



Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Theorema Project: Document Processing

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Jack Heseltine, BA

Begutachtet von Assoc. Prof. DI Dr. Wolfgang Windsteiger

Hagenberg, September 2024

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 thesis is identical with the electronic version submitted.

Date

Signature

Contents

Declaration	iii
Preface	vii
Abstract	viii
Kurzfassung	ix
1 Introduction	1
1.1 Mathematica as Both a Tool and an Object of Study, at RISC: The Theorema Project	1
1.1.1 Starting from Rewriting: Mathematica and Wolfram Language as a Programming Language	1
1.1.2 Wolfram Language Expressions, Comparison to Object Oriented Programming	2
1.1.3 Wolfram Language as a Document Representation Format	3
1.2 Motivation	4
1.2.1 Need: The L ^A T _E X-Standard for Academic Publishing in Mathematical Disciplines	4
1.2.2 Comparison to Existing Functionality in Mathematica	5
1.3 Development Environment	7
1.3.1 Tools Used	7
1.3.2 Platform-(In)dependence	7
1.4 Mathematica/Wolfram Language Today	7
1.4.1 Mathematica vs Wolfram Language (vs Wolfram Alpha)	7
1.4.2 The Wolfram Tool Chain Today and Its Criticisms	8
1.4.3 How Wolfram Research Views Mathematica/Wolfram Language: The Computational Language Idea	9
1.4.4 The Connection to First Order Predicate Logic	10
2 Theoretical Background	11
2.1 Theorema	11
2.1.1 Theorema vs Mathematica	11
2.1.2 <i>Theorema 2.0</i> - Theorema Commander, Current Project Structure	12
2.1.3 The Logic of Theorema	13
2.1.4 Theorema Environment and Surfacing the Theorema Language .	14

2.2	Large Systems with Wolfram Language	15
2.2.1	Modularity with Packages	17
2.2.2	Theorema as an Extensible Mathematica Package	17
2.3	Paradigms: The Project-Perspective on the Multi-Paradigm Approach in Wolfram Language	18
2.3.1	Functional vs Procedural Programming in Wolfram Language, and the High-Level Programming Paradigm	18
2.3.2	Symbolic Expressions Lead to Rule-Based Programming and Pattern Matching Approaches	19
2.3.3	Rule-Based Programming	19
2.3.4	Rule-Based Programming vs Rewriting, via Pattern Matching	21
3	Concept	23
3.1	Conceptual Cornerstones for this Project	23
3.1.1	WL-Native Approach for Direct Integration with Theorema	23
3.1.2	Existing (Kernel) Functionality: Source Code Deep-Dive	23
3.1.3	Package/MakeTeX-Specification	27
3.1.4	For This Project: No Layout-Information in the L ^A T _E X	29
3.1.5	MakeBoxes: An Alternative Typesetting-Pipeline	31
3.2	Double Recursive Descent Through Wolfram and Theorema Language Using Pattern Matching and Rule Based Programming	31
3.2.1	Pattern Matching to Realize L ^A T _E X-Transformation of Wolfram Language Notebook Code	32
3.2.2	Pattern Matching to Realize L ^A T _E X-Transformation of the Theorema Language Data Structure	33
3.3	Extensibility in Both L ^A T _E Xand Wolfram Language	33
3.3.1	A Note on Evaluation Criteria and Stability	33
3.3.2	WL-Messages and -Tests: Software Design Goals	33
3.3.3	Extensibility	34
4	Implementation	35
4.1	Overview of the Implementation	35
4.1.1	Note on Modular Programming in Wolfram Language	36
4.1.2	Overall Structure of the Package	36
4.2	High Level Programming in Practice	36
4.2.1	Client Functions	37
4.2.2	File-handling and L ^A T _E XDetai ls	39
4.3	Implementation of (Double) Recursive Descent with Pattern Matching .	40
4.3.1	General Remarks on Pattern Matching, and Execution Order, in Wolfram Language	40
4.3.2	Limited Approach of Specific Pattern Matching Rules	41
4.3.3	Generalized Parsing Approach for Theorema Data	42
5	Closing Words	46
5.1	Messages, Failures, and Testing in WL	46
5.1.1	Working with Messages in WL	47
5.1.2	Testing in the Wolfram Language	48

5.1.3	Testing Approach for this Project	49
5.2	Analysis and Review	51
5.3	Final Closing Remarks: Wolfram Language as a Software Engineering Tool and Integrating with Other Languages and Environments, Potential Future Work	51
5.3.1	Using Wolfram Language for Software Engineering	51
5.3.2	Potential Future Work	52
A	Technical Documentation/Source Code	53
B	Supplementary Materials (Submission Repository)	71
B.1	PDF Files	71
B.2	Program Files	71
C	Sample Document FirstTour	73
D	Exposé: A Tree Pattern Function in Mathematica	75
References		88
Literature		88

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:

- at the Red Cross Blood Bank and Transfusion Center in Upper Austria, at Linz General Hospital: DI Dr. Stephan Federsel and Dr. Norbert Niklas, IT leadership at the blood bank, for making generous allowances for pursuit of the part-time, Friday- (and Saturday-)form of the Software Engineering curriculum in Hagenberg, and for teaching me the value of quality software documentation;
- my advisor Assoc. Univ.-Prof. DI Dr. Wolfgang Windsteiger, but also the course management (Studiengangsleitung) for Software Engineering, for being blind to bureaucratic boundaries between Johannes Kepler University (JKU) and Fachhochschule Oberösterreich (Campus Hagenberg), so that the present thesis work could be situated at RISC, a JKU institute, but explore this Software Engineering topic in the domain of RISC's general interest in Symbolic Computation and using Mathematica in particular, and Prof. Windsteiger for his patience as well;
- the wonderful people in Educational Outreach under Mads Bahrami, PhD, at Wolfram Research, headquartered in Champaign, Illinois, for not just allowing me to participate in the Wolfram Summer School 2023, but also fully funding it, significantly accelerating learning of the Wolfram Language, and subsequently accepting my application for the role of Software Engineer in the Cloud Team at the company. In a full circle journey, I am teaching at Wolfram Summer Research Program 2024, the format for high schoolers, drawing on the understanding gained in the writing of this thesis, as well as the summer school, and looking forward to presenting results and methods deployed here to eager young students -
- - for which, again, I am thankful to Wolfram Research but especially my manager, John Pacey, for allowing this kind of flexibility. The same I would also like to thank for use of learning and time on the job as the work placement requirement in the field of Software Engineering for official completion of the course of study;
- finally I would also like to thank my colleague and team lead, Joel Klein, for the training in using Wolfram Language as an engineering tool and patience, and very often his marked pleasure, in answering my many technical questions.

I am immensely grateful for your trust, patience (once again), and also your commitment to continued learning and quality of your work, which is truly inspiring.

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 L^AT_EX typesetting 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 L^AT_EX typesetting 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 L^AT_EX-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 L^AT_EX-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.” [47]) 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, up to some theory, in rewriting and exploring Theorema particularly.

1.1.1 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” [63], but this disambiguation will be made again in [1.4.1]: 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 that seems to get to the core of what the language does and can do really well. 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 offeres 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]); and now it gets interesting.

Mathematica expressions are defined 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, \dots, T_n are expressions or sequence variables then

$$F[T_1, \dots, T_n]$$

is also an expression. (Such an expression is called a “compound expression” or “application expression.”) [7, p. 4]

For example, integers, or “ $\text{Sin}[\text{Plus}[2, x_{_}]]$ ” 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 $x \sim f \sim y$ [17], and for single-argument expressions, prefix and postfix using symbols, $f @ x$ and $x \backslash\backslash f$ [17] respectively.

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

$$\text{lhsExpr} := \text{rhsExpr}(/; \text{condition})$$

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 [49].) These ideas will be revisited in Chapter 2: the main point here is that the evaluation mechanism in Mathematica [TODO: reference] [TODO: formulare the application of rewrite rules until none are left, compare this mechanism to other programming languages and maybe matlab too, and make the connection to math and the name mathematica]

1.1.2 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`” [63]. 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 [63], 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 [36]).

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

That is, delayed assignments are associated with `quat`-objects (the “gs” of the relevant Wolfram TechDoc [76], 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.” [63], At the end of section 2.4.10, Associating Definitions with Different Symbols]

This becomes yet another interpretation of Wolfram Language expressions [67], The Meaning of Expressions].

1.1.3 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.” [40]

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 [63].

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,” demonstrated in Figures [1.1] and [1.2].

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.

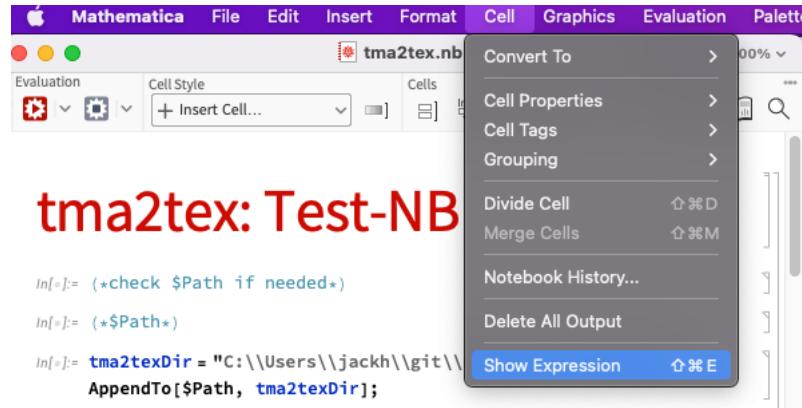


Figure 1.1: In Mathematica Desktop, Cell menu and then “Show Expression” reveals ...



Figure 1.2: ... the lower level expression encoding the typesetting information for what is rendered by the front-end part of the system.

1.2 Motivation

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 [30]: In their words, “ \LaTeX is the de facto standard for the communication and publication of scientific documents.” [31]

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. [30]

The LaTeX Project lists the system's current set of features as follows [29].

- 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 Comparison to Existing Functionality in Mathematica

Mathematica provides native L^AT_EX-export functionality, drawing on AMS-LaTeX, already listed 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] This can be shown by following the steps shown in Figures 1.3 and 1.4 to save a notebook in the L^AT_EX-format in the "Save as ..." menu.

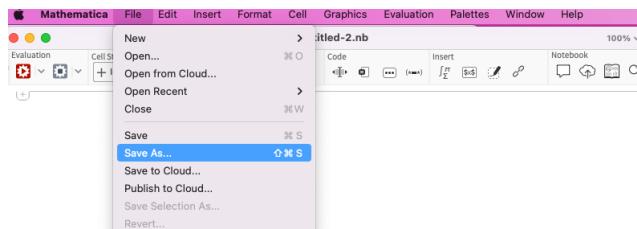


Figure 1.3: In Mathematica Desktop, file menu and then "Save as ..." leads to the option to save any notebook in L^AT_EXformat.

The .TeX-file produced consists of these lines, implementing related package imports and document setup commands.

```
%% AMS-LaTeX Created with the Wolfram Language : www.wolfram.com

\documentclass{article}
\usepackage{amsmath, amssymb, graphics, setspace}
```

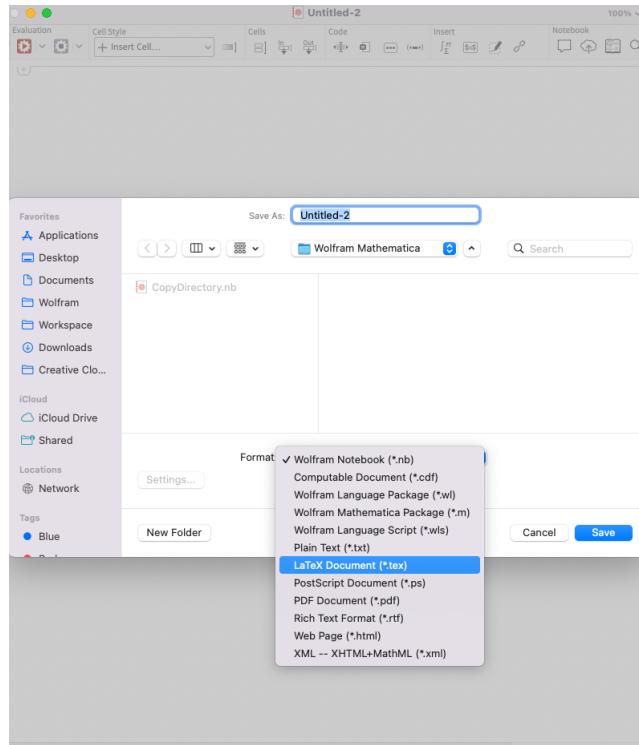


Figure 1.4: There are also other export options next to default Mathematica notebook save option, and L^AT_EX-Format, the option of interest at this point.

```
\newcommand{\mathsym}[1]{{}}
\newcommand{\unicode}[1]{{}{}}

\newcounter{mathematicapage}
\begin{document}

\end{document}
\end{LATEXCode}
\end{program}
```

Crucially for this project, this 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), see Figure ??, the PDF-rendering of the native L^AT_EX-export (based on FirstTourNativeExport.tex, also provided in the project repository).

The WL (in-notebook) function TeXForm handles the cell-level transformation abd will be the basic Kernel-level functionality that is expanded upon to realize the fundamental transformation functionality in this project, discussed in Section 3.1.2.

1.3 Development Environment

1.3.1 Tools Used

In the present work, Mathematica and WolframKernel 14.1 [63] 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 [28] is used in conjunction with the IntelliJ plugin Wolfram Language by Hal's Corner [64] 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.

1.3.2 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 [1.4.1] on the relevant terminology) as well, see section ??: 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-*independent* and benefits from the Wolfram Language ecosystem setup (see criticisms in Section [1.4.2]) 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.” [61] p. 72]

1.4 Mathematica/Wolfram Language Today

1.4.1 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: [63]

- 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. [64]
- 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 [63] and available for licensing.

- Wolfram|Alpha: publicly available at no cost in the base version [83] 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.” [66]

1.4.2 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. [62] Mathematica, the Desktop application, continues to be Wolfram Research’s core product, being marketed as the “world’s definitive system for modern technical computing” [64] and is distinguished from underlying technologies (Wolfram Language, Wolfram Cloud, Wolfram Knowledgebase, to name a few out of a longer list [64]) and contrasts with the more unified platform approach of Wolfram|One [84], the publicly available Wolfram|Alpha [83], a set of mobile apps [63] 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 1.4.4 for the exploration of this idea. The current count of in-built functions, 6602 for the latest major release 14.0 [71] and, for the newly released (minor) Version 14.1, up 89 to a new total of 6691 [82], with the trajectory since Version 1.0 in 1988 given in Figure 1.5.

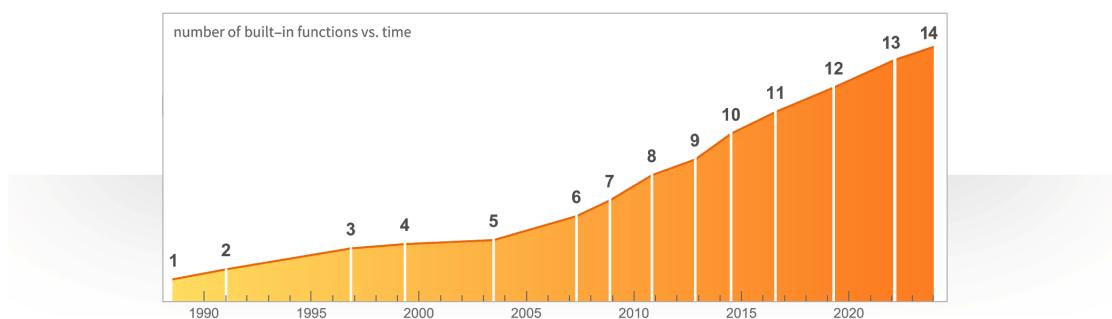


Figure 1.5: In Mathematica Desktop, there are now over 6000 “built-in functions.” [64]

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 [58]. Specifically the former has been ad-

dressed 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” [63] is the official customer facing answer in this proprietary context. A more extensive statement is given in [59], 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.

1.4.3 How Wolfram Research Views Mathematica/Wolfram Language: The Computational Language Idea

In addressing the question, “What Kind of a Thing Is the Wolfram Language?” [57], 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).’ [57] 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. [57]

Mars [42] and New York City [11] are examples of entities that can be addressed in WL using the so-called Wolfram Knowledgebase [63], 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 [57], in 2019

- now 6602 in Mathematica and WL Version 14.1 [55] 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. [57]

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.

1.4.4 The Connection to First Order Predicate Logic

In propositional logic, it is not possible to model predicates like “ x is prime”, nor can we reason about statements like “for all x , there exists y such that y is a factor of x ”. [35]

To handle these kinds of statements, predicates and quantifiers need to be introduced; this extended logic is referred to as predicate logic or first-order logic (FOPL):

$$\phi \in \text{Formulae} ::= \dots | P(x_1, x_2, \dots) | \forall x, \phi | \exists x, \phi$$

where x is drawn from a fixed set of variables. [35]

Formally, an interpretation can be modeled as a set D and a function $I : \text{Pred} \times D \times D \times D \times \dots \rightarrow \{T, F\}$ so that one can talk about the truth of a formula in a given interpretation. [35]

WL, as a framework, aligns with FOPL, in the sense of the Computational Language concept already explored, leveraging the ability of users to express computational ideas at a high level, in a FOPL style, but providing the means to evaluate expressions, integrating various data types and a sophisticated set of algorithms and front end, and other aspects of a “system for doing mathematics by computer” [69].

As a thesis in the field of Software Engineering, this work will limit its theoretical exploration of the project topic to the technical aspects of the implementation, rather than the mathematical ones, in the following chapter.

Chapter 2

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

This thesis already started off with rewriting-theory, so in this chapter I would like to focus the relevant insights from Theorema itself, as well as WL-systems-building and WL-paradigms, on practically relevant takeaways for the project part.

2.1 Theorema

Theorema is currently available in Version 2.0, under GPL [27] and including the full source code on GitHub [24]. A tutorial is available as well. [60]

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]

2.1.1 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* [44] and *FindEquationalProof* [20].

2.1.2 *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 2.1 (splash screen at application start), 2.2 (start screen under Windows) and 2.3 (again the Theorema Commander, this time under Linux, taken from [61]), profiting from GUI-friendly dynamic expressions [29], p. 76] and cascading stylesheets [48] 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 reliant on prior knowledge of the language:

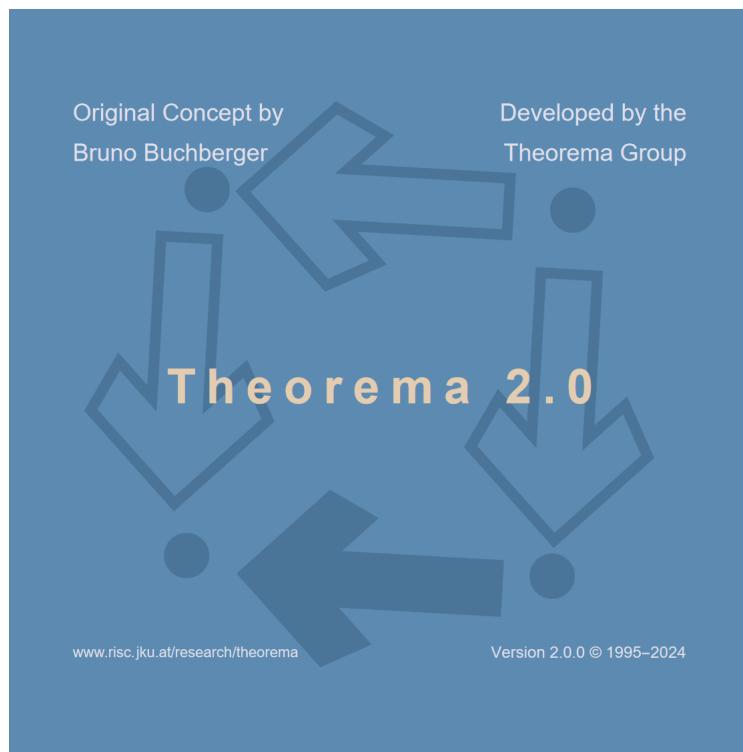


Figure 2.1: The Theorema splash screen introducing the project and including a spinning RISC logo in the background.

As an example, giving a definition meant evaluation of a `Definition[...]`-command, stating a theorem meant evaluation of a `Theorem[...]`-command,

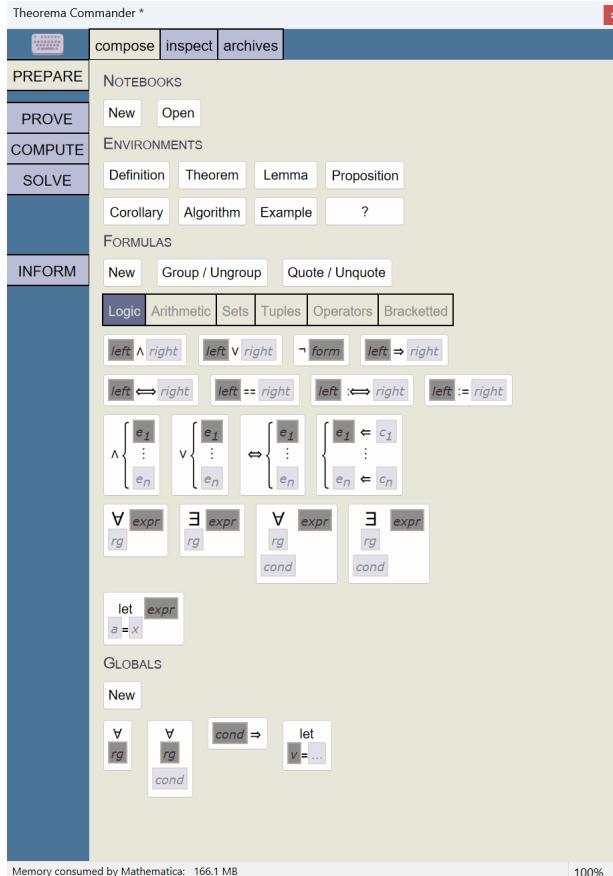


Figure 2.2: The start view in the Theorema Commander window: on Windows, the Commander opens in its own window, emulating an interface for a separate notebook.

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. [61, p. 73]

The target user groups are mathematicians and students of mathematics [61, 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]

The current project structure is also made explicit in Figure 2.5.

2.1.3 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 1.1.1

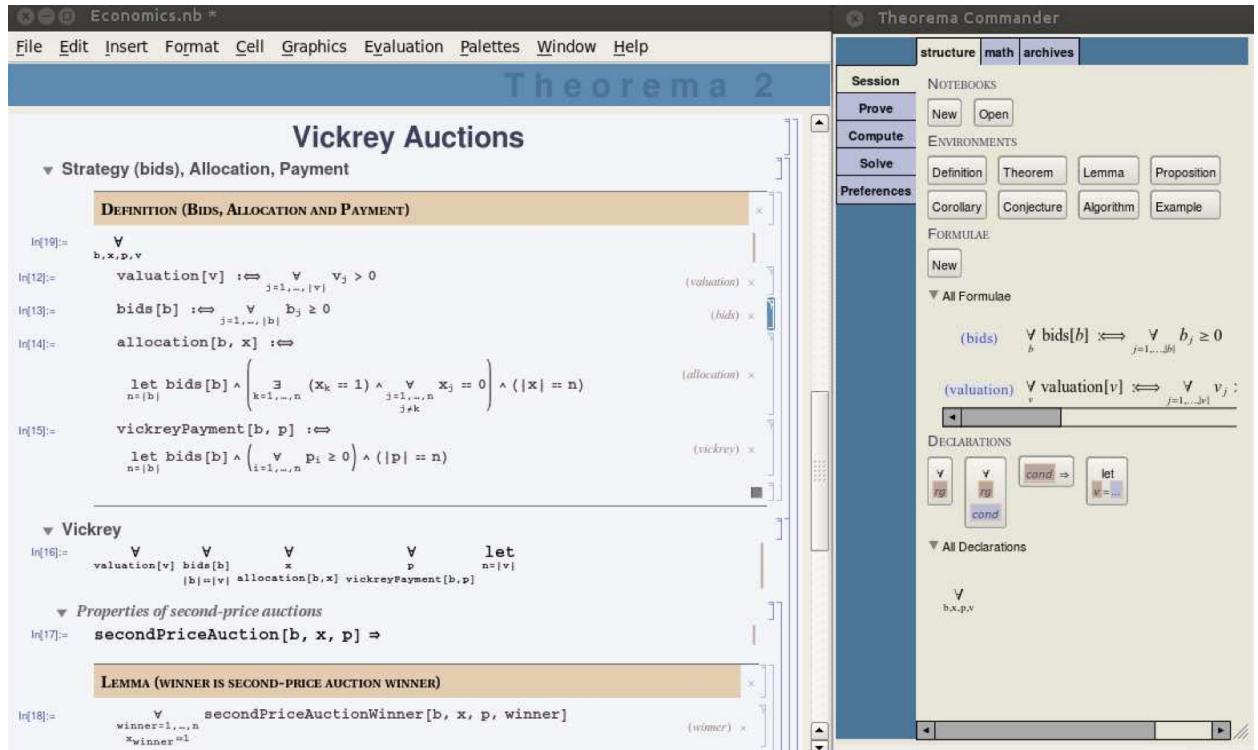


Figure 2.3: Taken from [61], this side-by-side view of a notebook with the Commander also shows off the virtual keyboard, including “keys” for formula elements: comparing to the current version of the Commander in [2.2], some changes have been made to the interface options, e.g. “Session” appears to be “Prepare”

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.

The consequence of this theory for the current project of transforming Theorema documents is that the L^AT_EX-transformation target language needs to be able to represent Predicate Logic, and particularly, First Order Predicate Logic (FOPL).

2.1.4 Theorema Environment and Surfacing the Theorema Language

Theorema data is read from Mathematica notebooks [61, 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 L^AT_EX-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 2.3.

As soon as the [Theorema] formula is passed to the system through Mathe-

matica'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. [61, p. 75]

We already saw a Theorema formula expression in the introductory chapter [1], Section ??: the structure starts like `{Theorema`Common`FML$[{"ID:169304498", ...}` where we are actually looking at a list of (denoted by curly braces) containing multiple `Theorema`Common`FML$` 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.” [61, 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 they refer) or an “orphaned” implication (missing the right hand side), or an abbreviation indicated by a “let.” [61] names this biimplication:

$$\text{bids}[b] : \iff \forall_{j=1, \dots, |b|} b_j \geq 0 [61, p. 76]$$

This actually translates to:

$$\forall_b \text{bids}[b] : \iff \forall_{j=1, \dots, |b|} b_j \geq 0 [61, p. 76]$$

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 [2.4], 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.

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.” [54]. 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]:

- Divide the System into Components
- Write and Use Unit Tests

Program 2.1: This is an excerpt from the notebook cell expression depicted in the front-end rendering in Figure 2.4 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"]

```

The screenshot shows the 'Domain Definitions' interface in Theorema. It displays three global declarations:

- `in[1]:= \mathbb{K}`
- `in[1]:= $\text{Mon}[\mathbb{K}] := \Delta_{\mathbb{K}}$`
- `in[1]:= $\forall m_1, m_2$`

For the third declaration, there is a detailed description:

$$\text{m1} * \text{m2} := \left(\text{m1}_1 * \text{m2}_1, \left((\text{m1}_2)_1 + (\text{m2}_2)_1 \mid_{1-1, \dots, |m_1|} \right) \right)$$

and a condition:

$$\left(\text{m1} < \text{m2} \right) \Leftrightarrow \text{m1}_2 <_{\text{lex}} \text{m2}_2$$

Figure 2.4: A global declaration in Theorema with three different terms.

- 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 [41], namespace management [38], and further scoping constructs [46] 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 [38], 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 the package if the specified context is not already in `$Packages`, the relevant context variable in this case. [39])

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, see [2.5]- the proposed format of the functionality implemented with this project is therefore also a WL-package, a file with file ending ".wl" (current) or ".m" (historically) and following the layout given by this Theorema template file ("Theorema /PackageTemplate.m" [56]) - including the copyright statement, which bakes the GNU (a recursive acronym, "GNU's Not Unix:" The GNU Project was initiated by Richard Stallman in 1983 and is a free software, mass collaboration project [52]) 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). [51] (However, the software is provided without any warranty.)

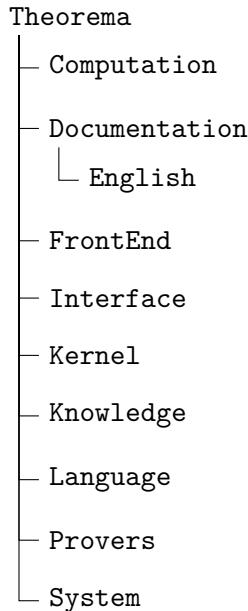


Figure 2.5: Directory Structure of the main directory in the Theorema project; directories are filled with .m-Mathematica/WL package files.

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." [43] 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 1.1.2.

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. The present author would like to reference his exposé [D] of the topic of contrasting these programming styles as they are applied to

one particular example, Program [2.3] and move on to the rule-based approach most relevant for the project at hand, after citing the seamless conversion of geometric objects to Unity (the game engine [45]) objects just one example of the high-level, abstracted idea of providing functionality in the WL ecosystem:

Drawing on its algorithmic power, Version 12 provides high-level functions that are uniquely easy to use to create and manipulate Unity objects. Easily create game objects with `CreateUnityGameObject` and directly include geospatial and socioeconomic data, and across thousands of domains. [26]



Figure 2.6: Plots, graphics, and geometric objects are automatically converted to Unity's mesh format during the creation of a game object. [26]

Curated knowledge (data packages) available directly inside the WL system complement this paradigm-combination, as is made apparent in this example - see also Figure 2.6.

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

2.3.3 Rule-Based Programming

Another prominent mathematical research institute in Linz works with Mathematica [53], extending its rule-based programming capacity in the *ρLog* package [33] - much like Buchberger, Marin and Piroi identify Mathematica as a powerful system implementing the paradigm they are interested in, rule-based programming, the more general one

Program 2.3: These functions extract a tree data structure in the form of certain integer mathematical proof IDs and the related children IDs from a grid expression in WL and serve well to illustrate lists and replacements, functional and rule-based programming, as well as recursion, for efficient implementations. An exposé with details is part of the appendix in the present work. [D]

```

1 proofID[Grid[{___, {ID, id_}, ___}, ___]] := id;
2 subproofs[
3 Grid[{___, {Proofs, OpenerView[{Arguments, Column[subproofs_, ___]}, ___]}, ___}, ___],
4 ___] := subproofs; subproofs[proof_] := {};
5 getLeanTree[proof_] := Tree[proofID[proof], getLeanTree /@ subproofs[proof],
    TreeElementLabelStyle All Directive[White, 16, FontFamily "Times New Roman"],
    TreeElementStyle All Directive[EdgeForm[Black], RGBColor["#B6094A"]]]

```

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 $\text{Expr}_1 \rightarrow_{rr} \text{Expr}_2$, where Expr_1 , Expr_2 are given expression schemata (patterns), and \rightarrow_{rr} is a specification of a sequence of rule reduction steps. Typically, the specification of \rightarrow_{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.

[33]

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 $\exists\{R_1, \dots, R_k\} \text{Expr}_1 \rightarrow_r \text{Expr}_2$ 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, \dots, R_k (see Section 2.2) [of [33]].

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.

[33]

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.

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.

Integration

In the context of Wolfram Language and systems like Theorema, rule-based programming and pattern matching are core tools used to process and transform symbolic expressions. The interplay of these techniques allows for flexible and powerful manipulation

of expressions, crucial for this project's task of converting mathematical expressions into L^AT_EX.

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 L^AT_EX 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

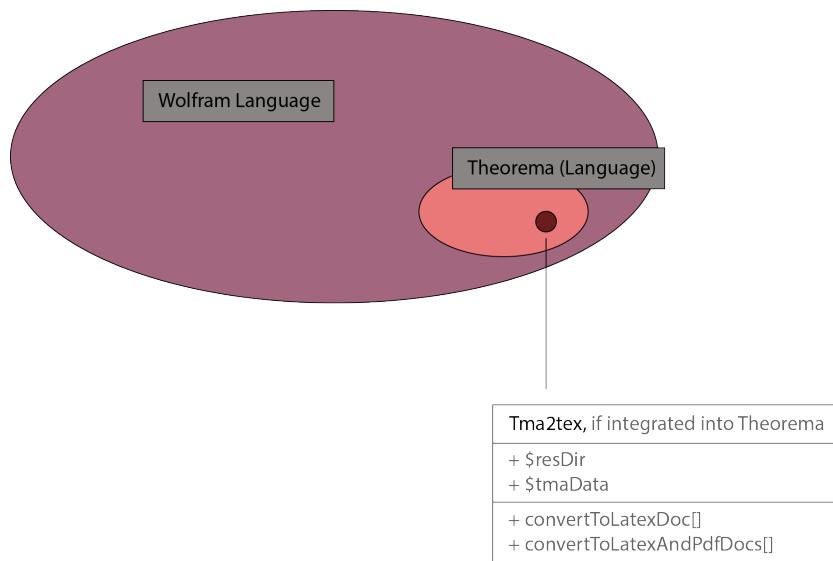
The design for the package is such that it is intended to be integrated with the Theorema project as a stand-alone package. Figure 3.1 illustrates the relationship of this package to Theorema (Language) and Wolfram Language, as a whole.

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 L^AT_EX-template is used for customizations, at output: the template is filled by the L^AT_EX generated by transforming the relevant Theorema Language formulas, particularly, but the Wolfram Language notebook as a whole, generally.

3.1.2 Existing (Kernel) Functionality: 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 [81] connection), evaluates the expression and prints the result to a device (typically a screen or a WSTP

**Figure 3.1**

connection).“ [79] 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 L^AT_EX. 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.

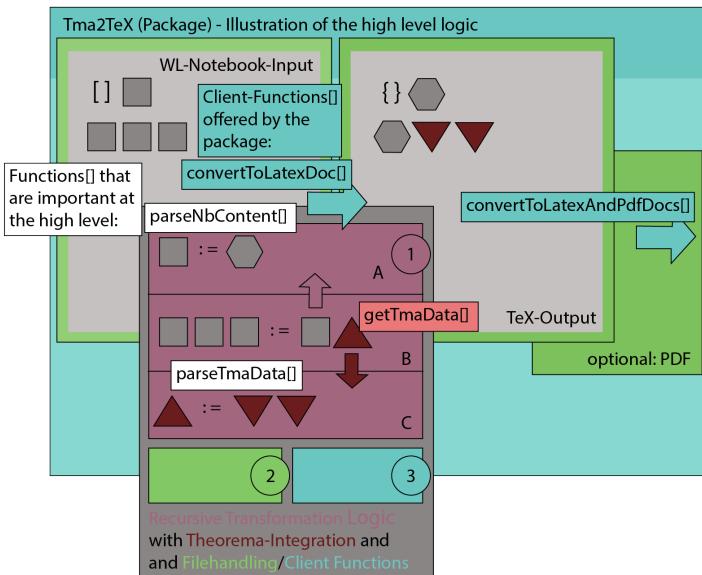


Figure 3.2

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

3.1.3 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)

- **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_, resourceDirOptional: 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 [3.1.5](#). 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 `MakeBoxes` [68] 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 `string`TM-named L^AT_EXcommand in `tmaTemplate`, e.g. `\newcommand{\ForallTM}{[2]{\forall}}` 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 convertToLatexAndPdfDocs, will be of interest, and however level MakeTex only secondarily for specific cell transformations as needed, or for customizations of the transformation, and especially for users well-aquainted with the package.

3.1.4 For This Project: No Layout-Information in the L^AT_EX

Having considered the native TeXForm implementation and its reliance on native MakeBoxes, 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 L^AT_EX, but to work directly with the Theorema-Formula and give the user complete control over layout via L^AT_EXand 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 L^AT_EX-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[{"\[ForAll]"},
      RowBox[["x", "ExpressionVariable"]]}],
      RowBox[{ 
        RowBox[{"P", "["},
        StyleBox["x", "ExpressionVariable"], "]"}], "\[Or]",
        RowBox[{"Q", "["},
        StyleBox["x", "ExpressionVariable"], "]"}]]}], "\[And]",
      RowBox[{ 
        RowBox[{"\[ForAll]"},
        UnderscriptBox["y", "ExpressionVariable"]]},
        RowBox[{ 
          RowBox[{"P", "["},
          StyleBox["y", "ExpressionVariable"], "]"}], "\[Implies]",
          RowBox[{"Q", "["},
          StyleBox["y", "ExpressionVariable"], "]"}]]}], ")"],
"\[DoubleLeftRightArrow]",
RowBox[{ 
  UnderscriptBox["x", "Q[x]"],
  RowBox[["x", "ExpressionVariable"]]}], TheoremaForm]

```

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

```

\left(\underset{x}{\forall } P[x]\vee Q[x]\wedge
\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.

3.1.5 MakeBoxes: An Alternative Typesetting-Pipeline

Both the Theorema and the TeXForm packages provide MakeBoxes-functionality [32], 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-L^AT_EX commands 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 be registered (on the Mathematica side) and specified (on the L^AT_EXtemplate side). Conceptually, for this implementation, this mechanism requires a symbol-level adaptation of TeXForm 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 L^AT_EXentirely, 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 cer-

tain 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 L^AT_EX-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 L^AT_EXcode far right on bottom in the graphic, with both document and formula level output.

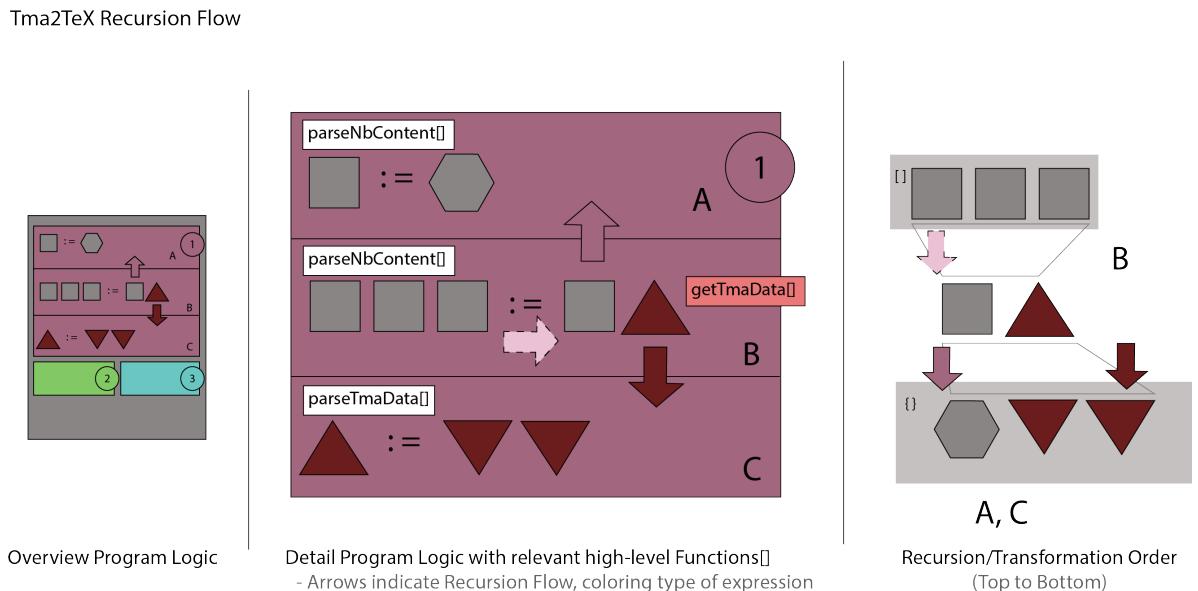


Figure 3.3

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[1_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 L^AT_EX/PDF output, to mark this distinction outwardly.

3.2.2 Pattern Matching to Realize L^AT_EX-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 like Theorema`Language`Iff$TM[Theorema`Language`And$TM[Theorema`Language`Forall$TM[... }}`

The easy suffix "TM" is chosen for the L^AT_EXoutput, 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 L^AT_EXresult 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 "Knowledge" in their context path. [65] 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 L^AT_EX-template file.

3.3 Extensibility in Both L^AT_EXand 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 [70] - 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.

Testing will make up a large part of Chapter 5.

```
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. [75]

3.3.3 Extensibility

Extensibility at both the WL and L^AT_EXlevels is the other priority, with the understanding that Theorema Language might change, Wolfram Language might be updated, and additional L^AT_EXcommands 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-L^AT_EX-output that maybe used for string replacements, for instance, at the level of the formula-L^AT_EXsnippets.
- 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 L^AT_EXis 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 3.1.3, 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 (rather 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

[This is related, probably: Building large software systems with Wolfram Language. See Concepts section. CompoundExpression [12], Module [37], ... Block? [5]]

Concretely for this package, in standalone form, the package loads Theorema via a `Needs[]` call: 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

In high-level programming within WL, the focus is primarily on manipulating symbolic expressions and applying transformation rules rather than managing low-level implementation details. This abstraction layer allows developers to craft powerful programs with minimal code, leveraging the language's built-in functions for data manipulation, pattern matching, and rule-based transformations. [Ref.]

The Wolfram Language offers several key features that support high-level programming, as seen in both filehandling and the client functions in this project:

- **Symbolic Computation**: All expressions in the Wolfram Language are treated symbolically, allowing functions to operate on concrete data, as well as symbolic representations of mathematical expressions, code, or documents.

- **Pattern Matching and Transformation Rules:** Advanced pattern matching capabilities facilitate the definition of rules for transforming expressions, simplifying the implementation of complex algorithms in a clear and concise manner.
- **Functional Programming Constructs:** Functions such as `Map`, `Apply`, and `Fold` support a functional programming style, enabling operations on lists and other data structures without explicit loops.
- **Built-In Algorithms and Computational Knowledge:** The language integrates a vast repository of algorithms for numerical computation, algebra, statistics, and other domains, alongside access to curated data, allowing for the resolution of problems at a high level without the need to develop standard methods from scratch.
- **Notebook Interface:** The Wolfram Language is often used within interactive notebooks, providing a rich environment for computation, visualization, and dynamic document creation. This interface enhances the development process by offering immediate feedback and facilitating exploratory programming.

[Ref.]

By leveraging these features, developers can write programs that are not only more concise and readable but also easier to maintain and adapt, emphasizing the core logic of their applications rather than low-level programming concerns.

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 functionality 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 \LaTeX Details

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[$documentProcessingLevel == "Full",
  "\color{blue}", ""]
parseNbContent[l_List] /; MemberQ[l, _Cell] :=
  StringJoin[If[$documentProcessingLevel == "Full",
    "\color{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.

[Ref.]

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 information, 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[$documentProcessingLevel != "None", "\\\begin{group} \\\section*{} "
    <> text <> "\\\endgroup \\n\\n", ""]
parseNbContent[Cell[text_String, "Section", ___]] :=
  If[$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 preceding 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 mathematics, 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 locate 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 expression that needs to be converted to something like

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

for example: The ranges and variable expressions are transformed to parameterizable L^AT_EX macros that can be defined exactly the way the user wishes, L^AT_EX-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 L^AT_EX 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 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

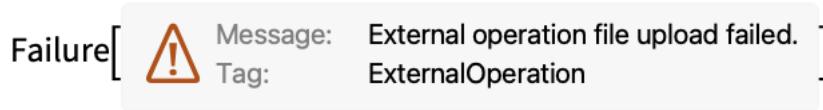


Figure 5.2: A Failure object using a template with positional parameters [18]

to determine the cause of the error and decide how to proceed: They are particularly 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 [34] 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. [34] 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). [78] More recently, TestCreate ([72]) and TestEvaluate ([73]) were introduced to the language, to create and run TestObjects, ([50]) 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 [74] 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 AbsoluteTiming functions are used to measure the performance of code in WL. In addition, BenchmarkReport can pro-

vide detailed performance analysis, comparing the execution time of different functions or code segments. [77]

- **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 [80] for commandline support - though so-called Testing Notebooks provide an interface for writing and running tests as well [78]

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 \LaTeX files, and handling file operations, are tested in an outside view (irrespective of \LaTeX code 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 \LaTeX code 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 instance. Functions like

`parseTmaData` and file generation methods would be profiled using `Timing` and `AbsoluteTiming` to identify any potential bottlenecks and ensure optimal performance. [78]

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 L^AT_EX. (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.

2.2.3 Global Declaration

$$\forall_{m1, m2}$$

$$\begin{aligned} m1 \underset{M}{*} m2 &:= & <m1_1 & \underset{K}{*} & m2_1, & <(m1_2)_i & + \\ (m2_2)_i & | & >> (m1 \underset{M}{<} m2) \text{Pattern not found! TagBox[RowBox[{:}, :], Identity]} \end{aligned}$$

Cell reached Cell reached Cell reached CellGroupData reached List of cells
reached

Figure 5.3: `parseNotebookContent[other_]` output in document with L^AT_EXcommand formatting

5.2 Analysis and Review

The goal of the project was to develop a robust mechanism for transforming Wolfram Language/Theorema notebooks into L^AT_EXdocuments: 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 L^AT_EXcounterparts. This approach was essential for preserving both the mathematical expressions and structural 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 effectiveness in automating the transformation process. Tests on various Theorema notebook structures demonstrated the accuracy of the L^AT_EXoutput, which faithfully represented 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 L^AT_EXpreparation significantly.

In conclusion, the project successfully addressed the initial project objective and provides a functional tool for automating document preparation in Theorema. The project also underscored the potential of Wolfram Language as a versatile tool for software engineering, 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.

The project repository, supporting documents, and an introductory exposé of WL are submitted with this thesis.

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-L^AT_EXtypsetting system and less reliance on output-side macros (the L^AT_EXtemplate 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.

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/heseltine/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
16      Cleaned–Up File
17      – Move pre–August 2024 work to "V0"
18
19 Purpose: This program recurses over the Theorema notebook structure to produce a LaTeX
20 representation, including of the
21 underlying Theorema–Datastructure: to this end it inserts the appropriate LaTeX–
22 commands into an output file, mediated
23 by a template in the $resDir. Both the notebook– and Theorema–level content is rendered
24 using the relevant LaTeX packages
25 and should be modified in LaTeX for the output syntax.
26
27 Part 1 of this package is concerned with the described double–recursion, drawing mainly
28 on the Theorema–data provisioned in Part 0.
29 For academic purposes, Part 1 is subdivided in Parts
30
31 * A: parseNbContent, the main recursive function, with the output–nearer LaTeX–
```

```

27   commands,
28   * B: also parseNbContent, higher level, Theorema—notebook specific pattern—recursion
29   rules,
30   * C: getTmaData and parseTmaData, concerned with establishing the connection between
31   the appropriate part in the input notebook/
32   output LaTeX and the given Theorema data, and parsing, again recursively, the formula
33   structure, respectively .
34
35 Part 0 is also subdivided in an out/inside—of—package Part A and B respectively, to
36 illustrate packaging in Wolfram Language.
37
38 The result is inserted into the appropriate LaTeX template (Part 2, Filehandling): the
39 main functions intended
40 for use by the client is at the end of the program, Part 3, specifically
41 convertToLatexAndPdfDocs[] as the all—in—one transformation
42 function for stand—alone—calling — but convertToLatexDoc[] for the basic Theorema use
43 case. Both functions take the path to the relevant
44 Theorema notebook as their single parameter.
45
46 ---- *)
47
48
49 (* << Theorema` *)
50 (* Uncomment the Tma—Get call if NOT called from inside the Tma—Package or an
51 environment that loads Tma already *)
52 Needs["Theorema`"]
53
54 (* -- Part 0.A.2 Global Variables: Important for interfacing with Theorema. -- *)
55 Tma2tex`$resDir::usage = "Defines the directory for LaTeX-templates and any other
56 resources."
57 Tma2tex`$resDir = "C:\\\\Users\\\\jackh\\\\git\\\\repository\\\\tma2tex\\\\res"
58
59
60 Tma2tex`$tmaData::usage = "Containies the Theorema-Datastructure that holds formula-
61 experessions and is therefore typically equivalent to
62 Theorema`Common`$tmaEnv on the Theorema-side, but can be used to show the content
63 according to Tma2Tex (as a separate package)."
64
65
66 (* -- Part 0.A.3 Client—Function—Usage Messages *)
67 convertToLatexDoc::usage="convertToLatexDoc[notebookPath] transforms a given WL
68 notebook (by file path) to TeX output, creating a new TeX file from a specified
69 resource template."
70 convertToLatexAndPdfDocs::usage="convertToLatexAndPdfDocs[notebookPath] transforms a
71 given WL notebook (by file path) to PDF file as final output, with TeX file as
72
```

```

    intermediary step, from a specified resource template."
69
70 (* Test-wise: *)
71 convertToLatexFromString::usage="convertToLatexFromString[nbContentString_,
    resourceDir_Optional]: Tma2tex`$resDir] is experimental and intended be called
    from the Cloud, simply transofrmimg Wolfram Language String Input to TeX Output
    (returned directly, not via file). Also uses a template, the resource for which
    can be passed as the second argument."
72
73 Begin["`Private`"]
74
75 (*Needs["Texformdump`\"]*) (* loaded in Private` for internal implementation details *)
76 (* See TeXForm Notes in the Thesis: this approach was not chosen, finally . *)
77
78 (* -- Part 0.B, Imports and Global Variables INSIDE-OF-PACKAGE -- *)
79
80 (* Custom messages for problems importing or invalid settings *)
81 tmaDataImport::empty = "`1`";
82 documentProcessingLevel::invalidParameter = "`1`";
83
84 (* The following holds the Tma-Formula-List as an association with keys from the IDs, gets
     filled in getTmaData[] *)
85 tmaDataAssoc = <||>;
86
87 (* Define the global private variable with a default value *)
88 Tma2tex`$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[$documentProcessingLevel
    == "Full", "\color{green}", ""] <> parseNbContent[1]
109 (* CellGroupData often contain relevant content *)
110
111
112 (* -- Part 1.A.1 -- Text Expressions (at the Cell Level): Not processed if
   DocumentProcessingLevel = "None", otherwise yes. *)
113
114 parseNbContent[Cell[text_String, "Text", ___]] := If[$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{\exists}{", parseNbContent[cond] <>
    "}"}, ""]
145 parseNbContent[UnderscriptBox["\[ForAll]", cond_]] :=
146 If[$documentProcessingLevel != "None", "\underset{\forall}{", parseNbContent[cond] <>
    "}"}, ""]
147
148 (* -- Part 1.A.4 -- Box-Structure Parsing -- See TeXForm Notes in Thesis and compare
   parseTmaData,
149 this was the formula parsing approach finally selected here *)
150
151 parseNbContent[RowBox[{left_, "[And]", right_}]] :=
152 If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left], " \land
    ", parseNbContent[right]], ""]

```

```

153 parseNbContent[RowBox[{left_, "\[Or]", right_}]] :=
154     If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left], " \\\lor
155         ", parseNbContent[right]], ""]
156
157 parseNbContent[RowBox[{left_, "\[DoubleLeftRightArrow]", right_}]] :=
158     If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left], " \\\
159         Leftrightarrow ", parseNbContent[right]], ""]
160
161 parseNbContent[RowBox[{left_, "\[Implies]", right_}]] :=
162     If[$documentProcessingLevel != "None", StringJoin[parseNbContent[left], " \\\
163         Rightarrow ", parseNbContent[right]], ""]
164
165 parseNbContent[RowBox[{left_, "<", right_}]] :=
166     If[$documentProcessingLevel != "None", parseNbContent[left] <> " < " <>
167         parseNbContent[right], ""]
168
169 parseNbContent[RowBox[{left_, ">", right_}]] :=
170     If[$documentProcessingLevel != "None", parseNbContent[left] <> " > " <>
171         parseNbContent[right], ""]
172
173 parseNbContent[RowBox[{left_, "\[SubsetEqual]", right_}]] :=
174     If[$documentProcessingLevel != "None", parseNbContent[left] <> "\\\subseteq" <>
175         parseNbContent[right], ""]
176
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____],
193     envcells____}, ____]]] :=
194     Module[{contentStrings},
195         contentStrings = StringJoin[parseNbContent /@ {envcells}]; (* Apply parsing to each
   cell *)
196         StringJoin[
197             "\\\begin{tmaenvironment}\n",
198             "\\\subsection{", parseNbContent[headertext], "} \n",
199             contentStrings,
200             "\\\end{tmaenvironment}\n"]
201     ]
202 *)

```

```

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 intermediate Tma-Definitions: 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

```

```

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]" -> "\\subsetneq"]
339
340
341
342 (* -- Part 1.C.0, Recursive Pattern Matching: getTmaData[] selects the relevant part in
   Theorema`Common`FML$ in preparation
   for a second recursive descent, see 1.B.2 -- *)
343
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       "}",
365       3, "{" <> parseTmaData[argList[[1]]] <> "}" <> parseTmaData[argList[[3]]] <>
366       "}",
367       _, "" (* unexpected number of arguments: abort the parse tree here *)
368     ];
369     "\\" <> ToString[nextOp] <> parsedArgs
370   ]
371
372 (* Alternative 2: Knowledge-case – Predicate or Function Symbol, not Language-Operator *)
373 parseTmaData[op_[args___]] :=
374   Module[{nextOp, argList, parsedArgs},
375     nextOp = prepareSymbolName[op];
376     argList = {args};
377     parsedArgs = Switch[
378       Length[argList], (* 1 argument in the majority of cases *)
379       1, "[" <> parseTmaData[argList[[1]]] <> "]",
380       2, "[" <> parseTmaData[argList[[1]]] <> ", " <> parseTmaData[argList[[2]]] <>
381       "]",
382       _, "" (* unexpected number of arguments: abort the parse tree here *)
383     ];
384     " " <> ToString[nextOp] <> parsedArgs (* Does not get prefixed with "|!" *)
385   ]
386
387 (* Alternative 3: Special Case Two-Argument-Sets e.g.
388   Theorema`Language`Annotated$TM/Theorema`Language`Less$TM,
389   Theorema`Language`SubScript$TM/Theorema`Knowledge`lex$TM]]
390   [Theorema`Language`VAR$/Theorema`Knowledge`VAR$a$TM],
391   Theorema`Language`VAR$/[Theorema`Knowledge`VAR$b$TM]]
392
393   ... transforms to more complex macro *)
394 parseTmaData[op_?isTmaLanguageSymbol[args___][args2___]] :=
395   Module[{nextOp, argList, argList2, parsedArgs, parsedArgs2},
396     nextOp = prepareSymbolName[op];
397     argList = {args};
398     parsedArgs = Switch[
399       Length[argList],
400       1, "{" <> parseTmaData[argList[[1]]] <> "}",
401       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[argList2[[1]]] <> "}" {" <> parseTmaData[argList2[[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$/Theorema`Knowledge`*
417   VAR$m1$TM], 2],
418   2 eventually gets processed*)
419   ToString[i]
420
421 (* Recursion-Stop (Alternative 5) Axiomatic Expression/No Operation *)
422 parseTmaData[ax_] := (* e.g. Theorema`Knowledge`VAR$x$TM, i.e. axioms/parse-tree leaves
423   *)
424   prepareSymbolName[ax]
425
426 (* -- Part 1.C.2, Auxilliary Functionality -- *)
427
428 isTmaLanguageSymbol[f_Symbol] := (* Context required here to distinguish from predicates like
429   Theorema`Knowledge`P$TM *)
430   With[{n = SymbolName[f], c = Context[f]},
431     c === "Theorema`Language`"
432   ]
433 isTmaLanguageSymbol[f_] := False
434
435 prepareSymbolName[op_Symbol] :=
436   With[{n = SymbolName[op]},
437     If[StringTake[n, -3] == "$TM",
438       If[StringTake[n, 4] == "VAR$", (* e.g. Theorema`Knowledge`VAR$x$TM *)
439         (*"||" <> *)StringDrop[StringDrop[n, 4], -3](* <> "TM"),
440         (*else*) (* e.g. Theorema`Language`Forall$TM, but: Theorema`Knowledge`Q$TM
441         *)
442         If[isTmaLanguageSymbol[op], (* only language should be suffixed with TM for
443           TeX-Macros *)
444             (*"||" <> *)StringDrop[n, -3] <> "TM",
445             (*else*)
446               StringDrop[n, -3]
447             ]
448           ],
449         (*else*)
450           If[StringTake[n, -1] == "$", (* e.g. VAR$ *)
451             StringDrop[n, -1] <> "TM",
452             (*else*)
453             ]
454           ]
455         ]
456       ]
457     ]
458   ]
459 
```

```

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,
502   latexPath, latexTemplatePath,
503   resourceDir = $resDir, texResult, sownData, filledContent, closeFlag = False,
504   documentProcessingLevel},

```

```

503 If[Length[$tmaData] == 0, (* Issue message if Theorema-Formula-Data not provisioned *)
504 Message[tmaDataImport::empty, "The Theorema-Formula-Datastructure is empty.
505 Did you evaluate a Theorema notebook before loading the package and calling the
506 conversion function?"];
507 (* Additional handling for empty data can be added here *)
508 Return[$Failed]
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 }];
530 filledContent = fillLatexTemplate[resourceDir,
531 <|
532 "nbContent" -> texResult,
533 "nbTitle" -> First[sownData[[1, 1]]],
534 "nbAuthor" -> First[sownData[[2, 1]]],
535 "nbDate" -> First[sownData[[3, 1]]]
536 |>];
537 Export[latexPath, filledContent, "Text"];
538 (*Print[Theorema`Common`$tmaEnv];*)
539
540 If[closeFlag === True, NotebookClose[notebookPath]];
541
542 (* Helper fn to determin if the notebook specified by the given path is open *)
543 isNotebookOpen[path_] :=
544 Module[{c},
545 Quiet[MemberQ[Notebooks[],
546 n_ /; (c = "FileName" /. NotebookInformation[n]; c[[2]]) ===
547 FileNameTake[path, -1]]]]
548
549 (* Helper fn to set the document processing level *)
550 SetDocumentProcessingLevel[level_String] :=
551 Module[{},
552 If[MemberQ[{"Full", "None", ""}, level],
553 Tma2tex`$documentProcessingLevel = level,
554 Message[documentProcessingLevel::invalidParameter, "Invalid parameter (option)
555 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,
561   pdfPath, compileCmd, conversionResult},
562   conversionResult = convertToLatexDoc[notebookPath, DocumentProcessingLevel->
563     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] :=
576   Module[
577     {nbContent, texResult, sownData, filledContent},
578
579     (* Convert the string representation to a Wolfram Language expression *)
580     nbContent = ToExpression[nbContentString, InputForm];
581
582     (* Process the notebook content *)
583     {texResult, sownData} = Reap[parseNbContent[nbContent], {"title", "author", "date"}];
584
585     (* Fill in the LaTeX template with parsed content *)
586     filledContent = fillLatexTemplate[resourceDir,
587       <|
588         "nbContent" -> texResult,
589         "nbTitle" -> First[sownData[[1, 1]]],
590         "nbAuthor" -> First[sownData[[2, 1]]],
591         "nbDate" -> First[sownData[[3, 1]]]
592       |>
593     ];
594
595     (* Return the filled LaTeX content as a string *)
596     filledContent
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: First TourÖOutput in LaTeX Format

```

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-
11 % greater-than-symbol-appear-wrong-as
12 \usepackage{lmodern}
13
14 %% For development purposes
15 % Define a command for light gray text for structure elements
16 \newcommand{\light}[1]{\color{lightgray}#1}
17
18 %% Structural elements of LaTeX document
19 % Gray box for Tma envs
20 \usepackage{tcolorbox}
21
22 %% Approach 1: Define a new tcolorbox environment for theorema environments
23 \newtcolorbox[tmaenvironment]{
24   colback=gray!20, % Background color: gray with 20% intensity
25   colframe=black, % Frame color: black
26   boxrule=0, % Frame thickness
27   arc=4pt, % Corner rounding
28   boxsep=5pt, % Space between content and box edge
29   left=5pt, % Left interior padding
30   right=5pt % Right interior padding
31 }
32
33 %% Strucutral elements of the symbol level
34 % QED - gray square
35 \usepackage{tikz}
36 \newcommand{\graysquare}{\tikz\fill[gray] (0,0) rectangle (0.2cm,0.2cm);\hspace{0.2cm}}
37
38 %% Centered diamond shape in a specified color
39 \newcommand{\colordiamond}[1]{%
40   \tikz[baseline={([yshift=-0.3ex]current bounding box.center)}]
41   \fill[#1] (0,0) -- (0.08cm,0.08cm) -- (0,0.16cm) -- (-0.08cm,0.08cm) -- cycle;\xspace
42 }
43
44 % e.g.
45 % \colordiamond{red}
46 % \colordiamond{green}
47 % \colordiamond{blue}
48
49 %% Define a command to create a right-aligned, smaller legend
50 \newcommand{\legend}{%
51   \begin{tcolorbox}[
52     title=Tma2tex-parsing Info/Legend,
53     colback=white,
54     colframe=black,
55     boxrule=0.5mm,
56     right=10pt,
57     fonttitle=\bfseries\tiny,
58     fontupper=\tiny,
59     fontlower=\tiny,
60     width=100pt,
61     height=100pt,
62     center,
63     sharp corners,
64     opacityback=0.5
65   ]}

```

```

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 \color{yellow} Yellow: Represents entry points to parsing. \\
64 \color{orange} Orange: Helper Definitions were defined in the Theorema
Notebook interface, but are directly included in the following formula. \\
65 \color{red} Red: Matches unspecified cells or generic content. \\
66 \color{blue} Blue: Represents lists of specific content. \\
67 \color{purple} Purple: Used for lists of generic cells. \\
68 \color{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
110 %% convention is
111 %% to use the symbol name in the Theorema code, but without dollar signs or context path.
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{\STEP RNGTM}[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])) \Leftarrow (\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{m2}}{ 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}}}{ \ElementTM{ \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

$$((\forall x (P[x] \vee Q[x]) \wedge \forall y (P[y] \rightarrow Q[y])) \Leftrightarrow \forall x Q[x])$$

2 Computing

Definition : LEXICAL ORDERING

$$\forall ab (LessTM_{lex} (a) b : \Leftrightarrow \exists (a_i < b_i \wedge \forall a_j = b_j))$$

Definition : MONOMIALS

$$\forall Km2 (Mon[K]_{TimesTM} (m1, m2) := (K_{TimesTM} (m1_1, m2_1), [PlusTM](m1_{2i}, m2_{2i})))$$

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

Exposé: A Tree Pattern Function in Mathematica

An exposé of the programming paradigms highlighted in Chapter 2 is included with this thesis work: It was produced as part of the companion seminar class, "Wissenschaftliches Arbeiten," i.e. Scientific Research and Writing, and serves to deliver some additional Theory about programming in WL, separately from the project focus of the main work. The author hopes this aids the interested reader and would like to point out that the document is interesting formally in that it was produced inside a Mathematica notebook, so that the cells containing text and executable WL code are of equal order inside an overall WL notebook expression, as outlined in [1], concerning WL expressions as document representation: as opposed to this project's L^AT_EXtransformation, or the native transformation also referenced in [1], this format represents another way, the non-L^AT_EX, Mathematica-native PDF-export, for making available WL notebook content.

Exposé: A Tree Pattern Function in Mathematica

For: “Wissenschaftliches Arbeiten,” Software Engineering at Fachhochschule Oberösterreich, Campus Hagenberg.

The present author has implemented the following set of Wolfram Language functions in a recent Wolfram Community project (Heseltine, 2023) and would like to expand upon the underlying concepts and the language used, in an effort to further his own understanding and present a helpful guide to the language and its functional programming paradigm support.

```
In[19]:= proofID[Grid[{___, {ID, id_}, ___}, ___]] := id;

subproofs[
  Grid[{___, {Proofs, OpenerView[{Arguments, Column[subproofs_, ___]}, ___]}, ___}, ___],
  ___] := subproofs;
subproofs[proof_] := {};

getLeanTree[proof_] := Tree[proofID[proof], getLeanTree /@ subproofs[proof],
  TreeElementLabelStyle → All → Directive[White, 16, FontFamily → "Times New Roman"],
  TreeElementStyle → All → Directive[EdgeForm[Black], RGBColor["#B6094A"]]]
```

An exposé of the concepts follows, drawing on all three functions (one function is overloaded to produce two forms of the same function), but first an introduction of both the problem space and language will be helpful. The complete code base is available as a GitHub repository.

The Problem Space

The objective is to extract a tree data structure in the form of certain integer mathematical proof IDs and the related children IDs from a grid expression in Wolfram Language. The grid is interpreted by Mathematica but comes from an external program called LEAN, an automated and interactive theorem prover. LEAN and the particular implementation details of processing the tree can be treated as a black box for the purposes of this study.

The Wolfram Language

Wolfram Research (2023) describes the language on a high level like this: “The Wolfram Language is a highly developed knowledge-based language that unifies a broad range of programming paradigms and uses its unique concept of symbolic programming to add a new level of flexibility to the very concept of programming.” Also, on the point of functional programming: “Functional programming is a highly developed and deeply integrated core feature of the Wolfram Language,

D. Exposé: A Tree Pattern Function in Mathematica
made dramatically richer and more convenient through the symbolic nature of the language.”

The functional aspects of Wolfram Language to the point needed to understand the function of interest will be of particular import here, but less so the symbolic side. Mathematica is the concrete programming environment used to explore the concepts with examples.

Lists and Replacements

Going back to our object of interest, here is the first of the set of functions we are presently studying.

```
In[=]:= proofID[Grid[{___, {ID, id_}, ___}, ___]] := id;
```

They use lists (demarcated with curly brackets, here in a nested fashion) in constructing a pattern, which will be discussed in a later section, but lists can also be generated: Lists are central, general objects in the Wolfram Language since they can be made to represent other objects.

Internally, they are, of course, functions.

```
In[=]:= FullForm[{1, 2, 3}]
Out[=]//FullForm=
List[1, 2, 3]
```

Starting with list creation utilities, here are some helpful functions for handling lists.

```
In[=]:= Range[3]
Out[=]=
{1, 2, 3}
```

This is the same as:

```
In[=]:= Range[1, 3, 1]
Out[=]=
{1, 2, 3}
```

Compare this to Table, which takes a Wolfram Language iterator in the form $\{i, imin, imax, step\}$:

```
In[=]:= Table[2 k, {k, 1, 10, 2}]
Out[=]=
{2, 6, 10, 14, 18}
```

As with Range, $imin$ and $step$ can be omitted (set to 1).

One might like to change the display orientation, here using “%” to access the evaluation immediately prior:

```
In[=]:= Column[%]
Out[=]=
2
6
10
14
18
```

Matrices are similarly a matter of display, of nested lists. A construction using two iterators might look as follows.

D. Exposé: A Tree Pattern Function in Mathematica

78

```
In[=]:= Table[i * j, {i, 1, 3}, {j, 1, 4}] // MatrixForm
Out[=]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \end{pmatrix}$$

```

It is important to note the difference to this expression reversing the order of the iterators.

```
Table[i * j, {j, 1, 4}, {i, 1, 3}] // MatrixForm (* // TableForm *)
Out[=]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \\ 4 & 8 & 12 \end{pmatrix}$$

```

That is, the first iterator determines the number of rows, in the 2D context. Another way to say this is that the inner iterator determines the value of the outer iterator. The mathematical expression being evaluated is, of course, commutative, however.

For measuring lists, the `Length`-Function gives the outermost dimension, whereas `Dimensions[]` gives all dimensions of nested lists. The number of dimensions is given by `ArrayDepth[]`.

For testing and sampling lists the following functions are available. `Position[list, element]` gives the (list of) index positions at which `element` sits, whereas `Select[list, predicate]` tests elements of the list against `predicate`, sampling those elements that satisfy, say, even parity with `EvenQ`. Given a position, elements can be extracted using `Part[]` or its short form in double-square-brackets notation, here as a matrix-example.

```
In[=]:= Table[a_{i,j}, {i, 3}, {j, 3}] // MatrixForm
Out[=]//MatrixForm=

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

In[=]:= %[[2, 1]]
Out[=]=
a_{2,1}
```

`Take[]` is similar to `Part[]` but allows sampling consecutively placed elements in a list.

This section ends with an analysis of the “_”-character to symbolize any other Mathematica object (Wolfram Language expressions), which is useful for patterns in general, and particularly, the `prooid` function at the beginning of this section. The crucial difference is between two underscores, “__” or `BlankSequence`, standing for any sequence of one or more expressions, and three underscores, “___” or `BlankNullSequence`, which also allows zero such expressions.

The following example, taken from Wolfram Research (2023), makes the limitation of `BlankSequence` clear.

```
In[=]:= f[x__] := Length[{x}]
In[=]:= {f[x, y, z], f[]}
Out[=]=
{3, f[]}
```

D. Exposé: A Tree Pattern Function in Mathematica

Contrast this with use of the BlankNullSequence:

79

```
In[1]:= f2[x___] := p[x, x]
In[2]:= {f2[], f2[1], f2[1, a]}
Out[2]= {p[], p[1, 1], p[1, a, 1, a]}
```

Blank ("_") is the most limited pattern character in that it can stand for any Wolfram Language Expression, but only exactly one such expression.

```
In[3]:= f3[x_] := "evaluated"
In[4]:= {f3[], f3[1], f3[1, "a"]}
Out[4]= {f3[], evaluated, f3[1, a]}
```

Functional Programming

With lists there is now a background in place to consider Wolfram Language's functional aspects in more detail. For a definition, I refer to the Mathematica Cookbook (Mangano, 2010, p. 31): "All functional languages emphasize the evaluation of expressions to produce values rather than commands or statements that are executed for their side effects." Since Mathematica also allows for functions like *Do[]*, the language is not purely functional, instead supporting at least also the procedural paradigm. The focus here however is functional programming by a given example, not a comparison of paradigms or more analysis of the functional one.

With this, the following core functional programming "primitives" are summed up in a helpful reference table in (Mangano, 2010, p. 26) and will lead to a discussion of the relative idioms and their application.

Out[107]=

Function	Operator	Description
Map[f, expr]	@/	Return the list that results from executing f on each element of an expr
Apply[f, expr]	@@	Return the result of replacing the head of a list with function f
Apply[f, expr, {1}]	@@@	Applies f at level 1 inside list. In other words, replace the head of all elements
Fold[f, x, {a1, a2, a3}]	N/A	If list has length 0, return x, otherwise return f[f[f[x, a1], a2], a3]...
FoldList[f, x, {a1, a2, a3, ...}]	N/A	Return the list {x, f[x, a1], f[f[x, a1], a2], ...}
Nest[f, expr, n]	N/A	Return the list f[f[f[...f[expr]...]]] (i.e. f applied n times)
NestList[f, expr, n]	N/A	Return the list {x, f[expr], f[f[expr]], ...} where f repeats up to n times

This also covers almost all of the sort-hand notation after blanks, where ampersand ("&") and hash ("#") still need mentioning: this ties in perfectly with the topic of functional operations and goes to

D. Exposé: A Tree Pattern Function in Mathematica
 show that indeed, functional is at the heart of the language, since the short-hand notation offered caters to the paradigm. This example illustrates the usages, in so-called pure functions (lambda expressions), starting with the named-function counter-example.

```
In[14]:= h[x_] := f[x] + g[x]
```

Combine this with *Map[]* in the following way.

```
In[15]:= Map[h, {a, b, c}]
Out[15]= {f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

The pure-function alternative is written as follows.

```
In[16]:= Map[f[#] + g[#] &, {a, b, c}]
Out[16]= {f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

Here ampersand demarcates the end of the function expression, and hash marks the arguments.
 The non-short-hand version is:

```
In[18]:= Map[Function[x, f[x] + g[x]], {a, b, c}]
Out[18]= {f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

I refer to the Functional Operators Tutorial (Wolfram Research, 2023) for the minor details including, for example, the modifications of the hash operator, such as `#n` or `##`. (Mangano, 2010) offer the helpful notion of idioms in this context, where the functions *Map[]* and *Apply[]* that we have already seen present one such idiom, useful for summing sublists: here is the relevant example (p. 33), in short-hand.

```
In[23]:= Plus @@ # & /@ {{1, 2, 3}, {4, 5, 6, 7, 8}, {9, 10, 11, 12}}
Out[23]= {6, 30, 42}
```

Plus is applied and mapped to each element, here each sublist, hence this would be the Map-Apply idiom. The same can be accomplished in other ways.

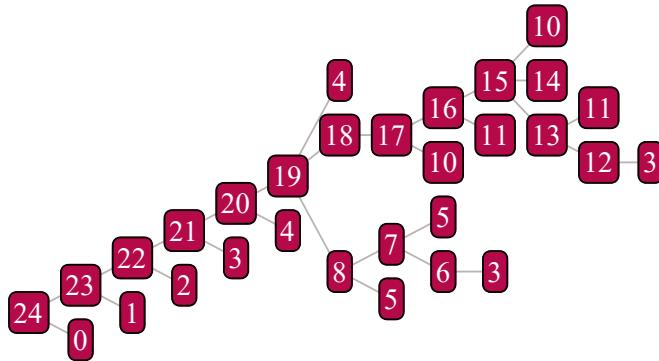
```
In[24]:= Plus @@@ {{1, 2, 3}, {4, 5, 6, 7, 8}, {9, 10, 11, 12}}
Out[24]= {6, 30, 42}
```

(Mangano, 2010) goes on to present further use cases and appropriate idioms in the functional programming chapter of his book. The functional programming elements discussed so far allow us to analyze this part of the code, the third of the three functions, we are investigating:

```
In[25]:= getLeanTree[proof_] := Tree[proofID[proof], getLeanTree /@ subproofs[proof],
  TreeElementLabelStyle → All → Directive[White, 16, FontFamily → "Times New Roman"],
  TreeElementStyle → All → Directive[EdgeForm[Black], RGBColor["#B6094A"]]]
```

There are a lot of options here, concerned with output style. (Something like the following gets generated as the final return value: the presentation is controlled by the options.)

D. Exposé: A Tree Pattern Function in Mathematica



There is also a recursive element here, which we will ignore just for the moment, in fact we want to focus in on just this part:

```
In[106]:= getLeanTree /@ subproofs [proof]
```

The "/@" - Mapping of the (recursively called) function `getLeanTree[]` is the Map - Apply idiom in action, where the function is applied to the output of another function `subproofs[]`, which is a list, with usually two or three elements, behind the scenes .

```
In[106]:= subproofs [leanProof]
```

```
Out[106]=
```

Proofs	Goal	$\forall \{m n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p + m \wedge n \rightarrow p + m$								
	ID	23								
	Rule	$\forall I$								
	▼ Arguments									
<table border="1"> <tbody> <tr> <td>Goal</td><td>$p : \mathbb{N}$</td></tr> <tr> <td>ID</td><td>0</td></tr> <tr> <td>Rule</td><td>Assumption</td></tr> </tbody> </table> ,			Goal	$p : \mathbb{N}$	ID	0	Rule	Assumption		
Goal	$p : \mathbb{N}$									
ID	0									
Rule	Assumption									
<table border="1"> <tbody> <tr> <td>Goal</td><td>$m : \mathbb{N}$</td></tr> <tr> <td>ID</td><td>1</td></tr> <tr> <td>Rule</td><td>Assumption</td></tr> </tbody> </table>			Goal	$m : \mathbb{N}$	ID	1	Rule	Assumption		
Goal	$m : \mathbb{N}$									
ID	1									
Rule	Assumption									
<table border="1"> <tbody> <tr> <td>Goal</td><td>$\forall \{n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p + m \wedge n \rightarrow p + m$</td></tr> <tr> <td>ID</td><td>22</td></tr> <tr> <td>Rule</td><td>$\forall I$</td></tr> <tr> <td>Proofs</td><td>► Arguments</td></tr> </tbody> </table>			Goal	$\forall \{n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p + m \wedge n \rightarrow p + m$	ID	22	Rule	$\forall I$	Proofs	► Arguments
Goal	$\forall \{n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p + m \wedge n \rightarrow p + m$									
ID	22									
Rule	$\forall I$									
Proofs	► Arguments									

Rule-based Programming

The grid structures presented above immediately bring us to pattern-oriented programming in Wolfram Language and we move on to the second function of our initial three to investigate this aspect.

```
In[108]:= subproofs [
  Grid[{___, {Proofs, OpenerView[{Arguments, Column[subproofs_, ___]}, ___]}, ___}, ___},
    ___]] := subproofs;
subproofs [proof_] := {};
```

To clear up a technical detail: the opener view is the collapsable arrow or triangle shape leading to more text display, here in a nested form when viewing in a Mathematica notebook. In a way, the pattern to tree functions are all about this un-collapsing of the grid structure, representing a mathe-

D. Exposé: A Tree Pattern Function in Mathematica
 matical proof in Lean. For completeness, here is another level of the grid expanded and viewable in PDF-file-format to illustrate the functionality. (This structure will become important again once we look at recursion.)

Out[*#*]=

<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>Goal</td><td>$\forall \{m\ n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p \cdot m \wedge n \rightarrow p \cdot m$</td></tr> <tr><td>ID</td><td>23</td></tr> <tr><td>Rule</td><td>$\forall I$</td></tr> </table> $\left\{ \begin{array}{ c c } \hline \text{Goal} & p : \mathbb{N} \\ \hline \text{ID} & 0 \\ \hline \text{Rule} & \text{Assumption} \\ \hline \end{array} \right.,$	Goal	$\forall \{m\ n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p \cdot m \wedge n \rightarrow p \cdot m$	ID	23	Rule	$\forall I$	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr> <td colspan="2" style="padding: 5px;">▼ Arguments</td> </tr> <tr> <td style="width: 10%;">Goal</td> <td>m : \mathbb{N}</td> </tr> <tr> <td>ID</td> <td>1</td> </tr> <tr> <td>Rule</td> <td>Assumption</td> </tr> </table> <table border="1" style="border-collapse: collapse; width: 100%;"> <tr> <td colspan="2" style="padding: 5px;">▼ Arguments</td> </tr> <tr> <td style="width: 10%;">Goal</td> <td>$\forall \{n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p \cdot m \wedge n \rightarrow p \cdot m$</td> </tr> <tr> <td>ID</td> <td>22</td> </tr> <tr> <td>Rule</td> <td>$\forall I$</td> </tr> </table> <table border="1" style="border-collapse: collapse; width: 100%;"> <tr> <td colspan="2" style="padding: 5px;">▼ Arguments</td> </tr> <tr> <td style="width: 10%;">Goal</td> <td>n : \mathbb{N}</td> </tr> <tr> <td>ID</td> <td>2</td> </tr> <tr> <td>Rule</td> <td>Assumption</td> </tr> </table> <table border="1" style="border-collapse: collapse; width: 100%;"> <tr> <td colspan="2" style="padding: 5px;">► Arguments</td> </tr> <tr> <td style="width: 10%;">Goal</td> <td>nat.prime p → p · m ∧ n → p · m</td> </tr> <tr> <td>ID</td> <td>21</td> </tr> <tr> <td>Rule</td> <td>$\forall I$</td> </tr> <tr> <td>Proofs</td> <td></td> </tr> </table>	▼ Arguments		Goal	m : \mathbb{N}	ID	1	Rule	Assumption	▼ Arguments		Goal	$\forall \{n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p \cdot m \wedge n \rightarrow p \cdot m$	ID	22	Rule	$\forall I$	▼ Arguments		Goal	n : \mathbb{N}	ID	2	Rule	Assumption	► Arguments		Goal	nat.prime p → p · m ∧ n → p · m	ID	21	Rule	$\forall I$	Proofs	
Goal	$\forall \{m\ n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p \cdot m \wedge n \rightarrow p \cdot m$																																								
ID	23																																								
Rule	$\forall I$																																								
▼ Arguments																																									
Goal	m : \mathbb{N}																																								
ID	1																																								
Rule	Assumption																																								
▼ Arguments																																									
Goal	$\forall \{n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p \cdot m \wedge n \rightarrow p \cdot m$																																								
ID	22																																								
Rule	$\forall I$																																								
▼ Arguments																																									
Goal	n : \mathbb{N}																																								
ID	2																																								
Rule	Assumption																																								
► Arguments																																									
Goal	nat.prime p → p · m ∧ n → p · m																																								
ID	21																																								
Rule	$\forall I$																																								
Proofs																																									

With this out of the way, rules are the essential building block in this chapter: “Everything that the Wolfram Language does can be thought of as derived from its ability to apply general transformation rules to arbitrary symbolic expressions” (Wolfram Research, 2023). They can be used for string operations like so, where the arrow gives the replacement rule.

In[115]:=

```
StringReplace["Mathematica is multi-paradigm", {"Mathematica" → "Wolfram Language"}]
```

Out[115]=

```
Wolfram Language is multi-paradigm
```

Here is short-hand notation and a useful list item swap rule:

In[117]:=

```
{\{\alpha, 1\}, {\beta, 2\}, {\gamma, 3\}} /. {x_, y_} → {y, x}
```

Out[117]=

```
\{\{1, α\}, {2, β\}, {3, γ\}}
```

At this point we also circle back to the blank-notation for patterns: The above example is taken from (Wellin, p. 150), and to highlight the concept’s importance and its relationship to lists: “Rule-based programming is such a useful construct for manipulating lists and arbitrary expressions that no user of *Mathematica* should be without a working knowledge of this paradigm.” (*Ibid.*)

The following built-in functions help to implement the paradigm as discussed so far: Here a generic test using blanks.

In[118]:=

```
MatchQ["Mathematica", _]
```

Out[118]=

```
True
```

D. Exposé: A Tree Pattern Function in Mathematica

83

And here the pattern matching function already discussed in terms of lists, the first of our three functions, implementing a more complex pattern matching routine.

```
In[120]:= proofID[Grid[{___, {ID, id_}, ___}, ___]] := id;
```

Combine this pattern with the built-on `Cases[]` to extract the matches and compile them to a list, given appropriate input.

```
In[120]:= pattern = Grid[{___, {ID, id_}, ___}, ___];  
In[127]:= Cases[subproofs[leanProof], pattern]
```

Out[127]=

<table border="1"> <tr><td>Goal</td><td>p : \mathbb{N}</td></tr> <tr><td>ID</td><td>0</td></tr> <tr><td>Rule</td><td>Assumption</td></tr> </table>	Goal	p : \mathbb{N}	ID	0	Rule	Assumption	<table border="1"> <tr><td>Goal</td><td>$\forall \{m n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p + m \wedge n \rightarrow p + m$</td></tr> <tr><td>ID</td><td>23</td></tr> <tr><td>Rule</td><td>$\forall I$</td></tr> <tr><td>Proofs</td><td>► Arguments</td></tr> </table>	Goal	$\forall \{m n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p + m \wedge n \rightarrow p + m$	ID	23	Rule	$\forall I$	Proofs	► Arguments	}
Goal	p : \mathbb{N}															
ID	0															
Rule	Assumption															
Goal	$\forall \{m n : \mathbb{N}\}, \text{nat.prime } p \rightarrow p + m \wedge n \rightarrow p + m$															
ID	23															
Rule	$\forall I$															
Proofs	► Arguments															

In this case, `Cases[]` simply returns the input list, because all elements match. A more useful application of cases might be:

```
In[129]:= Cases[{"here is a diverse list", -2, 2.0, 0, -0, -0.0, -0.1}, _?Negative]  
Out[129]= {-2, -0.1}
```

Here the negative-test (a predicate function, instead of a pattern) is applied, where `Negative[]` returns true in the appropriate case and false (or unevaluated, if the predicate is not applicable) in all others.

```
In[132]:= Negative /@ {-2, -0.1}  
Out[132]= {True, True}  
  
In[134]:= Negative /@ {"here is a diverse list", 2.0, 0, -0, -0.0}  
Out[134]= {Negative[here is a diverse list], False, False, False}
```

On the topic of predicates, pure functions can also be used to implement them, here in combination with a test for the head of an element (testing if it is a list).

```
In[139]:= MatchQ[{1, 2, 3}, _List?(Length[#] > 2 &)]  
Out[139]= True
```

Finally, the second of our functions does the same thing as the first, but for an even more complex pattern (again using `BlankSequences` liberally).

```
In[140]:= subproofs[  
  Grid[{___, {Proofs, OpenerView[{Arguments, Column[subproofs_, ___]}, ___]}, ___}, ___], ___], ___] := subproofs;  
subproofs[proof_] := {};
```

D. Exposé: A Tree Pattern Function in Mathematica 84
 But with one caveat of a base case for a later recursion, in the form of an overloaded function (`subproofs[]`) taking any possible pattern and binding it to the variable `proof`, but returning an empty list as a fixed value: this leads to the final topic to be discussed in this analysis, recursion.

Recursion

We now can put the functions together to explain the functionality in terms of recursion, another paradigm implemented in Wolfram Language. This is the set of functions we were considering:

```
proofID[Grid[{___, {ID, id_}, ___}, ___]] := id;
(* basically an extractor-function, using patterns *)

subproofs[Grid[{___,
  {Proofs, OpenerView[{Arguments, Column[subproofs_, ___]}, ___]}, ___}, ___]] :=
  subproofs; (* Same as above, but with more complex patterns *)
subproofs[proof_] := {}; (* Stopping condition preparing the recursion *)

getLeanTree[proof_] := Tree[proofID[proof], getLeanTree /@ subproofs[proof],
  TreeElementLabelStyle → All → Directive[White, 16, FontFamily → "Times New Roman"],
  TreeElementStyle → All → Directive[EdgeForm[Black], RGBColor["#B6094A"]]]
(* Putting it all together now, plus some options for styling the output *)
```

One note on *Proofs* and *Arguments*: these are simply the names of heads in the proof-grid-structure that is the input for the main function, `getLeanTree[]`.

Recursion is of course a call of a function to itself, which is seen here in `getLeanTree[]`, as the function constructs a tree with an integer delivered by `proofID` as the parent and the children as outputs from the recursive call mapped to the proofs returned by `subproofs[]`. Say two children are returned: the next recursion step makes a tree out of the “child proofs” (the “sub-proofs”) and again extracts the ids and makes the recursive call, now implementing the branching of the tree at the second level.

The fact that `subproofs[]` is overloaded to produce an empty list when no relevant pattern is present ensures that in the recursive mapping of `getLeanTree[]` eventually an empty list is mapped to:

```
In[143]:= Tree["id", {}]
Out[143]=
```

`id`

A leaf-node is reached, here corresponding to a mathematical axiom which in turn does not depend on other axioms: The recursion stops, a base case is reached in the sense that a tree with empty children is defined as a leaf, though the actual empty list is returned not by the recursing function (`getLeanTree[]`) but by another (`subproofs[]`).

This section closes with a more canonical example of recursion with a direct base case, the Fibonacci sequence, and an application to lists, showing how lists are once again the essential auxiliary concept.

D. Exposé: A Tree Pattern Function in Mathematica

85

```
In[157]:= f[1] = 1; (* Fibonacci *)
          f[2] = 1;
          f[n_] := f[n - 2] + f[n - 1] /; n > 2
```

It turns out specific rules are looked up before more general rules, which is helpful for implementing recursion, since we can omit the non-base condition after simply defining the base cases.

```
In[163]:= f[n_] := f[n - 2] + f[n - 1]
In[164]:= Table[f[i], {i, 15}]
Out[164]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610}
```

The following helpful function lets us analyze what is being evaluated in what order; this can be especially helpful when it comes to recursion, often implementing many steps in relatively sparse syntax.

```
In[169]:= TracePrint[f[3]]
```

D. Exposé: A Tree Pattern Function in Mathematica 86
 $f[3]$

```

f
3
RuleCondition[$ConditionHold[$ConditionHold[f[3 - 2] + f[3 - 1]]], 3 > 2]
RuleCondition
3 > 2
Greater
3
2
True
RuleCondition[$ConditionHold[$ConditionHold[f[3 - 2] + f[3 - 1]]], True]
$ConditionHold[$ConditionHold[f[3 - 2] + f[3 - 1]]]
$ConditionHold
f[3 - 2] + f[3 - 1]
Plus
f[3 - 2]
f
3 - 2
Plus
3
-2
1
f[1]
1
f[3 - 1]
f
3 - 1
Plus
3
-1
2
f[2]
1
1 + 1
2
Out[169]=
2

```

The final (simple!) list-recursion example echoes `getLeanTree[]` in the way it requires an empty list base case and recursively folds into itself.

D. Exposé: A Tree Pattern Function in Mathematica

87

```
In[170]:= l[list_] := l[Rest[list]] + 1 (* length of a list is rest (tail) + 1 *)
l[{ }] := 0

In[172]:= l[{1, 2, 3, 4, 5}]

Out[172]= 5
```

The observation here is that a simple problem can actually appear more complex, simply because various paradigms and potentially some complex patterns are being mixed at the same time.

Limitations and Summary

This study did not cover numerics, graphics programming, front-end programming, or writing packages, all major topics related to the current discussion: An Introduction to Programming with Mathematica (2005, Wellin et al.) completes the picture at the intended overview level of analysis, of exactly these missing topics. The focus here has been on one particular example, solving one problem, in Wolfram Language; it is involved enough to be viewed in context of the major themes chosen after introducing the language in broad strokes. These themes were lists; functional programming; rule-based programming as the particularly helpful approach in this situation; and recursion, which we could not have easily done without.

References

- Heseltine, J. (2023) *Empirical metamathematics: extending the Lean-to-Mathematica bridge*. Wolfram Community. Available at: <https://community.wolfram.com/groups/-/m/t/2957419> (Accessed: 26 July 2023).
- Mangano, S. (2010) *Mathematica Cookbook*. Sebastopol, CA: O'Reilly Media, Inc.
- Wellin, P. R. (2005) *An Introduction to Programming in Mathematica*. Cambridge, UK: Cambridge University Press.
- Wolfram Research (2023) Wolfram Language & System Documentation Center. Available at: <https://reference.wolfram.com/language/> (Accessed: 23 July 2023).

References

Literature

- [1] *\$Failed—Wolfram Language Documentation*. URL: [https://reference.wolfram.com/language/ref/\\$Failed.html](https://reference.wolfram.com/language/ref/$Failed.html) (visited on 02/28/2024) (cit. on p. 46).
- [2] *American Mathematical Society*. en. URL: <https://www.ams.org/arc/resources/ams-latex-about.html> (visited on 04/02/2024) (cit. on p. 5).
- [3] *Begin—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/Begin.html> (visited on 04/09/2024) (cit. on p. 17).
- [4] *BlankNullSequence—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/BlankNullSequence.html> (visited on 02/24/2024) (cit. on p. 2).
- [5] *Block—Wolfram Language Documentation*. URL: <https://reference.wolframcloud.com/language/ref/Block.html> (visited on 02/24/2024) (cit. on p. 36).
- [6] *BoxData—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/BoxData.html#:~:text=BoxData%20is%20typically%20used%20as,need%20to%20be%20used%20directly.> (visited on 04/02/2024) (cit. on p. 3).
- [7] Bruno Buchberger. ‘Mathematica as a Rewrite Language’ (Nov. 1996) (cit. on pp. 1, 2, 9).
- [8] *Building Large Software Systems in the Wolfram Language—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/tutorial/BuildingLargeSoftwareSystemsInTheWolframLanguage.html> (visited on 04/09/2024) (cit. on pp. 15, 17).
- [9] *Cell > Show Expression—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/menuitem>ShowExpression.html> (visited on 04/02/2024) (cit. on p. 3).
- [10] *Cell—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/Cell.html> (visited on 04/02/2024) (cit. on p. 3).
- [11] *City—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/entity/City.html> (visited on 04/02/2024) (cit. on p. 9).
- [12] *CompoundExpression—Wolfram Language Documentation*. URL: <https://reference.wolframcloud.com/language/ref/CompoundExpression.html> (visited on 02/24/2024) (cit. on pp. 2, 36).

- [13] *Curated Computational Data in the Wolfram Knowledgebase*. en. URL: <https://www.wolframalpha.com> (visited on 04/02/2024) (cit. on p. 9).
- [14] *End—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/End.html> (visited on 04/09/2024) (cit. on p. 17).
- [15] *EndPackage—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/EndPackage.html> (visited on 04/09/2024) (cit. on p. 17).
- [16] *Expression Structure—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/guide/ExpressionStructure.html> (visited on 04/02/2024) (cit. on p. 10).
- [17] *Expressions—Wolfram Language Documentation*. URL: <https://reference.wolfram.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. 46, 47).
- [19] *Find the Underlying Box Structure of a Formatted Expression—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/workflow/FindTheUnderlyingBoxStructureOfAFormattedExpression.html> (visited on 04/02/2024) (cit. on p. 3).
- [20] *FindEquationalProof—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/FindEquationalProof.html> (visited on 04/05/2024) (cit. on p. 12).
- [21] *Functional Operations—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/tutorial/FunctionalOperations.html> (visited on 04/04/2024) (cit. on p. 18).
- [22] G. Mayrhofer, S. Saminger % W. Windsteiger. ‘Theorema’. In: *The Seventeen Provers of the World*. Springer. URL: <https://www.cs.ru.nl/~freek/comparison/comparison.pdf> (visited on 04/05/2024) (cit. on pp. 11–14).
- [23] *Get—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/Get.html> (visited on 04/09/2024) (cit. on p. 17).
- [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. 11).
- [25] *Head—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/Head.html> (visited on 04/02/2024) (cit. on p. 10).
- [26] *High-Level Functions: New in Wolfram Language 12*. en. URL: <https://www.wolfram.com/language/12/built-in-interface-to-unity-game-engine/high-level-functions.html?product=language> (visited on 04/10/2024) (cit. on p. 19).
- [27] <https://www3.risc.jku.at/research/theorema/software/>. URL: <https://www3.risc.jku.at/research/theorema/software/> (visited on 04/05/2024) (cit. on p. 11).

- [28] *IntelliJ IDEA – the Leading Java and Kotlin IDE*. en. URL: <https://www.jetbrains.com/idea/promo/> (visited on 02/24/2024) (cit. on p. 7).
- [29] *Introduction to Dynamic—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/tutorial/IntroductionToDynamic.html> (visited on 04/05/2024) (cit. on pp. 5, 12).
- [30] *LaTeX Project Team*. URL: <https://www.latex-project.org/about/team/> (visited on 04/02/2024) (cit. on pp. 4, 5).
- [31] *LaTeX Project Team*. URL: <https://www.latex-project.org/about/team/#frank-mittelbach> (visited on 04/02/2024) (cit. on p. 4).
- [32] *MakeBoxes—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/MakeBoxes.html> (visited on 07/20/2024) (cit. on p. 31).
- [33] Mircea Marin and Florina Piroi. ‘Rule-Based Programming with Mathematica’. en () (cit. on pp. 19–21).
- [34] *Message—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/Message.html> (visited on 02/24/2024) (cit. on p. 47).
- [35] Michael George. *First order logic, Gödel’s theorem (CS 2800, Spring 2017)*. Aug. 2024. URL: <https://www.cs.cornell.edu/courses/cs2800/2017sp/lectures/lec41-gode.html> (visited on 08/26/2024) (cit. on p. 10).
- [36] Misc. *Quaternion*. en. Page Version ID: 1235436136. July 2024. URL: <https://en.wikipedia.org/w/index.php?title=Quaternion&oldid=1235436136> (visited on 08/26/2024) (cit. on p. 3).
- [37] *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. 36).
- [38] *Namespace Management—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/guide/NamespaceManagement.html> (visited on 04/09/2024) (cit. on p. 17).
- [39] *Needs—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/Needs.html> (visited on 04/09/2024) (cit. on p. 17).
- [40] *Notebook—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/Notebook.html> (visited on 04/02/2024) (cit. on p. 3).
- [41] *Package Development—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/guide/PackageDevelopment.html> (visited on 04/09/2024) (cit. on p. 17).
- [42] *Planet—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/entity/Planet.html> (visited on 04/02/2024) (cit. on p. 9).
- [43] *Procedural Programming—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/guide/ProceduralProgramming.html> (visited on 04/09/2024) (cit. on p. 18).
- [44] *ProofObject—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/ProofObject.html> (visited on 04/05/2024) (cit. on p. 12).

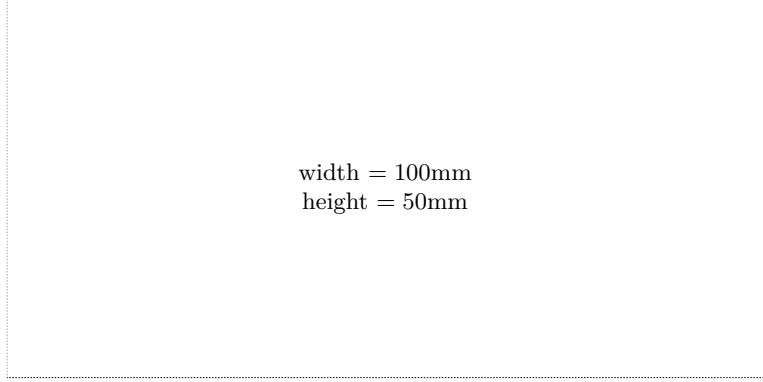
- [45] *Real-Time 3D Development Platform & Editor*. en. URL: <https://unity.com/products/unity-engine> (visited on 04/10/2024) (cit. on p. [19]).
- [46] *Scoping Constructs—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/guide/ScopingConstructs.html> (visited on 04/09/2024) (cit. on p. [17]).
- [47] *Software*. en-US. URL: <https://risc.jku.at/software/> (visited on 04/03/2024) (cit. on p. [1]).
- [48] *Stylesheets—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/guide/Stylesheets.html> (visited on 04/05/2024) (cit. on p. [12]).
- [49] *SUPPRESS THE OUTPUT OF A COMPUTATION—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/howto/SUPPRESSTHEOUTPUTOFAComputation.html> (visited on 04/03/2024) (cit. on p. [2]).
- [50] *TestObject—Wolfram Language Documentation*. Sept. 2024. URL: <https://reference.wolfram.com/language/ref/TestObject.html> (visited on 09/21/2024) (cit. on p. [48]).
- [51] *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. [18]).
- [52] *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. [17]).
- [53] *The Institute*. en-US. URL: <http://www.oeaw.ac.at/ricam/institute/about> (visited on 04/10/2024) (cit. on p. [19]).
- [54] *The Internals of the Wolfram System—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/tutorial/TheInternalsOfTheWolframSystem.html#28134> (visited on 04/09/2024) (cit. on p. [15]).
- [55] *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. [10]).
- [56] *Theorema/Theorema/PackageTemplate.m at master · windsteiger/Theorema*. URL: <https://github.com/windsteiger/Theorema/blob/master/Theorema/PackageTemplate.m> (visited on 04/05/2024) (cit. on p. [17]).
- [57] *What We've Built Is a Computational Language (and That's Very Important!)—Stephen Wolfram Writings*. en. May 2019. URL: <https://writings.stephenwolfram.com/2019/05/what-weve-built-is-a-computational-language-and-thats-very-important/> (visited on 02/24/2024) (cit. on pp. [9], [10]).
- [58] *Why isn't Wolfram more popular?* Reddit Post. Oct. 2022. URL: www.reddit.com/r/math/comments/ye2vj0/why_isnt_wolfram_more_popular/ (visited on 04/02/2024) (cit. on p. [8]).
- [59] *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-open-source-a-dozen-reasons/> (visited on 04/02/2024) (cit. on p. [9]).

- [60] 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.2017.7900016](https://doi.org/10.1109/SYNASC.2017.7900016) (Visited on 04/05/2024) (cit. on p. [11]).
- [61] 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](https://doi.org/10.4204/EPTCS.118.5) (Visited on 10/14/2023) (cit. on pp. [7, 12–15]).
- [62] 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. [8]).
- [63] Wolfram Research, Inc. URL: <https://reference.wolfram.com/legacy/v3/MainBook/2.5.1.html> (visited on 04/03/2024) (cit. on pp. [1–3, 7–9]).
- [64] Wolfram Research, Inc. URL: <https://reference.wolfram.com/legacy/v3/MainBook/2.4.10.html> (visited on 04/04/2024) (cit. on pp. [7, 8]).
- [65] Wolfram Research, Inc. *\$Context—Wolfram Language Documentation*. Aug. 2024. URL: [https://reference.wolfram.com/language/ref/\\$Context.html](https://reference.wolfram.com/language/ref/$Context.html) (visited on 04/09/2024) (cit. on p. [33]).
- [66] 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. [8]).
- [67] Wolfram Research, Inc. *Expressions—Wolfram Language Documentation*. Aug. 2024. URL: <https://reference.wolfram.com/language/tutorial/Expressions.html#20284> (visited on 08/26/2024) (cit. on p. [3]).
- [68] Wolfram Research, Inc. *MakeBoxes—Wolfram Language Documentation*. URL: <https://reference.wolfram.com/language/ref/MakeBoxes.html> (visited on 07/19/2024) (cit. on p. [28]).
- [69] Wolfram Research, Inc. *Mathematica: A System for Doing Mathematics by Computer, Second Edition*. Aug. 2024. URL: <https://www.wolfram.com/books/profile.cgi?id=3617> (visited on 08/26/2024) (cit. on p. [10]).
- [70] 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. [33]).
- [71] Wolfram Research, Inc. *Summary of New and Improved Features in 14.0—Wolfram Language Documentation*. Aug. 2024. URL: <https://reference.wolfram.com/language/guide/SummaryOfNewFeaturesIn140.html> (visited on 02/24/2024) (cit. on p. [8]).
- [72] 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. [48]).
- [73] 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. [48]).

- [74] 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. [48]).
- [75] Wolfram Research, Inc. *Textual Input and Output—Wolfram Language Documentation*. Aug. 2024. URL: <https://reference.wolfram.com/language/tutorial/TextualInputAndOutput.html#12413> (visited on 08/24/2024) (cit. on p. [34]).
- [76] Wolfram Research, Inc. *Transformation Rules and Definitions—Wolfram Language Documentation*. Aug. 2024. URL: <https://reference.wolfram.com/language/tutorial/TransformationRulesAndDefinitions.html#6972> (visited on 08/26/2024) (cit. on p. [3]).
- [77] 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. [49]).
- [78] 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. [48–50]).
- [79] 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. [24]).
- [80] 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. [49]).
- [81] 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. [23]).
- [82] 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. [8]).
- [83] *WolframAlpha Tour*. en. URL: <https://www.wolframalpha.com> (visited on 04/02/2024) (cit. on p. [8]).
- [84] *WolframOne: Cloud-Desktop Computation Platform*. en. URL: <http://www.wolfram.com/wolfram-one/> (visited on 04/02/2024) (cit. on p. [8]).

Check Final Print Size

— Check final print size! —



width = 100mm
height = 50mm

— Remove this page after printing! —