming Wolfram Language/Theorema notebooks into LaTeXdocuments: The transformation logic of the

final implementation relied heavily on pattern matching and rule-based programming
techniques to accurately map Theorema language constructs to their LATEXcounterparts.
This approach was essential for preserving both the mathematical expressions and struc-
tural elements inherent in Theorema notebooks. The solution is designed to integrate
seamlessly with the existing Theorema system, utilizing both existing data structures
and evaluation mechanisms to achieve a coherent mechanism for translation.

The modular design of the package is a key strength, allowing for straightforward
future extensions and modifications. This modularity ensures that additional features
or support for more complex Theorema constructs can be incorporated with minimal
changes to the existing codebase.

Review of the FirstTour prototype has confirmed the implementation approach's ef-
fectiveness in automating the transformation process. Tests on various Theorema note-
book structures demonstrated the accuracy of the LATEXoutput, which faithfully repre-
sented the input notebooks, including complex nested formulas and logical constructs.
Initial feedback from the main user, the Theorema developer, is positive, highlighting
the tool's ability to reduce the manual effort required for LATEXpreparation significantly.

In conclusion, the project successfully addressed the initial project objective and pro-
vides a functional tool for automating document preparation in Theorema. The project
also underscored the potential of Wolfram Language as a versatile tool for software en-
gineering, especially in the context of mathematical document processing: the present
work elaborates on the various engineering concepts in this context and presents research
on the Wolfram ecosystem extending the language in important ways in the current day.
Theorema itself, as a mature WL package and system, provided an ideal entry point to
this work.

5.3 Final Closing Remarks: Wolfram Language as a Software Engineering Tool and Integrating with Other Languages and Environments, Potential Future Work

5.3.1 Using Wolfram Language for Software Engineering

This project has demonstrated a use case for mathematical software use, the original
domain of Mathematica. In the process of researching and reading the documentation,
this researcher discovered a rich ecosystem for WL existing today, including not only
integrations with modern enterprise software frameworks, but actually a sophisticated
implementation of all the modern software engineering concepts and methodologies,
which this work has attempted to demonstrate in structured way.

5.3.2 Potential Future Work

Potential future work hinges on the possible integrations with the native TeXForm
functionality, for a more elaborate implementation of a to-LATEXtypsetting system and
less reliance on output-side macros (the LATEXtemplate file): possible approaches were
sketched out in chapter 3.

Aside from this, the immediately next task to be done is an integration into the
Theorema package itself, if the functionality is to be added to the project.

Appendix A

Technical Documentation/Source Code

The project repository is synced in its entirety with this document and listed below.

Development in the main branch is frozen on the submission date. Any changes after the submission of this document will be separated into a new branch. The final version will also be submitted with this document, see Appendix B (Supplementary Materials).

The final version with any further development can be viewed online at https://github.com/heseltime/Tma2TeX.

Listing A.1: Main Program

```
 1
 2
 3  (* Wolfram Language Raw Program *)
 4
 5  BeginPackage["Tma2tex`"];
 6
 7  (* ---- written by Jack Heseltine, July 2023 - July 2024
 8    Updates
 9      - August - December 2023: Set up project structure and basic recursion rule tests
10      - January 2024: Introduce common WL-Package structure
11      - February 2024: Add convertToLatexFromString for Cloud-testing
12      - March/April 2024: Split recursion into parseNbContent and getTmaData/parseTmaData
13      - May/June 2024: try approach with TeXForm transformation
14      - [Test-wise: July 2024: MakeTeX refactor using TeXForm implementation as basis]
15      - August 2024: Back to the Basics with Formula-In-Macro-Structure-Out Approach, in
        Cleaned-Up File
16        - Move pre-August 2024 work to "V0"
17
18    Purpose: This program recurses over the Theorema notebook structure to produce a LaTeX
        representation, including of the
19    underlying Theorema-Datastructure: to this end it inserts the appropriate LaTeX-
        commands into an output file, mediated
20    by a template in the $resDir. Both the notebook- and Theorema-level content is rendered
        using the relevant LaTeX packages
21    and should be modified in LaTeX for the output syntax.
22
23    Part 1 of this package is concerned with the described double-recursion, drawing mainly
        on the Theorema-data provisioned in Part 0.
24    For academic purposes, Part 1 is subdivided in Parts
25
26      * A: parseNbContent, the main recursive function, with the output-nearer LaTeX-
```

```
        commands,
27       * B: also parseNbContent, higher level, Theorema−notebook specific pattern−recursion
         rules,
28       * C: getTmaData and parseTmaData, concerned with establishing the connection between
         the appropriate part in the input notebook/
29         output LaTeX and the given Theorema data, and parsing, again recursively, the formula
         structure, respectively.
30
31     Part 0 is also subdivided in an out/inside−of−package Part A and B respectively, to
         illustrate packaging in Wolfram Language.
32
33     The result is inserted into the appropriate LaTeX template (Part 2, Filehandling): the
         main functions intended
34     for use by the client is at the end of the program, Part 3, specifically
         convertToLatexAndPdfDocs[] as the all−in−one transformation
35     function for stand−alone−calling − but convertToLatexDoc[] for the basic Theorema use
         case. Both functions take the path to the relevant
36     Theorema notebook as their single parameter.
37
38     −−−− ∗)
39
40
41
42
43 (∗ −− Part 0, Imports and Global Variables as per Theorema Specification −− ∗)
44
45 (∗ −− Part 0.A, Imports and Global Variables OUTSIDE−OF−PACKAGE −− ∗)
46
47 (∗ −− Part 0.A.1 Optional Theorema−Import with Get −− ∗)
48
49 (∗ << Theorema` ∗)
50 (∗ Uncomment the Tma−Get call if NOT called from inside the Tma−Package or an
        environment that loads Tma already ∗)
51
52 Needs["Theorema`"]
53
54 (∗ −− Part 0.A.2 Global Variables: Important for interfacing with Theorema. −− ∗)
55 Tma2tex`$resDir::usage = "Defines the directory for LaTeX-templates and any other
        resources."
56
57 Tma2tex`$resDir = "C:\\Users\\jackh\\git\\repository\\tma2tex\\res"
58
59
60 Tma2tex`$tmaData::usage = "Contains the Theorema-Datastructure that holds formula-
        experessions and is therefore typically equivalent to
61   Theorema`Common`$tmaEnv on the Theorema-side, but can be used to show the content
        according to Tma2Tex (as a separate package)."
62
63 Tma2tex`$tmaData = Theorema`Common`$tmaEnv;
64
65
66 (∗ −− Part 0.A.3 Client−Function−Usage Messages ∗)
67 convertToLatexDoc::usage="convertToLatexDoc[notebookPath] transforms a given WL
        notebook (by file path) to TeX output, creating a new TeX file from a specified
        resource template."
68 convertToLatexAndPdfDocs::usage="convertToLatexAndPdfDocs[notebookPath] transforms a
         given WL notebook (by file path) to PDF file as final output, with TeX file as
```

```
              intermediary step, from a specified resource template."
69
70  (* Test-wise: *)
71  convertToLatexFromString::usage="convertToLatexFromString[nbContentString_,
        resourceDir_Optional]: Tma2tex`$resDir] is experimental and intended be called
        from the Cloud, simply transofrming Wolfram Language String Input to TeX Output
        (returned directly, not via file). Also uses a template, the resource for which
        can be passed as the second argument."
72
73  Begin["`Private`"]
74
75  (*Needs["Texformdump`"]*) (* loaded in Private` for internal implementation details *)
76  (* See TeXForm Notes in the Thesis: this approach was not chosen, finally . *)
77
78  (* -- Part 0.B, Imports and Global Variables INSIDE-OF-PACKAGE -- *)
79
80  (* Custom messages for problems importing or invalid  settings *)
81  tmaDataImport::empty = "`1`";
82  documentProcessingLevel::invalidParameter = "`1`";
83
84  (* The following holds the Tma-Formula-List as an association with keys from the IDs, gets
        filled  in getTmaData[] *)
85  tmaDataAssoc = <||>;
86
87  (* Define the global private variable with a default value *)
88  Tma2tex`$documentProcessingLevel = ""; (* or "Full" or "None"
89    where leaving it empty defaults to basic text (some parseNbContent) and formulas (
        parseTmaData),
90    "Full" gives more content by visualizing  types of WL-structures (lists, cells ) ommitted (
        extensible ),
91    and "None" gives only the formulas *)
92
93  (* -- Part 1.A, Recursive Pattern Matching: parseNbContent[] with a focus on (mathematical)
        symbol-level transformations -- *)
94
95  (* -- Part 1.A.0 -- Structural Expressions: Colored diamonds to demarcate structural text
        output from content
96    - Only process these  if DocumentProcesingLevel is "Full" *)
97  parseNbContent[___] = If[$documentProcessingLevel == "Full", "\\colordiamond{black}"
        , ""];
98
99  parseNbContent[Notebook[l_List, ___]] := If[$documentProcessingLevel == "Full", "\\
        legend \\n\\n \\colordiamond{yellow}", ""] <> parseNbContent[l]
100   (* goes to parseNbContent[l_List], this our entry point to parsing *)
101
102 parseNbContent[c_Cell] := If[$documentProcessingLevel == "Full", "\\colordiamond{red
        }", ""] (* matches Cells that are not further specified (as relevant WL or TMA cells) below
        *)
103
104 parseNbContent[l_List] := If[$documentProcessingLevel == "Full", "\\colordiamond{
        blue}", ""] (* matches Lists that are not further specified (as relevant WL or TMA cells)
        below *)
105 parseNbContent[l_List] /; MemberQ[l, _Cell] := StringJoin[If[
        $documentProcessingLevel == "Full", "\\colordiamond{purple}", ""], ToString /@
        parseNbContent /@ l]
106   (* matches Lists with  at  least  one Cell *)
107
```

```
108  parseNbContent[Cell[CellGroupData[l_List, ___], ___]] := If[$documentProcessingLevel
          == "Full", "\\colordiamond{green}", ""] <> parseNbContent[l]
109    (* CellGroupData often contain relevant content *)
110
111
112  (* -- Part 1.A.1 -- Text Expressions (at the Cell Level): Not processed if
          DocumentProcessingLevel = "None", otherwise yes. *)
113
114  parseNbContent[Cell[text_String, "Text", ___]] := If[$documentProcessingLevel != "
          None", "\\begingroup \\section*{} " <> text <> "\\endgroup \n\n", ""]
115
116  parseNbContent[Cell[text_String, "Section", ___]] := If[$documentProcessingLevel !=
          "None", "\\section{" <> text <> "}\n\n", ""]
117
118
119  (* -- Part 1.A.2 -- Text/Math/Symbols at the String Level *)
120
121  (* Example: Operators *)
122  parseNbContent["<"] := If[$documentProcessingLevel != "None", "\\textless", ""]
123
124  parseNbContent[">"] := If[$documentProcessingLevel != "None", "\\textgreater", ""]
125
126  (* Example: Greek Letters *)
127  parseNbContent["\[CapitalDelta]"] := If[$documentProcessingLevel != "None", "\\Delta
          ", ""]
128
129
130  (* -- Part 1.A.3 -- Boxes *)
131
132  parseNbContent[Cell[BoxData[FormBox[content_, TraditionalForm]], "DisplayFormula",
          ___]] :=
133      If[$documentProcessingLevel != "None", StringJoin["\\begin{center}",
          parseNbContent[content], "\\end{center}\n"], ""]
134
135  (* This particular rule does a lot of the parsing through the Tma-Env. *)
136  parseNbContent[RowBox[list_List]] :=
137      If[$documentProcessingLevel != "None", StringJoin[parseNbContent /@ list], ""]
138
139  (* Underscriptboxes *)
140  parseNbContent[UnderscriptBox[base_, script_]] :=
141      If[$documentProcessingLevel != "None", StringJoin["\\underset{", parseNbContent[
          script], "}{", parseNbContent[base], "}"], ""]
142
143  parseNbContent[UnderscriptBox["\[Exists]", cond_]] :=
144      If[$documentProcessingLevel != "None", "\\underset{" <> parseNbContent[cond] <>
          "}{\\exists}", ""]
145  parseNbContent[UnderscriptBox["\[ForAll]", cond_]] :=
146      If[$documentProcessingLevel != "None", "\\underset{" <> parseNbContent[cond] <>
          "}{\\forall}", ""]
147
148  (* -- Part 1.A.4 -- Box-Structure Parsing - See TeXForm Notes in Thesis and compare
          parseTmaData,
149    this was the formula parsing approach finally selected here *)
150
151  parseNbContent[RowBox[{left_, "\[And]", right_}]] :=
152      If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left], " \\land
          ", parseNbContent[right]], ""]
```

```
153
154 parseNbContent[RowBox[{left_, "\[Or]", right_}]] :=
155     If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left], " \\lor
        ", parseNbContent[right]], ""]
156
157 parseNbContent[RowBox[{left_, "\[DoubleLeftRightArrow]", right_}]] :=
158     If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left], " \\
        Leftrightarrow ", parseNbContent[right]], ""]
159
160 parseNbContent[RowBox[{left_, "\[Implies]", right_}]] :=
161     If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left], " \\
        Rightarrow ", parseNbContent[right]], ""]
162
163 parseNbContent[RowBox[{left_, "<", right_}]] :=
164     If[$documentProcessingLevel != "None", parseNbContent[left] <> " < " <>
        parseNbContent[right], ""]
165
166 parseNbContent[RowBox[{left_, ">", right_}]] :=
167     If[$documentProcessingLevel != "None", parseNbContent[left] <> " >  " <>
        parseNbContent[right], ""]
168
169 parseNbContent[RowBox[{left_, "\[Equal]", right_}]] :=
170     If[$documentProcessingLevel != "None", parseNbContent[left] <> " = " <>
        parseNbContent[right], ""]
171
172 parseNbContent[RowBox[{left_, "\[SubsetEqual]", right_}]] :=
173   If[$documentProcessingLevel != "None", parseNbContent[left] <> "\\subseteq" <>
        parseNbContent[right], ""]
174
175 parseNbContent[RowBox[{left_, "\[Element]", right_}]] :=
176   If[$documentProcessingLevel != "None", parseNbContent[left] <> "\\in" <>
        parseNbContent[right], ""]
177
178
179
180 (* -- Part 1.B.0 -- Theorema-Language/-Notebook-specific Expressions,
181   these are the jumping off point to the second kind of recursive descent in this program,
182   parsing through the Theorema-Datastructure *)
183
184 (* Not needed: separating line in the tmanotebook *)
185
186 (*parseNbContent[Cell["", "OpenEnvironment", ____]] :=
187     "\\begin{openenvironment}\n\\end{openenvironment}"
188 *)
189
190 (* Not needed: the following  cell -> cellgroup -> list of cells  is the environment, with
        environment cells  after  the  header-cell *)
191
192 (*parseNbContent[Cell[CellGroupData[{Cell[headertext_, "EnvironmentHeader", options____],
        envcells____}, ____]]] :=
193     Module[{contentStrings},
194         contentStrings = StringJoin[parseNbContent /@ {envcells}]; (* Apply parsing to each
        cell *)
195         StringJoin[
196             "\\begin{tmaenvironment}\n",
197             "\\subsection{", parseNbContent[headertext], "}\n",
198             contentStrings,
```

```
199              "\\end{tmaenvironment}\n"
200          ]
201      ]
202  *)
203
204  (* Main Parser-Function to capture relevant formulas and create the link to the Tma-
         Datastructure *)
205  parseNbContent[
206      Cell[CellGroupData[{Cell[headertext_, "EnvironmentHeader", headeroptions___],
207                          Cell[formulaboxdata_, "FormalTextInputFormula", options___],
208                          furtherNotebookEnvCells___},
209                          envOptions___]]] :=
210  Module[{contentStrings, cellID, optionsAssociation, headerParts, smallCapsPart,
         regularPart},
211
212      (* Split the header text at the opening parenthesis and format parts accordingly *)
213      headerParts = StringSplit[headertext, "("];
214      smallCapsPart = StringTrim@ToLowerCase[headerParts[[1]]]; (* Make the first part
         small caps *)
215      regularPart = StringReplace[headerParts[[2]], ")" -> ""]; (* Remove the closing
         parenthesis *)
216
217      contentStrings = StringJoin[parseNbContent /@ {furtherNotebookEnvCells}];
218      optionsAssociation = Association[options];
219      cellID = optionsAssociation[CellID];
220
221      (* Print statement for debugging purposes *)
222      Print[smallCapsPart <> " " <> ToString[cellID] <> " found, linking to Tma-Data
         ..."];
223
224      StringJoin[
225          "\\EnvironmentWithFormat{", smallCapsPart, "}{", regularPart, "}\n",
226          If[cellID =!= None,
227              formatTmaData@parseTmaData[getTmaData[cellID]],
228              StringJoin["\\textcolor{red}{", "No ID Found: Did you load Theorema and
         evaluate the Theorema cells from the same kernel as this call?", "}\n"]
229          ],
230          "\\end{EnvironmentWithFormat}\n\n"
231      ]
232  ]
233
234
235
236  (* Same as above but with inermediate Tma-Defintions: these are captured  directly  in the
         formula gotten with getTmaData however *)
237  parseNbContent[Cell[CellGroupData[{Cell[headertext_, "EnvironmentHeader",
         headeroptions___],
238  (* Tma-Definitions: *) __, (* These could be outputed to PDF if needed *)
239  Cell[formulaboxdata_, "FormalTextInputFormula", options___],
240  furtherNotebookEnvCells___},
241  envOptions___]]] :=
242      Module[{contentStrings, cellID, optionsAssociation, headerParts, smallCapsPart
         , regularPart},
243
244          (* Split the header text at the opening parenthesis and format parts accordingly *)
245          headerParts = StringSplit[headertext, "("];
246          smallCapsPart = StringTrim@ToLowerCase[headerParts[[1]]]; (* Make the first
```

```
         part small caps *)
247          regularPart = StringReplace[headerParts[[2]], ")" -> ""]; (* Remove the
     closing parenthesis *)
248
249          contentStrings = StringJoin[parseNbContent /@ {furtherNotebookEnvCells}];
250          optionsAssociation = Association[options];
251          cellID = optionsAssociation[CellID];
252
253          (* Print statement for debugging purposes *)
254          Print[smallCapsPart <> " " <> ToString[cellID] <> " with helper definitions
     found, linking to Tma-Data ..."];
255
256          StringJoin[
257              "\\EnvironmentWithFormat{", smallCapsPart, "}{", regularPart, "}\n",
258              If[$documentProcessingLevel == "Full", "\\colordiamond{orange}", ""], (*
     This is included for explainability in the PDF-output *)
259              If[cellID =!= None,
260                  formatTmaData@parseTmaData[getTmaData[cellID]],
261                  StringJoin["\\textcolor{red}{", "No ID Found: Did you load Theorema
     and evaluate the Theorema cells from the same kernel as this call?", "}\n"]
262              ],
263              "\\end{EnvironmentWithFormat}\n\n"
264          ]
265      ]
266
267 (* Similar to Tma-Envs, but no Header Text: these shoudl also not be printed *)
268
269 (*parseNbContent[Cell[BoxData[
270  RowBox[{envcells____}]], "GlobalDeclaration", ____]] :=
271   Module[{contentStrings},
272     contentStrings = StringJoin[parseNbContent /@ {envcells}];
273         StringJoin[
274             "\\begin{tmaenvironmentgd}\n",
275             "\\subsubsection{Global Declaration}\n", (* Maybe *)
276              contentStrings,
277              "\\end{tmaenvironmentgd}\n"
278         ]
279   ]*)
280
281
282 (* Global Declarations should not be printed *)
283
284 (*parseNbContent[Cell[BoxData[content_], "GlobalDeclaration", ____]] :=
285     Module[{contentStrings},
286         contentStrings = parseNbContent[content]; (* Directly pass the content to
     parseNbContent *)
287         StringJoin[
288             "\\begin{tmaenvironmentgd}\n",
289             "\\subsubsection{Global Declaration}\n", (* Optional title *)
290             contentStrings,
291             "\\end{tmaenvironmentgd}\n"
292         ]
293     ]
294 *)
295
296
297
```

```
298  (* Parse the cells in the theorema environment list one by one: the empty string below
         generally marks the beginning of a Tma Cell in the TeX *)
299  parseNbContent[Cell[BoxData[rowboxes___], "FormalTextInputFormula", ___]] := "" <>
         StringJoin[parseNbContent /@ {rowboxes}]
300
301  (* ensure we handle nested RowBox instances correctly by recursively parsing their content *)
302  parseNbContent[RowBox[list_List]] :=
303      StringJoin[parseNbContent /@ list]
304
305  (* rowbox on list in Part 1A.0.2.0 *)
306
307  (* Tma−Env elements that occur within {lists}, often inside RowBox[] *)
308  parseNbContent[TagBox["(","AutoParentheses"]] := "\\left("
309  parseNbContent[TagBox[")","AutoParentheses"]] := "\\right)"
310  (*parseNbContent[UnderscriptBox["\[ForAll]", "x"]] := "forAll "*)
311  parseNbContent[RowBox[{n_, "[", var_, "]"}]] := n <> "[" <> var <> "]"
312  parseNbContent[TagBox["\[DoubleLeftRightArrow]", ___]] := " \\Leftrightarrow "
313
314  (* Subscriptboxes *)
315  parseNbContent[SubscriptBox[base_, subscript_]] :=
316      parseNbContent[base] <> "_{" <> parseNbContent[subscript] <> "}"
317
318  parseNbContent[TagBox[content_, _, SyntaxForm -> "a\[Implies]b"]] :=
319      "\\rightarrow "
320
321
322  parseNbContent[Cell["\[GraySquare]", "EndEnvironmentMarker", ___]] :=
323      " \\graysquare{}"
324
325
326  (* −− Part 1.B.1 −− Out−of−Flow Expressions: Reap and Sow mechanism to process in a
         different order than the expressions are encountered in *)
327
328  parseNbContent[Cell[t_String, "Title", ___]] := (Sow[t, "title"]; Sow["", "author"];
         Sow["", "date"]; "") (* author and date currently not included in sample doc *)
329
330
331
332  (* −− Part 1.B.2 −− Key for Testing? Highlight Unclaimed Expressions *)
333
334  parseNbContent[other_] := StringJoin["\\textcolor{red}{", "Pattern not found! ",
         ToString[other], "}"]
335
336
337  (* −− Part 1.B.3 −− String Processing for Symbols Occuring In−text in the Notebook *)
338  parseNbContent[s_String] := StringReplace[s, "\[SubsetEqual]" -> "\\subseteq"]
339
340
341
342  (* −− Part 1.C.0, Recursive Pattern Matching: getTmaData[] selects the relevant part in
         Theorema`Common`FML$ in preparation
343    for a second recursive descent, see 1.B.2 −− *)
344
345  getTmaData[id_Integer] := Module[{assoc, cleanStringKeysAssoc, numericKeysAssoc},
346      assoc = Association[Cases[$tmaData, Theorema`Common`FML$[{idFormula_, _}, expr_,
         no_] :> (idFormula -> expr), {1}]];
347      cleanStringKeysAssoc = Association[StringReplace[#, "ID:" -> ""] -> assoc[#] & /
```

```
          @ Keys[assoc]];
348     numericKeysAssoc = Association[ToExpression[#] -> cleanStringKeysAssoc[#] & /@
          Keys[cleanStringKeysAssoc]];
349     numericKeysAssoc[id]
350 ]
351
352 (* −− Part 1.C.1, Recursive Pattern Matching: Second Recursive Descent  −− *)
353
354 (* Alternative 1: Operation is known, it is in the language:
355   In this case, there should be a TeX−macro, i.e. Curly−Brackets−Case *)
356 parseTmaData[op_?isTmaLanguageSymbol[args___]] :=
357   Module[{nextOp, argList, parsedArgs},
358   nextOp = prepareSymbolName[op];
359   argList = {args};
360     parsedArgs = Switch[
361       Length[argList], (* 1 argument in the majority of cases *)
362       1, "{" <> parseTmaData[argList[[1]]] <> "}",
363       2, "{" <> parseTmaData[argList[[1]]] <> "}{" <> parseTmaData[argList[[2]]] <>
        "}",
364       3, "{" <> parseTmaData[argList[[1]]] <> "}{" <> parseTmaData[argList[[3]]] <>
        "}",
365       _, "" (* unexpected number of arguments: abort the parse tree here *)
366     ];
367     " \\" <> ToString[nextOp] <> parsedArgs
368   ]
369
370 (* Alternative 2: Knowledge−case − Predicate or Function Symbol, not Language−Operator *)
371 parseTmaData[op_[args___]] :=
372   Module[{nextOp, argList, parsedArgs},
373   nextOp = prepareSymbolName[op];
374   argList = {args};
375     parsedArgs = Switch[
376       Length[argList], (* 1 argument in the majority of cases *)
377       1, "[" <> parseTmaData[argList[[1]]] <> "]",
378       2, "[" <> parseTmaData[argList[[1]]] <> ", " <> parseTmaData[argList[[2]]] <>
        "]",
379       _, "" (* unexpected number of arguments: abort the parse tree here *)
380     ];
381     " " <> ToString[nextOp] <> parsedArgs (* Does not get prefixed with "\"! *)
382   ]
383
384 (* Alternative 3: Special Case Two−Agument−Sets e.g.
385
386   Theorema`Language`Annotated$TM[Theorema`Language`Less$TM,
387   Theorema`Language`SubScript$TM[Theorema`Knowledge`lex$TM]]
388     [Theorema`Language`VAR$[Theorema`Knowledge`VAR$a$TM],
389   Theorema`Language`VAR$[Theorema`Knowledge`VAR$b$TM]]
390
391   ... transforms to more complex macro *)
392 parseTmaData[op_?isTmaLanguageSymbol[args___][args2___]] :=
393   Module[{nextOp, argList, argList2, parsedArgs, parsedArgs2},
394     nextOp = prepareSymbolName[op];
395     argList = {args};
396   parsedArgs = Switch[
397     Length[argList],
398     1, "{" <> parseTmaData[argList[[1]]] <> "}",
399     2, "{" <> parseTmaData[argList[[1]]] <> "}{" <> parseTmaData[argList[[2]]] <> "}
```

```
      ",
400   3, "{" <> parseTmaData[argList[[1]]] <> "}{" <> parseTmaData[argList[[3]]] <> "}
      ",
401   _, "" (* Unexpected number of arguments: stop parsing here *)
402 ];
403
404   argList2 = {args2};
405 parsedArgs2 = Switch[
406   Length[argList2],
407   1, "{" <> parseTmaData[argList2[[1]]] <> "}",
408   2, "{" <> parseTmaData[argList2[[1]]] <> "}{" <> parseTmaData[argList2[[2]]] <>
      "}",
409   3, "{" <> parseTmaData[argList[[1]]] <> "}{" <> parseTmaData[argList[[3]]] <> "}
      ",
410   _, "" (* Unexpected number of arguments: stop parsing here *)
411 ];
412   " \\" <> ToString[nextOp] <> parsedArgs <> parsedArgs2
413 ]
414
415 (* Special Case/Alternative 4: Numbers *)
416 parseTmaData[i_Integer] := (* e.g. in Theorema`Language`VAR$[Theorema`Knowledge`
      VAR$m1$TM], 2],
417  2 eventually gets processed *)
418   ToString[i]
419
420 (* Recursion−Stop (Alternative 5) Axiomatic Expression/No Operation *)
421 parseTmaData[ax_] := (* e.g. Theorema`Knowledge`VAR$x$TM, i.e. axioms/parse−tree leaves
      *)
422   prepareSymbolName[ax]
423
424 (* −− Part 1.C.2, Auxilliary Functionality −− *)
425
426 isTmaLanguageSymbol[f_Symbol] := (* Context required here to distinguish from predicates like
        Theorema`Knowledge`P$TM *)
427   With[ {n = SymbolName[f], c = Context[f]},
428       c === "Theorema`Language`"
429   ]
430 isTmaLanguageSymbol[f_] := False
431
432 prepareSymbolName[op_Symbol] :=
433   With[{n = SymbolName[op]},
434       If[StringTake[n, -3] == "$TM",
435         If[StringTake[n, 4] == "VAR$", (* e.g. Theorema`Knowledge`VAR$x$TM *)
436           (*"\\" <> *)StringDrop[StringDrop[n, 4], -3](* <> "TM"*),
437           (*else*) (* e.g. Theorema`Language`Forall$TM, but: Theorema`Knowledge`Q$TM
      *)
438             If[isTmaLanguageSymbol[op], (* only language should be suffixed with TM for
      TeX−Macros *)
439               (*"\\" <> *)StringDrop[n, -3] <> "TM",
440             (*else*)
441               StringDrop[n, -3]
442             ]
443         ],
444       (*else*)
445         If[StringTake[n, -1] == "$", (* e.g. VAR$ *)
446           StringDrop[n, -1] <> "TM",
447         (*else*)
```

```
448              n <> "TM"
449          ]
450        ]
451     ]
452
453 (* —— Part 1.C.3, Final String—level Replacements —— *)
454
455 formatTmaData[parsedExpression_String] :=
456   Module[{replacedString}, (* As needed *)
457   replacedString = StringReplace[parsedExpression, ""->""];
458   replacedString <> "\\n\\n" (* Take care that LaTeX outputs are on their own lines here *)
459 ]
460
461 (* —— Part 2, Filehandling —— *)
462
463 writeToLatexDoc[latexPath_, nbContent_] :=
464  Module[{strm }, strm = OpenWrite[latexPath];
465   WriteString[strm, parseNbContent[nbContent]];
466   Close[strm]] (* stream handling, call to pattern Matching part *)
467
468 getLatexPath[notebookPath_String] :=
469  Module[{directory, baseName, latexExtension = ".tex"},
470   directory = DirectoryName[notebookPath];
471   baseName = FileBaseName[notebookPath];
472   FileNameJoin[{directory,
473     baseName <>
474      latexExtension}]] (* get the latex file name (the new .tex file \
475 that is being filled with the nb—content) with full path from a given \
476 notebook path*)
477
478 getLatexTemplatePath[notebookPath_String] :=
479  Module[{directory, baseName, latexExtension = ".tex"},
480   directory = DirectoryName[notebookPath];
481   baseName = FileBaseName[notebookPath] <> "TheoremaTemplate";
482   FileNameJoin[{directory,
483     baseName <>
484      latexExtension}]] (* get the latex file name (the .tex—template \
485 which includes the main nb—content—.tex—file correctly) with full \
486 path from a given notebook path*)
487
488 fillLatexTemplate[resDir_String, data_Association] :=
489  Module[{texContent, template,
490    filledContent},(*Import the LaTeX template*)
491   texContent =
492    Import[FileNameJoin[{resDir, "tmaTemplate.tex"}], "Text"];
493   (*no template object needed here*)template = texContent;
494   (*Apply the data to the template*)
495   filledContent = TemplateApply[template, data];
496   (*Return the filled  content*)filledContent]
497
498
499 (* —— Part 3, Main Functions for Client —— *)
500 Options[convertToLatexDoc] = {DocumentProcessingLevel -> ""};
501 convertToLatexDoc[notebookPath_, OptionsPattern[]] :=  Module[{nb, content,
       latexPath, latexTemplatePath,
502    resourceDir = $resDir, texResult, sownData, filledContent, closeFlag = False,
       documentProcessingLevel},
```

```
503    If[Length[$tmaData] == 0, (* Issue message if Theorema−Formula−Data not provisioned *)
504    Message[tmaDataImport::empty, "The Theorema-Formula-Datastructure is empty.
505      Did you evaluate a Theorema notebook before loading the package and calling the
          conversion function?"];
506    (* Additional handling for empty data can be added here *)
507    Return[$Failed]
508    ];
509
510    (* Retrieve, validate and set document processing level *)
511    documentProcessingLevel = OptionValue[DocumentProcessingLevel];
512    If[documentContentProcessingLevel =!= "", (* Default for fns and pkg is empty string *)
513      SetDocumentProcessingLevel[documentProcessingLevel]
514    ];
515
516    nb = If[isNotebookOpen[notebookPath],
517      NotebookOpen[notebookPath],
518      NotebookOpen[notebookPath, Visible->False]; closeFlag = True];
519
520    content = NotebookGet[nb];
521    NotebookEvaluate[content]; (* on content: important,
522      so that Tma env. variables are available in any case *)
523    latexPath = getLatexPath[notebookPath];
524    latexTemplatePath = getLatexTemplatePath[notebookPath];
525    (* filledContent =
526      fillLatexTemplate [
527      resourceDir, <|"nbName" −> FileBaseName[notebookPath]|>];*)
528    {texResult, sownData} = Reap[parseNbContent[content], {"title", "author", "date"
          }];
529    filledContent = fillLatexTemplate[resourceDir,
530    <|
531      "nbContent" -> texResult,
532      "nbTitle" -> First[sownData[[1, 1]]],
533      "nbAuthor" -> First[sownData[[2, 1]]],
534      "nbDate" -> First[sownData[[3, 1]]]
535    |>];
536    Export[latexPath, filledContent, "Text"];
537    (*Print[Theorema`Common`$tmaEnv];*)
538
539    If[closeFlag === True, NotebookClose[notebookPath]];
540  ]
541
542  (* Helper fn to determin if the notebook specified by the given path is open *)
543  isNotebookOpen[path_] :=
544  Module[{c},
545    Quiet[MemberQ[Notebooks[],
546      n_ /; (c = "FileName" /. NotebookInformation[n]; c[[2]]) ===
547        FileNameTake[path, -1]]]]
548
549  (* Helper fn to set the document processing level *)
550  SetDocumentProcessingLevel[level_String] :=
551  Module[{},
552    If[MemberQ[{"Full", "None", ""}, level],
553      Tma2tex`$documentProcessingLevel = level,
554      Message[documentProcessingLevel::invalidParameter, "Invalid parameter (option)
        DocumentProcessingLevel set: should be empty string (default), Full, or None."];
555    Return[$Failed]
556    ]
```

```
557  ]
558
559  Options[convertToLatexAndPdfDocs] = {DocumentProcessingLevel -> ""};
560  convertToLatexAndPdfDocs[notebookPath_, OptionsPattern[]] := Module[{latexPath,
          pdfPath, compileCmd, conversionResult},
561    conversionResult = convertToLatexDoc[notebookPath, DocumentProcessingLevel->
          OptionValue[DocumentProcessingLevel]];
562    If[conversionResult === $Failed,
563      Return[$Failed]
564    ];
565    (* Compile LaTeX to PDF using pdflatex *)
566    latexPath = getLatexPath[notebookPath];
567    pdfPath = StringReplace[latexPath, ".tex" -> ".pdf"];
568    compileCmd =
569     "pdflatex -interaction=nonstopmode -output-directory=" <>
570      DirectoryName[latexPath] <> " " <> latexPath;
571    RunProcess[{"cmd", "/c", compileCmd}];
572  ]
573
574  (* Test-Implementation *)
575  convertToLatexFromString[nbContentString_, resourceDir_Optional: Tma2tex`$resDir] :=
          Module[
576      {nbContent, texResult, sownData, filledContent},
577
578      (* Convert the string representation to a Wolfram Language expression *)
579      nbContent = ToExpression[nbContentString, InputForm];
580
581      (* Process the notebook content *)
582      {texResult, sownData} = Reap[parseNbContent[nbContent], {"title", "author", "
          date"}];
583
584      (* Fill in the LaTeX template with parsed content *)
585      filledContent = fillLatexTemplate[resourceDir,
586          <|
587              "nbContent" -> texResult,
588              "nbTitle" -> First[sownData[[1, 1]]],
589              "nbAuthor" -> First[sownData[[2, 1]]],
590              "nbDate" -> First[sownData[[3, 1]]]
591          |>
592      ];
593
594      (* Return the filled LaTeX content as a string *)
595      filledContent
596  ]
597
598
599  (*Remove["Texformdump`*"]*)
600
601  End[]
602
603  EndPackage[];
```

Note: The following is the output of a test notebook (tma2tex/tma2tex.nb in the repo) that calls the main program printed above; the Mathematica-notebook-basis (the input notebook) is not printed here due to its length but also available in the repo.

Listing A.2: Ërst TourÖutput in LaTeX Format

```latex
 1  %% AMS−LaTeX Created with the Wolfram Language : www.wolfram.com
 2  % from Wolfram template for LaTeX
 3
 4  \documentclass{article}
 5
 6  %% Packages
 7  \usepackage{amsmath, amssymb, graphics, setspace, xcolor}
 8
 9  % for symbol encoding, e.g. "<", see also the following resource:
10  % https://tex.stackexchange.com/questions/2369/why−do−the−less−than−symbol−and−the−
         greater−than−symbol−appear−wrong−as
11  \usepackage{lmodern}
12
13  %% For development purposes
14  % Define a command for light gray text for structure elements
15  \newcommand{\light}[1]{{\color{lightgray}#1}}
16
17  %% Structural elements of LaTeX document
18  % Gray box for Tma envs
19  \usepackage{tcolorbox}
20
21  %% Approach 1: Define a new tcolorbox environment for theorema environments
22  \newtcolorbox{tmaenvironment}{
23    colback=gray!20, % Background color: gray with 20% intensity
24    colframe=black, % Frame color: black
25    boxrule=0, % Frame thickness
26    arc=4pt, % Corner rounding
27    boxsep=5pt, % Space between content and box edge
28    left=5pt, % Left interior padding
29    right=5pt % Right interior padding
30  }
31
32  %% Structural elements of the symbol level
33  % QED − gray square
34  \usepackage{tikz}
35  \newcommand{\graysquare}{\tikz\fill[gray] (0,0) rectangle (0.2cm,0.2cm);\hspace{0.2
         cm}}
36
37  % Centered diamond shape in a specified color
38  \newcommand{\colordiamond}[1]{%
39      \tikz[baseline={(([yshift=-0.3ex]current bounding box.center)}]
40      \fill[#1] (0,0) -- (0.08cm,0.08cm) -- (0,0.16cm) -- (-0.08cm,0.08cm) -- cycle;\
         xspace
41  }
42
43  % e.g.
44  % \colordiamond{red}
45  % \colordiamond{green}
46  % \colordiamond{blue}
47
48  % Define a command to create a right−aligned, smaller legend
49  \newcommand{\legend}{
50    \begin{tcolorbox}[
51      title=Tma2tex-parsing Info/Legend,
52      colback=white,
53      colframe=black,
54      boxrule=0.5mm,
```

```
55      arc=4pt,
56      width=6cm, % Adjust width to make it smaller
57      float=right, % Moves the legend to the right
58      halign=left, % Aligns the text within the box to the  left
59      left=2mm, % Interior left margin
60      right=2mm, % Interior right margin
61      boxsep=1mm % Reduces the separation between the text and the border
62    ]
63      \colordiamond{yellow} Yellow: Represents entry points to parsing. \\
64      \colordiamond{orange} Orange: Helper Definitions were defined in the Theorema
          Notebook interface, but are directly included in the following formula. \\
65      \colordiamond{red} Red: Matches unspecified cells or generic content. \\
66      \colordiamond{blue} Blue: Represents lists of specific content. \\
67      \colordiamond{purple} Purple: Used for lists of generic cells. \\
68      \colordiamond{green} Green: Represents a CellGroupData Element with a List
          inside, a relevant content structure typically. \\
69    \end{tcolorbox}
70 }
71
72
73 %% Approach 2: Dynamic environments
74 \usepackage{xparse} % To allow more flexible macro definitions
75
76 % Define a generic environment handler
77 \NewDocumentCommand{\EnvironmentWithFormat}{m m o}{%
78      \IfValueTF{#3} % Check if there is an optional argument
79      {%
80          \begin{#1}[#3]%
81          \textsc{#2}%
82          \newline % Add a newline before the content
83      }
84      {%
85          \begin{#1}%
86          \textsc{#2}%
87          \newline % Add a newline before the content
88      }
89 }
90
91 % Define sample environments for demonstration
92 \newenvironment{proposition}[1][]{
93   \noindent\textbf{Proposition #1:}
94 }{\par}
95
96 \newenvironment{definition}[1][]{
97   \noindent\textbf{Definition #1:}
98 }{\par}
99
100 \newenvironment{lemma}[1][]{
101   \noindent\textbf{Lemma #1:}
102 }{\par}
103
104
105 %% ---
106 %% Theorema Symbols are defined as custom LaTeX commands here! (BEGIN)
107 %%
108 %% In no particular order, except that it generally matches the parsing rules found in tma2
        tex.wl, Part 1.C.1,
```

```
109  %%  these commands define the syntax for theorema symbols in the LaTeX output. The naming
         convention is
110  %%  to use the symbol name in the Theorema code, but without dollar signs or context path.
111  %%
112  %% So for example, the symbol
113  %%  TheoremaAnd$TM (with full context) becomes AndTM, and so on.
114  %% −−−
115
116  % Logical Operations
117  \newcommand{\IffTM}[2]{\left(#1 \iff #2\right)}
118  \newcommand{\AndTM}[2]{\left(#1 \land #2\right)}
119  \newcommand{\ImpliesTM}[2]{\left(#1 \rightarrow #2\right)}
120  \newcommand{\OrTM}[2]{\left(#1 \lor #2\right)}
121
122  % Quantifiers
123  \newcommand{\ForallTM}[2]{\forall #1 \, #2}
124  \newcommand{\ExistsTM}[2]{\exists #1 \, #2}
125
126  % Variables, Ranges, and Predicates
127  \newcommand{\RNGTM}[1]{#1}
128  \newcommand{\SIMPRNGTM}[1]{#1}
129  \newcommand{\STEPRNGTM}[1]{#1}
130  \newcommand{\VARTM}[1]{#1}
131
132  % Definitions
133  \newcommand{\IffDefTM}[2]{\left(#1 :\iff #2\right)}
134  \newcommand{\EqualDefTM}[2]{\left(#1 := #2\right)}
135
136  % Annotations and Subscripts
137  \newcommand{\AnnotatedTM}[3]{#1_{#2}\left(#3\right)}
138  \newcommand{\SubscriptTM}[2]{#1_{#2}}
139
140  % Operations and Relations
141  \newcommand{\LessTM}[2]{#1 < #2}
142  \newcommand{\EqualTM}[2]{#1 = #2}
143  \newcommand{\SubsetEqualTM}[2]{#1 \subseteq #2}
144
145  % Domain−specific Operations
146  \newcommand{\DomainOperationTM}[4]{#1_{#2}\left(#3, #4\right)}
147  \newcommand{\TupleTM}[2]{\left(#1, #2\right)}
148  \newcommand{\TupleOfTM}[2]{#1_{#2}}
149  \newcommand{\IntegerIntervalTM}[2]{[#1, #2]}
150
151  % Specific Mathematical Notations
152  \newcommand{\Mon}[1]{\text{Mon}\left[#1\right]}
153  \newcommand{\TimesTM}{\times}
154  \newcommand{\PlusTM}{+}
155
156  %% −−−
157  %% Theorema Symbols are defined as custom LaTeX commands here! (END)
158  %% −−−
159
160  % From Wolfram template for LaTeX
161  \newcommand{\mathsym}[1]{{}}
162  \newcommand{\unicode}[1]{{}}
163
164  \newcounter{mathematicapage}
```

```
165  \begin{document}
166
167  % \input{}
168
169  \title{Theorema 2.0: A First Tour}
170  \author{}
171  \date{}
172  \maketitle
173
174  % Insert legend at the Parsing Entry point, if DocumentProcessingLevel is so specified
175  \begingroup \section*{} We consider proving ,   computing  , and    solving   as the
          three basic mathematical activities.\endgroup
176
177  \section{Proving}
178
179  \begingroup \section*{} We want to prove\endgroup
180
181  \begin{center}(\underset{x}{\forall}(P[x] \lor Q[x])) \land (\underset{y}{\forall}(P
          [y] \Rightarrow Q[y])) \Leftrightarrow (\underset{x}{\forall}Q[x]) .\end{center}
182  \begingroup \section*{} To prove a formula like the above, we need to enter it in
          the context of a Theorema environment.\endgroup
183
184  \EnvironmentWithFormat{proposition}{First Test, 2014}
185  \IffTM{ \AndTM{ \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{x}}}}{ \OrTM{ P[ \VARTM{x}]}{
          Q[ \VARTM{x}]}}}{ \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{y}}}}{ \ImpliesTM{ P[ \
          VARTM{y}]}{ Q[ \VARTM{y}]}}}}{ \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{x}}}}{ Q[ \
          VARTM{x}]}}\n\n\end{EnvironmentWithFormat}
186
187  \section{Computing}
188
189  \EnvironmentWithFormat{definition}{Lexical Ordering}
190  \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{a}}}{ \SIMPRNGTM{ \VARTM{b}}}}{ \IffDefTM{ \
          AnnotatedTM{LessTM}{ \SubScriptTM{lex}}{ \VARTM{a}}{ \VARTM{b}}}{ \ExistsTM{ \
          RNGTM{ \STEPRNGTM}}{ \AndTM{ \LessTM{ \SubscriptTM{ \VARTM{a}}{ \VARTM{i}}}{ \
          SubscriptTM{ \VARTM{b}}{ \VARTM{i}}}}{ \ForallTM{ \RNGTM{ \STEPRNGTM}}{ \EqualTM
          { \SubscriptTM{ \VARTM{a}}{ \VARTM{j}}}{ \SubscriptTM{ \VARTM{b}}{ \VARTM{j
          }}}}}}}}\n\n\end{EnvironmentWithFormat}
191
192  \EnvironmentWithFormat{definition}{Monomials}
193  \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{K}}}{ \SIMPRNGTM{ \VARTM{m2}}}}{ \EqualDefTM{
           \DomainOperationTM{ Mon[ \VARTM{K}]}{TimesTM}{ \VARTM{m1}}{ \VARTM{m2}}}{ \
          TupleTM{ \DomainOperationTM{ \VARTM{K}}{TimesTM}{ \SubscriptTM{ \VARTM{m1}}{1}}{
           \SubscriptTM{ \VARTM{m2}}{1}}}{ \TupleOfTM{ \RNGTM{ \STEPRNGTM}}{ \
          DomainOperationTM{ \IntegerIntervalTM}{PlusTM}{ \SubscriptTM{ \SubscriptTM{ \
          VARTM{m1}}{2}}{ \VARTM{i}}}{ \SubscriptTM{ \SubscriptTM{ \VARTM{m2}}{2}}{ \VARTM
          {i}}}}}}}}\n\n\end{EnvironmentWithFormat}
194
195  \section{Set Theory}
196
197  \EnvironmentWithFormat{definition}{subset}
198  \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{x}}}{ \SIMPRNGTM{ \VARTM{y}}}}{ \EqualDefTM{
          \SubsetEqualTM{ \VARTM{x}}{ \VARTM{y}}}{ \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{z
          }}}}{ \ImpliesTM{ \ElementTM{ \VARTM{z}}{ \VARTM{x}}}{ \ElementTM{ \VARTM{z}}{ \
          VARTM{y}}}}}}\n\n\end{EnvironmentWithFormat}
199
200  \EnvironmentWithFormat{proposition}{transitivity of }
201  \ForallTM{ \RNGTM{ \SIMPRNGTM{ \VARTM{a}}}{ \SIMPRNGTM{ \VARTM{c}}}}{ \ImpliesTM{ \
```

```
        AndTM{ \SubsetEqualTM{ \VARTM{a}}{ \VARTM{b}}}{ \SubsetEqualTM{ \VARTM{b}}{ \
        VARTM{c}}}}{ \SubsetEqualTM{ \VARTM{a}}{ \VARTM{c}}}}\n\n\end{
        EnvironmentWithFormat}
202
203
204
205 \end{document}
```

See Appendix C for the PDF-rendering of this LaTeX-Code.

Appendix B

Supplementary Materials (Submission Repository)

List of supplementary data submitted to the degree-granting institution for archival storage (in ZIP format) including the FirstTour example used throughout this work as final PDF rendering of the LaTeX-transformed Wolfram Language Notebook (see Appendix A).

These documents are included in the final submission repository.

B.1   PDF Files

Path: /

    main.pdf . . . .   Bachelor thesis (complete document)

Path: /tma2tex

    FirstTour.pdf . . . . . .   Sample Transformation Document FirstTour (included in thesis document as well)

B.2   Program Files

Path: /tma2tex and /tma2tex/res

    *.wl, *.m . . . .   Wolfram Language Code presented in this work and printed in Appendix A
    *.nb . . . .   Wolfram Language Code designed to be displayed as an interactive notebook in Wolfram Mathematica or an online software version: FirstTour.nb as the sample document tested with this project is the most relevant and printed in Appendix A, but includes notebook presentation information that makes it cumbersome to read. Further .nb-files are in the repository and intended to be displayed in Mathematica.

*.tex . . . . Template, intermediary and final .tex files presented in this work and printed in Appendix A

Path: /tma2tex/test

*.mt . . . . Wolfram Language Testing Framework files

Path: /tma2tex/.settings

*.prefs . . . . Wolfram Workbench/Eclipse settings file used for this project

Appendix C

Sample Document FirstTour

The main sample document this project is tested with is included in its final LaTeX-transformed and PDF-rendered output format: the .tex-file this document is rendered from is printed in Appendix A and the original Wolfram Language notebook is included in the submission (and online) repository detailed in Appendix B.

Theorema 2.0: A First Tour

We consider "proving", "computing", and "solving" as the three basic mathematical activities.

1 Proving

We want to prove

$$(\forall_x (P[x] \vee Q[x])) \wedge (\forall_y (P[y] \Rightarrow Q[y])) \Leftrightarrow (\forall_x Q[x]).$$

To prove a formula like the above, we need to enter it in the context of a Theorema environment.

Proposition : First Test, 2014
$$((\forall x \ (P[x] \vee Q[x]) \wedge \forall y \ (P[y] \rightarrow Q[y])) \iff \forall x \ Q[x])$$

2 Computing

Definition : Lexical Ordering
$$\forall ab \ (LessTM_{lex} (a) \, b : \iff \exists \ (a_i < b_i \wedge \forall a_j = b_j))$$
Definition : Monomials
$$\forall Km2 \ \big(Mon[K]_{TimesTM} (m1, m2) := \big(K_{TimesTM} (m1_1, m2_1) \,_{,[\,, PlusTM](m1_{2i}, m2_{2i})} \big)\big)$$

3 Set Theory

Definition : subset
$$\forall xy \ (x \subseteq y := \forall z \ (zx \rightarrow zy))$$
Proposition : transitivity of
$$\forall ac \ ((a \subseteq b \wedge b \subseteq c) \rightarrow a \subseteq c)$$

Appendix D

Mathematica and Wolfram Language Today

D.0.1 Mathematica vs Wolfram Language (vs Wolfram|Alpha)

This disambiguation should be helpful for anyone new to the Wolfram ecosystem or "tech stack," as it is currently marketed: [59]

- Mathematica: the Desktop application, first introduced in 1988 and available for download in Version 14.1 currently. It is a proprietary technology available at a subscription cost. [60]
- Wolfram Language: Frequently described as a "symbolic language," it is also the language that the Mathematica kernel is developed in and runs on. WL can be executed inside Mathematica. A Mathematica package typically has the file ending ".wl" (previously ".m") and can be called from inside Mathematica (a Mathematica notebook). It is also in principle closed source [59] and available for licensing.
- Wolfram|Alpha: publicly available at no cost in the base version [78] this product is sometimes conflated with Mathematica or Wolfram Language due to its public profile. "Wolfram|Alpha's long-term goal is to make all systematic knowledge immediately computable and accessible to everyone." [62]

D.0.2 The Wolfram Tool Chain Today and Its Criticisms

Mathematica, first appearing in 1988, is available in Version 14.1 at the time of writing and is being actively developed by Wolfram Research, with "new and improved" features (since Version 13.3) spanning topical categories like Mathematical Computation, Machine Learning and Neural Networks, High-Dimensional Visualization and Astronomy in addition to Core Language, Importing and Exporting and similar base categories. [58]] Mathematica, the Desktop application, continues to be Wolfram Research's core product, being marketed as the "world's definitive system for modern technical computing" [60] and is distinguished from underlying technologies (Wolfram Language, Wolfram Cloud, Wolfram Knowledgebase, to name a few out of a longer list [60]) and contrasts with the more unified platform approach of Wolfram|One [79], the publicly available Wolfram|Alpha [78], a set of mobile apps [59] and further, more dedicated, products and services.

The size of the program and progress in development is typically measured in number of "in-built functions," that is, functions providing specific functionality, many levels of abstraction higher than the data structure and algorithm oriented functions provided by conventional languages and frameworks: see the following section D.0.3 for the exploration of this idea. The current count of in-built functions, 6602 for the latest major release 14.0 [66] and, for the newly released (minor) Version 14.1, up 89 to a new total of 6691 [77], with the trajectory since Version 1.0 in 1988 given in Figure D.1.
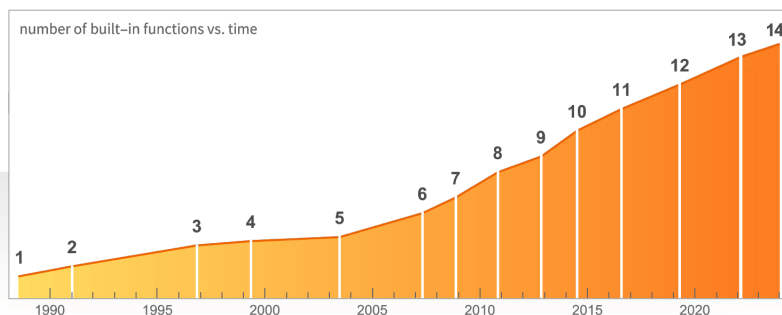

Figure D.1: In Mathematica Desktop, there are now over 6000 "built-in functions." [60]

Criticism of the Wolfram products typically hinge on its closed-source, for-cost nature, though can also be extended to performance and trust in/vendor-lock-in with the company maintaining it, see for example [54]. Specifically the former has been addressed as matters of philosophy when it comes to developing a programming language and associated ecosystem: "The simple answer is that large-scale, unified design requires centralized control and sustained effort that we feel is less achievable with free and open-source software" [59] is the official customer facing answer in this proprietary context. A more extensive statement is given in [55], outlining 12 reasons for being closed-source, proprietary and at-cost, similar but slightly different aspects of this question.

Since this chapter started with Bruno Buchberger, I want to bring this particular question to his view, also mentioned in [7]: the professional development and marketing of Mathematica "is a feature that may have disadvantages for the research community because the code of the kernel of professional systems is normally not open for the user. On the other hand, it also provides some definite advantages as, for example, professional maintenance, high performance [!], professional software production tools, and - in the case of Mathematica - a fantastic front end." (p. 2) Buchberger was writing to Version 3.0 of Mathematica (in 1996) and is of course speaking of performance as it relates to efficient algorithms implemented in C that the kernel relies on, especially complex calculations "implementing the presently best mathematical methods in various fields" (p. 2), rather than performance as it pertains to the system and Mathematica kernel evaluations as a whole and how these might be compared against fully native C or other, lower level programming languages.

D.0.3 How Wolfram Research Views Mathematica/Wolfram Language: The Computational Language Idea

In addressing the question, "What Kind of a Thing Is the Wolfram Language?" [53], Stephen Wolfram, in some position to answer this, as the founder and current CEO of Wolfram Research and originator of the language, espouses the idea of a "computational" language, 'a way to apply the computational paradigm directly to almost anything: we have a language and a notation for doing computational X, for basically any field "X" (from archaeology to zoology, and beyond).' [53] Such a multi-purpose tool is differentiated from conventional programming languages in the following way, relevant to the present work.

> First and foremost, it's that a computational language tries to intrinsically be able to talk about whatever one might think about in a computational way—while a programming language is set up to intrinsically talk only about things one can directly program a computer to do. So for example, a computational language can intrinsically talk about things in the real world—like the planet Mars or New York City or a chocolate chip cookie. A programming language can intrinsically talk only about abstract data structures in a computer. [53]

Mars [40] and New York City [11] are examples of entities that can be addressed in WL using the so-called Wolfram Knowledgebase [59], which also powers Wolfram Alpha [13], another Wolfram Research product. The idea of the computational language turns on this easy access to data, as well as pre-built, high-level functions ("while the core of a standard programming language typically has perhaps a few tens of primitive functions built in, the Wolfram Language has more than 5600" at the time of writing [53], in 2019 - now 6602 in Mathematica and WL Version 14.1 [51] in 2024, five years on) operating on the WL expression structure.

This latter and the symbolic notion in an extended sense are seen as key in this presentation; to conclude it, and to give the intuition for the symbolic expressions and how they relate to the central pattern matching approach in Wolfram Language programming for processing expressions, not too different from Regular Expressions matching for strings (of text characters) in more conventional programming contexts:

> In most standard programming languages, x on its own without a value doesn't mean anything; it has to stand for some structure in the memory of the computer. But in a computational language, one's got to be able to have things that are purely symbolic, and that represent, for example, entities in the real world—that one can operate on just like any other kind of data.[53]

For the purposes of this work, in document transformation, where it has already been established that the document in question, a Mathematica notebook, is also such a symbolic expression, the idea simply means that any document following the (symbolic, that is expressions-)structure of a Theorema notebook can be processed using the Tma2TeX package: the symbolic in a symbolic expression is to mean something like a class of object, where their particular expression structure ("A foundational idea in the

Wolfram Language is that all expressions—whatever they may represent—ultimately have a uniform tree-like structure," [16]) including the individual expressions (their so-called expression "heads," [25]) match at the relevant level of abstraction. This level will be the defining mechanism that makes out the pattern matching approach, explored in the Theory chapter (2) in this work, Section 2.3.2.

Appendix E

TeXForm – Existing (Kernel) Functionality

E.0.1 A Source Code Deep-Dive

The kernel sits at the core of the WL system (in keeping with the term in operating systems) and is available as a stand-alone program as well: "A standalone kernel session normally reads input from a device (typically a keyboard or a WSTP [76] connection), evaluates the expression and prints the result to a device (typically a screen or a WSTP connection)." [74] Therefor the kernel needs to contain the main functionality expected for the WL system (independently of the notebook environment). From a software development standpoint, the kernel repository is the place to find the implementation for any essential function that a package outside of the kernel might rely on.

For this project, the main kernel function of relevance is TeXForm, which takes a WL expression and transforms it to its typesetting correlate in LaTeX. Looking directly at the implementation reveals that the expedient way to handle this transformation is to exhaustively list and maintain the transformation rules in the form of a WL association. In order, the following categories (with one example each) make up most of the currently about 2000 lines of code inside the TeXForm package.

Variable	Category of Symbol	Replacement Rule Example
`$GreekLetters`	Greek Letters	`"\[Alpha]" -> "\\alpha "`
`$CaligraphicLetters`	Caligraphy	`"\[ScriptA]" -> "\\mathit{a}"`
`$GothicLetters`	Gothic	`"\[GothicA]" -> "\\mathfrak{a}"`
`$DoubleStruckLetters`	Double-struck/Emphasis	`"\[DoubleStruckA]" -> "\\mathbf{a}"`
`$AccentedLetters`	Accented Letters	`"\[AGrave]" -> "\\text{\\ a}"`
`$MiscellaneousSymbols`	Miscellaneous Symbols	`"\[ConstantC]" -> "c"`
`$Shapes`	Shapes	`"\[FilledSquare]" -> "\\blacksquare"`
`$TextualForms`	Special Characters	`"\[DotlessI]" -> "\\text{\\i}"`
`$Operators`	Operators	`"\[Times]" -> "\\times"`
`$RelationSymbols`	Symbols for Various Relations	`"\[NotEqual]" -> "\\neq"`
`$Arrows`	Arrow-Symbols	`"\[LeftArrow]" -> "\\leftarrow"`
`$Spaces`	Space-Characters	`"\[ThickSpace]" -> "\\thickspace"`
`$Others`	Other (Mathematical) Symbols	`"\[Conjugate]" -> "*"`
`$LeftTeXDelimiterReplacements`	Left Delimiters	`"(" -> {"("}`
`$RightTeXDelimiterReplacements`	Right Delimiters	`")" -> {")"}`
`$TeXDelimiterReplacements`	LaTeX-Delimiters	`"\\" -> {"\\backslash "}`
`$BasicEscapes`	Escaped Characters	`"#" -> "\\#"`
`$ASCIIUnchanged`	ASCII characters	Function

This tabulation gives an idea of the type of transformations that would need to occur and implies for this project: while a mapping of a Theorema Language to WL expressions is possible in some cases (in fact a straightforward task in cases where the transformation is removing the suffix `$TM` and if needed, the full context with Theorema reference), any Theorema-to-LaTeXtransformation would be limited to symbols available in WL, negating the need and use of Theorema Language. Specific as-is output using notebook-level LaTeX-transformation via file-export was already shown to be lacking in section 1.2.2.

Further, the pattern-matching-based `parseNbContent` allows for exactly the parts of the notebook of interest to the relevant user group, making the overall idea for the project a bespoke Theorema-TeXForm at notebook-level, in terms of existing WL-functionality, sitting neatly somewhere between a solution that solves expression-level TeX-transformation and notebook-level export.

As far as WL-TeXForm is concerned, all replacements are grouped together in an exhaustive list of replacements:

```
$TeXReplacements = Join[
  $ASCIIUnchanged, $BasicEscapes, $TeXDelimiterReplacements,
  $GreekLetters, (*$GreekWords,*) $AccentedLetters, $Spaces,
  $CaligraphicLetters, $GothicLetters, $DoubleStruckLetters,
  $MiscellaneousSymbols, $Shapes, $TextualForms,
  $Operators, $RelationSymbols, $Arrows, $Others
]
```

These `TeXReplacements` inform the actual parsing in TeXForm. As has been argued already, TeXForm needs to be expanded in some way to permit Theorema-language

parsing. Now a problem arises, however, because it is not as straightforward as a simple case-by-case switching between TeXForm and TheoremaTeXForm, say.

The (closed) source code for the Kernel would need to be opened to intertwine Theorema-specific functionality with TeXForm-functionality. This is illustrated by the fact that TexForm needs the complete expression to operate on successfully (maintaining correct nesting) and would not be useful called on an independent part (at any level in the expression hierarchy) individually: so `TeXForm[And[Or[a,b], Or[c,d]]]` would yield `(a\\lor b)\\land (c\\lor d)` natively, but called on the Theorema-language correlate at the outer level, that is `ExpressionToTeX[And$TM[Or[a, b], Or[c, d]]]`, gives `\\text{And$TM }(a\\lor b,c\\lor d)`: The text-rendering of unrecognized symbols is a sensible default. Calling a Theorema-version of TeXForm on only the And$TM-expression, however, is problematic, because the TeXForm, as implemented in texformdump.wl and provided in the project repo, is fundamentally recursive to handle nested expressions. Without access to the way the operative recursive function (MakeTeX as it turns out) is called, a native-only TeXForm-solution would not work to handle Theorema language expressions in the proposed method.

The MakeTeX/makeTeX distinction is significant because it realizes the separation between the caller package and `Texformdump\``, the package. makeTeX is the driver of the recursion-parsing, but calls MakeTeX in turn, which is only defined exactly once inside this base package, suggesting its usage to the user, the caller package:

```
(* Only built-in rule *)
MakeTeX[boxes_] := maketex[boxes]
```

Any overloading of MakeTeX overrides all corresponding maketex rules, so while it is important that maketex recursively calls MakeTeX, not maketex, maketex is doing the heavy lifting, and MakeTeX allows the user to override definitions at any expression level. MakeTeX-rules can also be defined inside the caller package in this way and it is by this mechanism that this project realizes the TeXForm-customization, building functionality into the natively used transformation rules and defaulting to this as needed, in the case of no special Theorema transformation rules targeting the corresponding LaTeX.

The process hinges on the functions MapTeX, the list-input version of MakeTeX -

```
MapTeX[stuff_List] := Map[MakeTeX, stuff]
MapTeX[stuff___] := MapTeX[{stuff}]
```

- as well as some auxiliary logic informing the central set of makeTeX functions making up the last quarter of the TeXForm implementation. The first makeTeX, for example, is:

```
maketex[RowBox[{l___, lb:DelimiterPattern, mid___, rb:DelimiterPattern, r___}]] :=
Module[{delimQ},
  DebugPrint["----------------------------------"];
  DebugPrint["maketex[RowBox[{l___, lb:DelimiterPattern,
    mid___, rb:DelimiterPattern, r___}]]"];
  DebugPrint["l: ", l];
  DebugPrint["lb: ", lb];
```

```
  DebugPrint["mid: ", mid];
  DebugPrint["rb: ", rb];
  DebugPrint["r: ", r];
  delimQ = DelimiterBoxQ[mid];
  StringJoin[
    MapTeX[l],
    If[delimQ, InsertDelimiters["left", lb], MakeTeX[lb]],
    MapTeX[mid],
    If[delimQ, InsertDelimiters["right", rb], MakeTeX[rb]],
    MapTeX[r]
 ]
]
```

References

Literature

[1]   *$Failed—Wolfram Language Documentation*. URL: https://reference.wolfram.com /language/ref/$Failed.html (visited on 02/28/2024) (cit. on p. 35).

[2]   *American Mathematical Society*. en. URL: https://www.ams.org/arc/resources/am slatex-about.html (visited on 04/02/2024) (cit. on p. 5).

[3]   *Begin—Wolfram Language Documentation*. URL: https://reference.wolfram.com/l anguage/ref/Begin.html (visited on 04/09/2024) (cit. on p. 12).

[4]   *BlankNullSequence—Wolfram Language Documentation*. URL: https://reference.w olfram.com/language/ref/BlankNullSequence.html (visited on 02/24/2024) (cit. on p. 2).

[5]   *Block—Wolfram Language Documentation*. URL: https://reference.wolframcloud.c om/language/ref/Block.html (visited on 02/24/2024) (cit. on p. 26).

[6]   *BoxData—Wolfram Language Documentation*. URL: https://reference.wolfram.co m/language/ref/BoxData.html#:~:text=BoxData%20is%20typically%20used%20as ,need%20to%20be%20used%20directly. (visited on 04/02/2024) (cit. on p. 4).

[7]   Bruno Buchberger. "Mathematica as a Rewrite Language" (Nov. 1996) (cit. on pp. 2, 3, 64).

[8]   *Building Large Software Systems in the Wolfram Language—Wolfram Language Documentation*. URL: https://reference.wolfram.com/language/tutorial/BuildingL argeSoftwareSystemsInTheWolframLanguage.html (visited on 04/09/2024) (cit. on pp. 10, 12).

[9]   *Cell > Show Expression—Wolfram Language Documentation*. URL: https://ref erence.wolfram.com/language/ref/menuitem/ShowExpression.html (visited on 04/02/2024) (cit. on p. 4).

[10] *Cell—Wolfram Language Documentation*. URL: https://reference.wolfram.com/la nguage/ref/Cell.html (visited on 04/02/2024) (cit. on p. 4).

[11] *City—Wolfram Language Documentation*. URL: https://reference.wolfram.com/la nguage/ref/entity/City.html (visited on 04/02/2024) (cit. on p. 65).

[12] *CompoundExpression—Wolfram Language Documentation*. URL: https://refer ence.wolframcloud.com/language/ref/CompoundExpression.html (visited on 02/24/2024) (cit. on pp. 3, 26).

[13]   *Curated Computational Data in the Wolfram Knowledgebase*. en. URL: https://w
       ww.wolframalpha.com (visited on 04/02/2024) (cit. on p. 65).

[14]   *End—Wolfram Language Documentation*. URL: https://reference.wolfram.com/la
       nguage/ref/End.html (visited on 04/09/2024) (cit. on p. 12).

[15]   *EndPackage—Wolfram Language Documentation*. URL: https://reference.wolfram
       .com/language/ref/EndPackage.html (visited on 04/09/2024) (cit. on p. 12).

[16]   *Expression Structure—Wolfram Language Documentation*. URL: https://referenc
       e.wolfram.com/language/guide/ExpressionStructure.html (visited on 04/02/2024)
       (cit. on p. 66).

[17]   *Expressions—Wolfram Language Documentation*. URL: https://reference.wolfra
       m.com/language/tutorial/Expressions.html#:~:text=Expressions%20in%20the
       %20Wolfram%20System,6%2D1%5D%20performs%20factorization. (visited on
       04/02/2024) (cit. on pp. 2, 3).

[18]   *Failure—Wolfram Language Documentation*. URL: https://reference.wolfram.com
       /language/ref/Failure.html (visited on 02/28/2024) (cit. on pp. 35, 36).

[19]   *Find the Underlying Box Structure of a Formatted Expression—Wolfram Language
       Documentation*. URL: https://reference.wolfram.com/language/workflow/FindThe
       UnderlyingBoxStructureOfAFormattedExpression.html (visited on 04/02/2024) (cit.
       on p. 4).

[20]   *FindEquationalProof—Wolfram Language Documentation*. URL: https://referenc
       e.wolfram.com/language/ref/FindEquationalProof.html (visited on 04/05/2024)
       (cit. on p. 8).

[21]   *Functional Operations—Wolfram Language Documentation*. URL: https://refe
       rence.wolfram.com/language/tutorial/FunctionalOperations.html (visited on
       04/04/2024) (cit. on p. 13).

[22]   G. Mayrhofer, S. Saminger % W. Winsteiger. "Theorema". In: *The Seventeen
       Provers of the World*. Springer. URL: https://www.cs.ru.nl/~freek/comparison/co
       mparison.pdf (visited on 04/05/2024) (cit. on pp. 7, 8).

[23]   *Get—Wolfram Language Documentation*. URL: https://reference.wolfram.com/lan
       guage/ref/Get.html (visited on 04/09/2024) (cit. on p. 12).

[24]   *GitHub - windsteiger/Theorema: Theorema: A System for Automated Reasoning
       (Theorem Proving) and Automated Theory Exploration based on Mathematica*.
       URL: https://github.com/windsteiger/Theorema (visited on 04/02/2024) (cit. on
       p. 7).

[25]   *Head—Wolfram Language Documentation*. URL: https://reference.wolfram.com/la
       nguage/ref/Head.html (visited on 04/02/2024) (cit. on p. 66).

[26]   *https://www3.risc.jku.at/research/theorema/software/*. URL: https://www3.risc.jk
       u.at/research/theorema/software/ (visited on 04/05/2024) (cit. on p. 7).

[27]   *IntelliJ IDEA – the Leading Java and Kotlin IDE*. en. URL: https://www.jetbrain
       s.com/idea/promo/ (visited on 02/24/2024) (cit. on p. 5).

[28]   *Introduction to Dynamic—Wolfram Language Documentation.* URL: https://ref
       erence.wolfram.com/language/tutorial/IntroductionToDynamic.html (visited on
       04/05/2024) (cit. on pp. 4, 8).

[29]   *LaTeX Project Team.* URL: https://www.latex-project.org/about/team/ (visited
       on 04/02/2024) (cit. on p. 4).

[30]   *LaTeX Project Team.* URL: https://www.latex-project.org/about/team/#frank-mi
       ttelbach (visited on 04/02/2024) (cit. on p. 4).

[31]   *MakeBoxes—Wolfram Language Documentation.* URL: https://reference.wolfram
       .com/language/ref/MakeBoxes.html (visited on 07/20/2024) (cit. on p. 21).

[32]   Mircea Marin and Florina Piroi. "Rule-Based Programming with Mathematica".
       en () (cit. on pp. 13, 14).

[33]   *Message—Wolfram Language Documentation.* URL: https://reference.wolfram.co
       m/language/ref/Message.html (visited on 02/24/2024) (cit. on p. 36).

[34]   Misc. *Quaternion.* en. Page Version ID: 1235436136. July 2024. URL: https://e
       n.wikipedia.org/w/index.php?title=Quaternion&oldid=1235436136 (visited on
       08/26/2024) (cit. on p. 3).

[35]   *Module: Create a scoping construct for local variables—Wolfram Documentation.*
       URL: https://reference.wolframcloud.com/language/ref/Module.html (visited on
       02/24/2024) (cit. on p. 26).

[36]   *Namespace Management—Wolfram Language Documentation.* URL: https://ref
       erence.wolfram.com/language/guide/NamespaceManagement.html (visited on
       04/09/2024) (cit. on p. 12).

[37]   *Needs—Wolfram Language Documentation.* URL: https://reference.wolfram.com/l
       anguage/ref/Needs.html (visited on 04/09/2024) (cit. on p. 12).

[38]   *Notebook—Wolfram Language Documentation.* URL: https://reference.wolfram.co
       m/language/ref/Notebook.html (visited on 04/02/2024) (cit. on p. 4).

[39]   *Package Development—Wolfram Language Documentation.* URL: https://referenc
       e.wolfram.com/language/guide/PackageDevelopment.html (visited on 04/09/2024)
       (cit. on p. 12).

[40]   *Planet—Wolfram Language Documentation.* URL: https://reference.wolfram.com
       /language/ref/entity/Planet.html (visited on 04/02/2024) (cit. on p. 65).

[41]   *Procedural Programming—Wolfram Language Documentation.* URL: https://ref
       erence.wolfram.com/language/guide/ProceduralProgramming.html (visited on
       04/09/2024) (cit. on p. 12).

[42]   *ProofObject—Wolfram Language Documentation.* URL: https://reference.wolfram
       .com/language/ref/ProofObject.html (visited on 04/05/2024) (cit. on p. 8).

[43]   *Scoping Constructs—Wolfram Language Documentation.* URL: https://reference
       .wolfram.com/language/guide/ScopingConstructs.html (visited on 04/09/2024)
       (cit. on p. 12).

[44]   *Software.* en-US. URL: https://risc.jku.at/software/ (visited on 04/03/2024) (cit.
       on p. 1).

[45] *Stylesheets—Wolfram Language Documentation*. URL: https://reference.wolfram.c om/language/guide/Stylesheets.html (visited on 04/05/2024) (cit. on p. 8).

[46] *Suppress the Output of a Computation—Wolfram Language Documentation*. URL: https://reference.wolfram.com/language/howto/SuppressTheOutputOfAComputati on.html (visited on 04/03/2024) (cit. on p. 3).

[47] *TestObject—Wolfram Language Documentation*. Sept. 2024. URL: https://refere nce.wolfram.com/language/ref/TestObject.html (visited on 09/21/2024) (cit. on p. 37).

[48] *The GNU General Public License v3.0 - GNU Project - Free Software Foundation*. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 04/10/2024) (cit. on p. 12).

[49] *The GNU Operating System and the Free Software Movement*. URL: https://www .gnu.org/home.en.html (visited on 04/10/2024) (cit. on p. 12).

[50] *The Internals of the Wolfram System—Wolfram Language Documentation*. URL: https://reference.wolfram.com/language/tutorial/TheInternalsOfTheWolframSyste m.html#28134 (visited on 04/09/2024) (cit. on p. 10).

[51] *The Story Continues: Announcing Version 14 of Wolfram Language and Mathematica*. en. Jan. 2024. URL: https://writings.stephenwolfram.com/2024/01/the -story-continues-announcing-version-14-of-wolfram-language-and-mathematica/ (visited on 04/05/2024) (cit. on p. 65).

[52] *Theorema/Theorema/PackageTemplate.m at master · windsteiger/Theorema*. URL: https://github.com/windsteiger/Theorema/blob/master/Theorema/Packa geTemplate.m (visited on 04/05/2024) (cit. on p. 12).

[53] *What We've Built Is a Computational Language (and That's Very Important!)—Stephen Wolfram Writings*. en. May 2019. URL: https://writings.stephe nwolfram.com/2019/05/what-weve-built-is-a-computational-language-and-thats-ve ry-important/ (visited on 02/24/2024) (cit. on p. 65).

[54] *Why isn't Wolfram more popular?* Reddit Post. Oct. 2022. URL: www.reddit.c om/r/math/comments/ye2vj0/why_isnt_wolfram_more_popular/ (visited on 04/02/2024) (cit. on p. 64).

[55] *Why Wolfram Tech Isn't Open Source—A Dozen Reasons—Wolfram Blog*. en. Apr. 2019. URL: https://blog.wolfram.com/2019/04/02/why-wolfram-tech-isnt-op en-source-a-dozen-reasons/ (visited on 04/02/2024) (cit. on p. 64).

[56] Wolfgang Windsteiger. "Theorema 2.0: A Brief Tutorial". en. In: *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. Timisoara: IEEE, Sept. 2017, pp. 36–38. DOI: 10.1109/SYNASC .2017.00016. (Visited on 04/05/2024) (cit. on p. 7).

[57] Wolfgang Windsteiger. "Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System". *Electronic Proceedings in Theoretical Computer Science* 118 (July 2013). arXiv:1307.1945 [cs], pp. 72–82. DOI: 10.4204/EPTCS.118.5 . (Visited on 10/14/2023) (cit. on pp. 6, 8–10).

[58] Wolfram Research. *Export: Output data to a specified file format—Wolfram Documentation.* URL: https://reference.wolfram.com/language/ref/Export.html (visited on 10/14/2023) (cit. on p. 63).

[59] *Wolfram Research, Inc.* URL: https://reference.wolfram.com/legacy/v3/MainBook /2.5.1.html (visited on 04/03/2024) (cit. on pp. 2–5, 63–65).

[60] *Wolfram Research, Inc.* URL: https://reference.wolfram.com/legacy/v3/MainBook /2.5.11.html (visited on 04/03/2024) (cit. on pp. 5, 63, 64).

[61] Wolfram Research, Inc. *$Context—Wolfram Language Documentation.* Aug. 2024. URL: https://reference.wolfram.com/language/ref/$Context.html (visited on 04/09/2024) (cit. on p. 23).

[62] Wolfram Research, Inc. *About Wolfram|Alpha: Making the World's Knowledge Computable.* en. Aug. 2024. URL: https://www.wolframalpha.com (visited on 08/26/2024) (cit. on p. 63).

[63] Wolfram Research, Inc. *Expressions—Wolfram Language Documentation.* Aug. 2024. URL: https://reference.wolfram.com/language/tutorial/Expressions.html#20 284 (visited on 08/26/2024) (cit. on p. 3).

[64] Wolfram Research, Inc. *MakeBoxes—Wolfram Language Documentation.* URL: https://reference.wolfram.com/language/ref/MakeBoxes.html (visited on 07/19/2024) (cit. on p. 19).

[65] Wolfram Research, Inc. *Message—Wolfram Language Documentation.* Aug. 2024. URL: https://reference.wolfram.com/language/ref/Message.html (visited on 08/24/2024) (cit. on p. 24).

[66] Wolfram Research, Inc. *Summary of New and Improved Features in 14.0—Wolfram Language Documentation.* Aug. 2024. URL: https://reference.wolfram.com/la nguage/guide/SummaryOfNewFeaturesIn140.html (visited on 02/24/2024) (cit. on p. 64).

[67] Wolfram Research, Inc. *TestCreate—Wolfram Language Documentation.* Sept. 2024. URL: https://reference.wolfram.com/language/ref/TestCreate.html (visited on 09/21/2024) (cit. on p. 37).

[68] Wolfram Research, Inc. *TestEvaluate—Wolfram Language Documentation.* Sept. 2024. URL: https://reference.wolfram.com/language/ref/TestEvaluate.html (visited on 09/21/2024) (cit. on p. 37).

[69] Wolfram Research, Inc. *TestReport—Wolfram Language Documentation.* Sept. 2024. URL: https://reference.wolfram.com/language/ref/TestReport.html (visited on 09/21/2024) (cit. on p. 37).

[70] Wolfram Research, Inc. *Textual Input and Output—Wolfram Language Documentation.* Aug. 2024. URL: https://reference.wolfram.com/language/tutorial/TextualI nputAndOutput.html#12413 (visited on 08/24/2024) (cit. on p. 24).

[71] Wolfram Research, Inc. *Transformation Rules and Definitions—Wolfram Language Documentation.* Aug. 2024. URL: https://reference.wolfram.com/languag e/tutorial/TransformationRulesAndDefinitions.html#6972 (visited on 08/26/2024) (cit. on p. 3).

[72] Wolfram Research, Inc. *Using Testing Notebooks—Wolfram Language Documentation.* Sept. 2024. URL: https://reference.wolfram.com/language/tutorial/UsingTestingNotebooks.html (visited on 09/21/2024) (cit. on p. 37).

[73] Wolfram Research, Inc. *Using the Testing Framework—Wolfram Language Documentation.* Sept. 2024. URL: https://reference.wolfram.com/language/tutorial/UsingTheTestingFramework.html (visited on 09/21/2024) (cit. on pp. 37, 38).

[74] Wolfram Research, Inc. *WolframKernel—Wolfram Language Documentation.* July 2024. URL: https://reference.wolfram.com/language/ref/program/WolframKernel.html (visited on 07/10/2024) (cit. on p. 67).

[75] Wolfram Research, Inc. *WolframScript—Wolfram Language Documentation.* Sept. 2024. URL: https://reference.wolfram.com/language/ref/program/wolframscript.html (visited on 09/21/2024) (cit. on p. 38).

[76] Wolfram Research, Inc. *WSTP: Wolfram Symbolic Transfer Protocol.* en. July 2024. URL: https://www.wolfram.com/wstp/ (visited on 07/10/2024) (cit. on p. 67).

[77] Wolfram Research, Inc. *Yet More New Ideas and New Functions: Launching Version 14.1 of Wolfram Language & Mathematica.* en. July 2024. URL: https://writings.stephenwolfram.com/2024/07/yet-more-new-ideas-and-new-functions-launching-version-14-1-of-wolfram-language-mathematica/ (visited on 08/26/2024) (cit. on p. 64).

[78] *Wolfram|Alpha Tour.* en. URL: https://www.wolframalpha.com (visited on 04/02/2024) (cit. on p. 63).

[79] *Wolfram|One: Cloud-Desktop Computation Platform.* en. URL: http://www.wolfram.com/wolfram-one/ (visited on 04/02/2024) (cit. on p. 63).

Check Final Print Size