Formal Proving Language (FPL)

A Proposal How To Write and Read Proof-Based Mathematics (PBM)

Andreas Piotrowski

andreas.piotrowski@bookofproofs.org

June 14, 2021

Leibniz's 17th-century vision of a *"Characteristica universalis"*, a precise symbolic language to express human thoughts, will never come true. Such a language is impossible due to Gödel's discoveries in logic made in the 20th century. Nevertheless, people often attempted to develop a language like this, in which at least *"Proof-Based Mathematics"* (PBM) could be expressed in a human-readable form and still be parsed and interpreted by computers. As of 2021, none of these attempts have produced the desired breakthrough to become accepted by a broad mathematical community.

In this document, we analyze the features of PBM and derive critical requirements for a language we call the Formal Proving Language (FPL) that meets these requirements. FPL is a proposal for a human-readable and computer-parseable language expressing PBM. As a proof-of-concept, we also present a possible grammar and vocabulary of FPL and discuss examples of how some given mathematical theories (e.g., basic arithmetics and the theory of sets) could be formulated using FPL.

Contents

1 Goals and Scope of this Document

The scope of this document is **Proof-Based Mathematics** (PBM). As opposed to non-proof-based mathematics that is about calculations and concrete problem solving, PBM is the part of mathematical language to express *deduction*, i.e., it focuses on logical arguments about abstract, idealized, well-defined mathematical objects.[1]

The goal of this document is to analyze the features of PBM and, based on this analysis, to derive requirements for a language to express PBM in a way that facilitates exchanging PBM contents in the domains human-to-human, human-to-computer, computer-to-human, and computer-to-computer. We will call this language the **Formal Proving Language** (FPL). We will also discuss a proof of concept based on an implementation of FPL in `python` that meets this specification.

2 Project and Principles

This document is the high-level design of FPL published under CC BY-SA 4.0, as laid out in the corresponding collaborative project initiative [108], published 14th July 2020. According to this initiative, we are going to align the design with the following design principles:

Principle 1 (Readabilty)

> *Code written in FPL should be human-readable and catchy, memorable, and easy to learn so that FPL can be learned even by non-programmers.*

Principle 2 (Richness of Notation)

> *The syntax of FPL should be inspired by modern mathematical notation while preserving the readability principle.*

Principle 3 (Axiomatic Method)

> *FPL should incorporate the axiomatic method. Every theory written in FPL should start with definitions of mathematical concepts, axioms about these concepts asserting that they are true, and deriving new theorems based on these axioms and logical inference rules.*

Principle 4 (Theory Independence)

> *While using the axiomatic method, FPL should not stick to a pre-defined set of axioms and inference rules. Its meta syntax and semantics should allow developing any theory using the axiomatic method.*

[1] We adopt the term *"proof-based mathematics"* from [32] pp. vii-ix.

Principle 5 (Theory Standardization and Extensibility)

The FPL framework should encourage using some standard set of axioms and inference rules written in FPL to promote normative notation for widely agreed mathematical concepts and a shared sense of mathematical theories. In addition, the FPL syntax should establish an explicit notation and namespaces for these theories to distinguish and re-use widely agreed mathematical concepts.

This principle is not in this document's scope, but we list the principle for completeness reasons. The *"FPL framework"*, in comparison to the specification of *"FPL"* given in this document, comprises a public, open, and freely accessible repository of pre-formulated theories in FPL with commonly agreed axioms and rules of inference, as well as a notation to facilitate re-use. According to [108], the repository will be created under the URL `https://www.bookofproofs.org`.

Principle 6 (Formalism)

The syntax and semantics of FPL should enable the creation of automated aids and tools.

3 Normative References

When we will derive and define requirements for FPL, we will follow [89] best practices by using the following keywords:

1. MUST, REQUIRED, SHALL: These terms mean absolute requirements in this specification.
2. MUST NOT, SHALL NOT: These terms mean absolute prohibitions in this specification.
3. SHOULD, RECOMMENDED: These terms mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
4. SHOULD NOT, NOT RECOMMENDED: These terms mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood, and the case carefully weighed before implementing any behavior described with this label.
5. MAY, OPTIONAL: Optional items of the specification, however, an implementation that does not include a particular option must be prepared to interoperate with another implementation that does include the option and vice versa.

4 Preface

4.1 Vision: FPL as a Common PBM Language for Humans and Computers

In the historical notes of [22] about *"Truth and Provability"*, Hoffmann reminds us that already Gottfried Wilhelm Leibniz (1646 - 1716) envisioned a universal language he called *"Characteristica universalis"*. Leibniz dreamed of a language in which we could formulate all human thinking and knowledge (including but not limited to mathematics) using a precise symbolic notation. Leibniz proposed that by applying a fixed set of rules, he called the *"Calculus ratiocinator"*, it would become possible to derive the *truth* and the *accuracy* of all human thoughts. Leibniz's dream seemed within reach after 200 years in the 19th century when logicians and mathematicians developed the propositional and predicate logic and the first formal systems.[2]

Today, we know that the concepts of *truth* and *provability* are fundamentally separated from each other, making Leibniz's vision a dream that will never come true.[3] Nevertheless, we still ask: *"Could we reach at least some parts of Leibniz's utopic vision?"* We *affirm* this question if we significantly weaken his original vision of *"Characteristica universalis"*. We, therefore, formulate our first two requirements limiting the design of FPL:

Requirement 1 (Scope Limitation to PBM)

> *FPL* MUST NOT *aim to cover all human thinking and knowledge. It* SHOULD *be limited to the mathematical thinking and knowledge, more specifically,* ***self-contained theories*** *in PBM.*

Requirement 2 (FPL Separating Truth and Consistency)

> *FPL* MUST *in some explicit* ***syntax*** *separate the concepts of truth and accuracy.*

At this stage, many terms used above, especially *truth*, *consistency*, *symbolic notation*, and *self-contained theories* are still not explained. We will analyze and explain them in much more detail later.

4.2 Market Need for a Standard of Expressing PBM

Mathematicians have always endeavored to develop a language, notation, and conventions that would enable them to correctly and unambiguously formulate PBM. After millennia of development of mathematics in general and its abstract language PBM in particular, there is a growing market need to have a *standard* in place, in which we could express PBM. Below, we list some key reasons for the market need for such a standard.

[2] [22] p. 3
[3] We will be talking about these limitations in this document later in the necessary detail.

4.2.1 Massive Growth and Diversity of PBM Publications

Today, thousands of mathematical textbooks and papers are published a year using PBM in different natural languages, most of them in English. Mathematics became a discipline growing both in-depth and breadth. Nevertheless, computers still cannot verify the publications for correctness despite available computing power since all these publications follow no standard and structure. Researchers tried to overcome this problem, but no approach, including machine learning, natural languages, or controlled natural languages, successfully created a breakthrough.

⇒ If we had a standard for writing mathematical texts, it would become much easier to verify them using computers.

4.2.2 Unredeemed Promise of Logic and Formal Systems

Most modern mathematicians would agree that formal systems and predicate logic are means to formalize mathematics. However, there are at least two significant drawbacks in formulating PBM this way. First, pure logic and symbolic notation to express mathematics can become very cumbersome. The second drawback is less known but at least as much as serious: The syntax of formal systems fundamentally does not allow formulating mathematical definitions[4], in which the vocabulary of PBM can be increased[5]. On the other hand, PBM is full of definitions. So how likely is it that mathematicians would ever accept a formal system to replace their current language and depriving them of a way to introduce and increase their vocabulary?

⇒ We need a standard of formulating PBM that formalizes the real, unstructured mathematical language, still preserving its advantages.

4.2.3 Lack of Self-Containment of PBM Publications

Mathematical publications are rarely self-contained in the sense that they contain all the axioms and definitions. If at all, the authors of textbooks often do with sentences like *"We assume that the reader is familiar with [...]"*, or briefly reference to other sources. We cannot be sure that these other sources use the same notation, axioms, and definitions in a given publication referencing it. The situation is even worse in mathematical papers. Therefore, hoping that computers will ever be able to verify a given publication formally is illusory.

⇒ If mathematical publications followed a standard for PBM, that standard could address the issue of self-containment in a manner that verifying these texts becomes tractable by computers.

[4] [22], pp. 71-82
[5] [9] p. 19

4.2.4 The Foundations Pitfall

PBM, like every language, has developed over centuries and is still developing. It is, therefore, illusory to expect that 20th-century foundations of mathematics, like the *Peano axioms* or the *Zermelo-Fraenkel set theory*, to last forever. Most contemporary approaches to formalizing PBM *incorporate* these foundations rather than are *independent* from them. While the formal language is trying to express mathematics, it generates the risk of remaining trapped in its syntax and semantics.

⇒ We need a standard of expressing PBM that is independent from mathematical foundations, allowing us to read and write PBM at a meta level.

Notably, the point is the *independence* of any mathematics foundations, not replacing them with some new foundations. We will demonstrate later why this is very important for FPL.

4.2.5 Mathematical Education

Teachers often teach elementary and popular level mathematics focussing on concrete problem-solving rather than on PBM. The result is that only a tiny percentage of pupils ever learn basic rules of logical thinking.

⇒ If a standard for PBM existed, it could be taught like a foreign language, increasing the overall competence in logical thinking.

⇒ Moreover, a standard could save a lot of prose text (and time) otherwise necessary for reading and writing lecture scripts.

4.2.6 International Cooperation and Research

Cooperation and research use PBM expressed in natural language, mixed with symbolic notation. Researchers are (mostly) forced to use the English language on an international level, including mathematical terminology.

⇒ If a standard to express PBM existed, it would be independent of existing natural languages and thus further facilitate international cooperation and research in mathematics.

4.3 Related Approaches and Work

4.3.1 The QED Manifesto

In 1994, in the spirit of the *"Bourbaki series"*, an anonymous group of experts called out for cooperation on the *"QED"* project[6]. They proposed *"to build a computer system that effectively represents all essential mathematical knowledge and techniques [...] including the use of strict formality in the internal representation of knowledge and the use*

[6] [90]

of mechanical methods to check proofs of the correctness [...]." Bourbaki acknowledged the complexity of the undertaking, still proposing a four-step approach, motivated the project, anticipated possible objections, and related it to *Artificial Intelligence* (AI) and *Automated Reasoning* (AR).

Readers can regard this specification as a small contribution to QED. However, we neither aim to *build a specific knowledge system* nor do we believe that mathematicians could ever agree upon some common canonical representation of their knowledge. Maybe the most significant overlap between FPL and QED is the need for what Bourbaki calls *"root logic"*. Bourbaki explained this need as *"agreeable to all practicing mathematicians"* and as *"sufficiently strong to check any explicit computation"*. In this document, we aim to investigate and specify FPL as a language to anchor such a *"root logic"*.

4.3.2 Approaches Centered on (Controlled) Natural Languages

The Naproche project[7] and the doctoral thesis *"The Language of Mathematics"* of Ganesalingam[8] follow a similar vision to the vision formulated in 4.1, calling it a *"long-term project to construct a computer language for expressing pure mathematics in a way that was as close as possible to real mathematics"*. While Ganesalingam provides a framework for such a computer language, the Naproche project aims to develop a concrete computer programm. Both aim to

1. Parse a given mathematical textbook or publication written in natural English language ([9]) or in a *controlled natural language* ([71]) that encompasses both, prose and symbolic notation,
2. Overcome unavoidable *ambiguities*,
3. Extract and represent the *semantics*.[9]

The Naproche project goes even further:

4. Check the *correctness*[10]

Despite all similarities, the development of FPL is different in a critical aspect: FPL is not a computer language accepting the unstructured way how mathematicians express PBM in prose mixed with symbolic notation. Instead of overcoming all difficulties connected with disambiguation and semantics representation, we aim to reach these goals more efficiently and effectively by *expressing* PBM in a structured way, possibly avoiding ambiguities, and incorporating ways to represent semantics. Another difference is that we are (at least more explicitly) cautious about our total disability of creating a computer that can validate at once the *correctness* of mathematical proofs and the *truth* of the corresponding semantic representation.

[7] [71] p. 1
[8] [9] p. X
[9] [9] pp. XI-XII, [71] p. 6
[10] [71] p. 6

4.3.3 Automated Deduction Systems

Several specialized computer languages and solutions, both free and commercial, provide automated expert help in specialized domains. Such tools range from *"computer algebra systems, automated deduction systems, modelling systems, matrix packages, numerical prototyping languages, decision trees for scientific computing software, etc."*[11]. In particular, the *automated deduction systems* (like Coq[12], Isabelle[13], or Mizar[14]) are tools enabling users to encode mathematical statements and either derive or validate their proofs.

At first glance, these systems fit well into our vision of FPL. However, we can observe that they are still not the method of choice for the working mathematicians. In [75], an analysis of these tools was performed, advising on how to improve *user acceptance*. In our analysis, user acceptance of FPL will remain one of the main goals we want to achieve.

4.3.4 Formal Mathematical Language (FMathL)

Maybe the most similar approach to FPL has been undertaken at the University of Vienna in the Formal Mathematical Language (FMathL). Neumaier[15] describes it as *"the working title for a modeling and documentation language for mathematics, suited to the habits of mathematicians"*. Readers can find technical details and design principles of FMathL in [85]. As we will be doing in our analysis for FPL, Neumaier addresses the question of whether to *base* FMathL *on* some existing foundations of mathematics. He demonstrates how current foundations[16] are *incompatible*. He therefore proposes new foundations of mathematics containing 27 axioms[17]. This axiomatic approach is the result of an important analytical result presented by Neumaier: the distinction between the *"subject and object levels"*[18] of mathematical objects. According to Neumaier, subjects (in our case, both humans or computers) would never be able to access the objects on their object level. They only had access to their very own subject-level objects.[19] We will come to similar conclusions in our later analysis[20]

There is a risk of perceiving our intended specification of FPL as very similar to the work done in FMathL. However, FPL is going to differ from FMathL by design fundamentally. In our analysis, we will challenge *any* approach to a language for expressing

[11] [86]
[12] [69]
[13] [80]
[14] [83]
[15] [84], retrieved on August 23, 2020
[16] such as *"Zermelo-Fraenkel set theory [...], illative combinatory logic, [...] and CCAF (the category of categories)."* [85], 10ff.
[17] [85] 56ff.
[18] [85], pp. 19-24
[19] [85] p. 29: *"Objects themselves are not accessible directly in mathematical discourse. This is necessary since different subjects may have completely different implementations of the 'same' object."*
[20] compare **Assertion** 2 or **Limitation** 5.

PBM that would itself be based on *axioms*. We will then conclude that *any* axiomatic approach (like FMathL's axiomatic approach) may already be *a too strong corset* to mathematics and its language. With this respect, we hope that our work and FPL could constructively contribute to the further development of FMathL.

4.3.5 Differences Regarding PBM features

The features of PBM identified in this document will sometimes significantly deviate from the features identified in other sources, in particular:

1. In our analysis, we often confront different perspectives, a linguistic, mathematical, logical, philosophical, psychological, computer-scientific, or natural-scientific. This synoptic approach will sometimes reveal controversial views of experts. In these cases, we will need to decide which views we want to follow. Occasionally, we gain novel insights.

2. Most of the available sources base their analysis on contemporary mathematics. We are going to analyze the mutability of mathematics over time in its historical context.

3. Sometimes, we will be confronted with deviating terminology across different sources so that we have to stick to a single one. We will highlight which definition we prefer.

4.3.6 Acknowledgements

I dedicate this document to my math professor, Wolfgang Schwarz († 2013), who opened my eyes to the beauty of mathematics and wakened my interest in this subject at the Johann-Wolfgang-Goethe University in Frankfurt am Main (Germany). Moreover, I could not have written this document without the insights and genius of other people. I thank all mathematicians, logicians, philosophers, linguists, physicians, and other scientists, both listed as the authors of the sources used in this document and generations of their teachers and mentors.

4.4 Scientific Approach

4.4.1 Analysis of 'Relevant Features of PBM?'

Before we can even think about how FPL might look like, we have first to identify and better understand the language features of PBM. We will do this in the section *5 Identifying and Understanding the Features of PBM*. Each identified feature of PBM will start with a short objective, followed by an analysis and mostly some conclusions. In the conclusion part, we might list up to three types of different conclusions:

- *Observations* of phenomena that can be *verified* (at least) using the sample of references to this document,

- *Requirements* we derive for the specification of FPL,

- *Assertions* of which we think they are helpful *working hypotheses* to create the specification of FPL.
- *Limitations* of which we think build *obstacles* to create the specification of FPL.

All observations, requirements, and assertions will be *numbered* for the sake of easy reference later in the text.
We will make special mention in cases where we think the identified phenomena and observations made are *novel* concerning the references used in this document.

4.4.2 Synthesis in a 'High-Level Design of FPL'

In *6 High-Level Design of the FPL Standard*, we will recall all our observations, requirements, assertions, and limits. Taking the limits into account, we will conclude a possibly high-level design of FPL. Some of the limits are fundamental and not tractable. Nevertheless, some will turn to be only technical so that we can overcome them with a due design.

4.4.3 Proof of Concept

In *7 Proof of Concept (PoC)*, we will present an example parser implementation of FPL based on `python` and present how existing mathematical theories can be expressed using FPL code. Based on these real-life use cases, we will demonstrate the *"look and feel"* of the FPL language implemented according to the high-level design specification. The use cases should also help the interested readers to have a first assessment of the flexibility, scalability, usability in the domains human-to-human, human-to-computer, computer-to-human (, and possibly even computer-to-computer), as well as the feasibility of implementing possible FPL parsers and interpreters.

4.5 Disclaimer

We do not claim this document to be complete and correct regarding the identified features and analysis performed. This document is provided AS IS and WITHOUT WARRANTY.

4.6 Copyright

[21] [68]

5 Identifying and Understanding the Features of PBM

5.1 What Is PBM?

5.1.1 Objective

The question if there are *discriminant criteria* distinguishing the language of PBM from other languages is crucial for the development of FPL since FPL is going to express PBM. Therefore, we are going to start with the criteria we can find in other sources, which are (said to be) *characteristic* to the language of PBM. We will also check if these criteria are discriminant enough to build a basis for the development of FPL.

5.1.2 Analysis

To our best knowledge there are only a few sources that try to actually characterize the language of PBM (we present [9], [71], [88], [32], [23], [42], [70], [31], [15], [20], [18], [8], and [36]). Moreover, the term *"proof-based mathematics"* is only used in [32].[22] Other source use different paraphases but mean (we hope!) the same subset of mathematical language, for instance *"pure mathematics"*[23], *"advanced mathematics"*[24], *formal register of mathematical discourse*[25], or just *"mathematics"*[26].

The **Table** 1 lists 12 criteria mentioned in the sources to be characteristic for the language of PBM.

Next, we will test how discriminant these criteria are when applied to real examples of mathematical publications. As real-world examples, we pick the two publications [113], a paper dealing with efficient computation of the Euler-Kronecker constant (published 2019), and [114], a paper about the largest known twin primes and Sophie Germain primes (published 1999). Of course, this sample is by far not representative. The point is that they are already sufficient because they are counterexamples showing that the above criteria are either not discriminant enough or unsuitable for our purposes for other reasons.

1. **Mixing natural language and symbols**: Applied to our sample, the criterion is not discriminant enough since not all pure PBM text passages mix text and symbol[27]. Moreover, there are also non-mathematical texts fulfilling this criterion, for instance from chemistry or engineering.

[22] In fact, we have incorporated this term from this source and abbreviate it by PBM.
[23] [9] p. 1
[24] [32] p. 51
[25] [70] p. 3
[26] [23] p. 1
[27] Counterexample from [113], p. 6 *"Unfortunately in the scripting language of PARI/Gp the DFT-functions work only if* $q = 2^l + 1$, *for some* $l \in \mathbb{N}$."; counterexample from [114], p. 1 *"Some days before he found the largest known Sophie Germain prime, with 5089 decimal digits (see [8], p. 330), beating our records* $157324389 \cdot 2^{16352} - 1$ *and* $470943129 \cdot 2^{16352} - 1$ *with* 4931 *and* 4932 *decimal digits, respectively [...]"*

No.	Criterion Description	Sources
1	Mixing natural language and symbols	[9] pp. 17-19,[71] p. 4
2	Avoidance of constructions that are hard to disambiguate	[71] p. 4
3	Specific symbolic notation	[88] p. 3, [9] pp. 25-32 also adopted in [71]
4	Explicit structure in form of separately typeset *"building blocks"* (e.g. definitions, theorems, or proofs).	[71] p. 4, [9] pp. 32-34, [32] p. 51, [23] pp. 1-2
5	Axiomatic (deductive) method: Building blocks called *"axioms"* are presumed. All other statements are derived by logical inference.	[42] p. 21, [20] pp. 14-18, [15] pp. 192-206, [23] pp. 2,7, [36] pp. 5-9, [18] pp. 32-35
6	*Formal* vs. *material* modes of speech	[8] pp 10, 285, [70] pp. 3-4
7	Besides pure deduction, there is also abduction and induction in PBM.	[70] pp. 3-4, [31] p. 291
8	Assumptions (in proofs) can be introduced and retracted.	[71] p. 4
9	Proof steps are justified by referring to previous text passages.	[71] p. 4
10	Diverse specific linguistic features of PBM	[9] pp. 21-25
11	Adaptivity: The vocabulary, notation and semantics can be derived from definitions.	[9] pp. 19, 27; [71] p. 4
12	Self-Containment	[9] p. 5

Table 1: Possible Candidates For PBM Discriminants

2. **Avoidance of constructions that are hard to disambiguate**: This criterion cannot be applied to our sample at all since we (and a computer parsing a publication) can never be sure which *"constructions"* the author of a mathematical publication as avoided. Besides, even the disambiguation of pure PBM publications is a big issue.[28]. Therefore, this criterion is not a promising candidate for a discriminant criterion.

3. **Specific Symbolic Notation**: Note that this is not a single criterion but a whole bunch of notational characteristics investigated in the sources [88], [9], and [71]. A computer parsing our sample and syntactically applying this criterion is likely to accept the sample as PBM. In addition to the counterexamples that we found above, sources referencing this criterion deal with the *contemporary* symbolic notation of PBM only. As we will see later in 5.12, the presumed symbolic notation is not at all *specific* for PBM in its natural development as a language. For this reason, this criterion is also of little use for our purpose.

4. **Separate Building Blocks**: This criterion is not applicable [113] in our sample since this publication does not contain separate building blocks like *definitions*, *theorems*, *proofs*, or *axioms*. If at all, they are stated inline[29]. In [114] p. 1329, there is only one separate building block named *conjecture*. As we will see later, only one of four sources dealing with building blocks lists conjectures explicitly as a building block of PBM.[30] Thus, different sources disagree on the view if a conjecture is a valid building block of PBM or not.

5. **Axiomatic (deductive) method**: This criterion does not apply to our sample since there are no axioms mentioned in this sample. As with conjectures, the sources [9], and [71] do not list axioms among other building blocks. Nor do these two sources address the *axiomatic method* as such.[31] Therefore, different sources disagree on the view if axioms are valid building blocks of PBM or not and whether the axiomatic method is characteristic for PBM or not.

6. ***Formal* vs. *material* modes of speech**: The criterion was proposed by Carnap [8][32]. [70] uses the terms *formal* and *informal* modes of speech. The criterion

[28] compare [9] pp. 87-111, but also in disambiguation techniques found in [71] pp. 8, 9, 11, 24

[29] For instance like in *"we define the Euler-Kronecker constant for the cyclotomic field* $\mathbb{Q}(\zeta_q)$ *as [...]"* [113] p. 1

[30] Only the source [23] lists conjectures as a building block of PBM. The sources [71], [9], and [32] do not.

[31] Nevertheless, in [71] p. 7, the way is described how the Naproche system deals with axioms in the input text; in [9], Ganesalingam notes on p. 182 that there is *"collection of axioms which underpins mathematics"*.

[32] According to [8] p. 10, *"a theory, a rule, a definition, or the like is to be called* formal *when no reference is made in it either to the meaning of the symbols (for example, the words) or to the sense of the expressions (e.g., the sentences), but simply and solely to the kinds and order of the symbols constituting the expressions."* This formal syntax is opposed to *"those logical sentences which assert something about the* meaning, content, or sense *of sentences or linguistic expressions of any domain."* ([8] p. 285)

applies to our sample. In both publications [113] and [114], there is a mix of text passages containing formal statements, logical arguments, or definitions (*formal mode of speach*) with other text passages explaining them informally (*material mode of speech*). Notably, both publications are neither entirely written in the formal nor in the material mode. We would need additional criteria to recognize which parts of these publications are formal and which are material. Thus, programming a computer applying this criterion would require additional techniques.

7. **Deduction** vs. abduction and induction: This criterion means that PBM uses the deductive (axiomatic) method. However, it also contains informal arguments, conjectures, analogies, examples, or remarks necessary to understand the purely deductive blocks of definitions, axioms, theorems, and proofs. We will be talking about these abductive and inductive arguments much much more detail later. For the time being, we only note that this criterion applies to our sample. [33] Thus, the criterion is applicable. However, we cannot answer if it is discriminant since our sample is too small. Indeed, it would need a more detailed investigation to program a computer that applies this criterion when parsing PBM publications.

8. **Assumptions (in proofs) can be introduced and retracted**: [71] proposes this criterion noting that only in PBM texts we can observe the phenomenon of special use of notions like *"suppose"*, *"conversely"*, etc. These notions introduce assumptions about mathematical variables but are revoked later in the text, especially in so-called *proofs by contradiction*, about which we will be talking about later. It may be that this phenomenon is characteristic to PBM, but it is unfortunately not applicable to our sample since there are no proofs by contradiction in it.

9. **Proof steps are justified by referring to previous text passages**: Also [71] proposes this criterion as characteristic to PBM. It applies to [113] of our sample. In [114], there are no proofs in the previous text. However, the source references proofs in other sources.

10. **Diverse specific linguistic features of PBM**. This criterion was proposed in [9] and titled there as *"textual mathematics"*. It encompasses a list of different linguistic features that may contrast mathematical texts from other texts. As an example, such a feature might be the use of the first person plural ('we') *"typically to refer to the mutual intent of the author and reader"*[34]. When applied to our sample, the personal pronoun *"we"* is indeed used correspondingly.

[33] An instance of induction in [114] p. 1319: *"Using the strong probabilistic primality test, the candidates p were tested. [...] Altogether, 182488 candidates were tested. We found 597 probable primes."*; an instance of abduction (conjecture) in [114] (p. 1320): *"it is reasonable to state that the probability that $f_1(n), \ldots, f_s(n)$ are simultaneously prime is [...]"*; an instance of induction in [113] p. 7: *"This let us to evaluate the Euler-Kronecker constants for larger 'good' candidates q (in the sense that their measures using the greedy sequence of prime offsets were larger than* 1.2*)."*

[34] [9] p. 21

11. [9] and [71] propose both **adaptivity** as a characteristic criterion for PBM. However, this criterion is problematic since it is subjective rather than objective. For instance, Ganesalingam defines it as follows: Adaptivity is the *"phenomenon, by which the grammar of an individual mathematician changes as definitions are encountered."*[35]. It *"occurs when certain mathematical statements, known as definitions, are read."*[36]. [71] describes adaptivity as *"Definitions add new symbols and expressions to the vocabulary and fix their meaning."*[37] As we will see later, this second description also depends on the subjectivity (of mathematicians reading or computers parsing the text) because it depends on the individual *interpretation* of definitions. However, both sources consider adaptivity as a major difference to natural languages[38], there are counterexamples from legal texts or ISO standard texts, in which both definitions and their adaptivity occur. Even in natural languages, adaptivity might occur when we look up unknown terms in lexicons or glossaries. Therefore, adaptivity is not a discriminant criterion for PBM, and we cannot consider it an objective feature of PBM because it is subjective by definition.

12. **Self-Containment**: In our sample, the publications are not self-contained because they contain references to other documents. However, we should note that [9] does not provide a sufficiently precise definition to apply this criterion to our sample. Ganesalingam intends to create a framework that applies to particular mathematical *"textbooks and papers"*[39]. However, he bases his framework on a feature he attributes to the whole *"language of mathematics (which is) is completely self-contained [...] It is this property that allowed us to even formulate our goal of describing mathematics using a fully adaptive theory."*[40] The term *"self-containment"* is not defined more precisely in his framework. However, Ganesalingam proposes that, given the language of mathematics being self-contained, it was possible to *"adaptively"* extract the meaning of all mathematical terms and lexemes from definitions read by a mathematician (or parsed and interpreted by a computer).

5.1.3 Summary and Conclusions

The results of our analysis above let us make the following observations:

Observation 1 (Deviating Expert Views on PBM Features)

> *There is no "canonical" expert opinion on which features characterize PBM. Experts may even deviate in which features they consider key for PBM (e.g., axiomatic method vs. self-containment).*

[35] [9] p. 19
[36] [9] p. 27
[37] [71] p. 4
[38] [9] p. 17: *"(Adaptivity is) the most important way in which the language of mathematics differs from natural languages."*; [71] p. 3: *"Some of the features mentioned are also found in general language but are much more prevalent in the language of mathematics (which is) [...] adaptive."*
[39] [9] p. 1
[40] [9] p. 5

Observation 2 (Insufficient Discriminant Criteria)

> *Most of the above criteria characterize contemporary PBM but are not unique for this language since counterexamples fulfill these criteria that are not in the domain of PBM.*

The lack of a canonical, consistent expert view on what characterizes the PBM language and the lack of discriminant criteria differentiating PBM texts from other texts impressively demonstrate that building a computer that can parse contemporary mathematical textbooks and papers and recognize text passages belonging to PBM is not a promising approach. Therefore, these two observations lead us to another major requirement regarding FPL:

Requirement 3 (Structured PBM in FPL)

> *Recognizing PBM text passages* SHALL *be based on expressing them in a structured way in FPL rather than based on teaching computers how to distinguish them from non-PBM text passages. Thus, instead of mimicking the way PBM is expressed in natural languages, FPL* MUST *be distinct from any natural language to clearly separate its code from text passages written in prose.*

Moreover, in [9] the phenomena of *self-containement* and *adaptivity* are described in the limited sense applicable to the semantics and the notation of mathematical terms. Indeed, there are examples of this in our sample.[41] However, self-containment of PBM also applies to logical inference[42]. This leads us to the following novel observation:

Observation 3 ('Self-Containment' and 'Adaptivity' of Logic)

> *The concepts of self-containment and adaptivity are not limited to complementing a lexicon of semantics and notation of PBM. They also apply to complement all the necessary logical arguments of PBM texts.*

Because of this broader sense, self-containment will become so important that we will require all FPL interpreters not to accept a text that is not *"self-contained"* in some precise sense as an PBM instance. We cannot answer the question, whether this is feasible. However, we make the following assertion for the high-level design of FPL:

Assertion 1 (Automated 'Self-Containment' Checks)

> *We assert that we can use a computer-aided process of adaptivity as a technical means to verify the self-containment of texts written in FPL.*

[41] On page 4 in [113], the authors do not define *"even Dirichlet characters"* but use a reference to *"Definition 2.2.25 of Cohen [2]"* instead.

[42] For instance, on page 5 in [113], the authors provide the following logical argument: The *"[...] Euler-Kronecker constant [...] is directly connected with the S-function at a/q since, according to eq. (10) of Moree [11] and (11), we have [...]"*. Thus, there is a reference to a previous logical argument numbered *"(11)"* in the same publication and another logical argument provided in a separate publication *"eq. (10) of Moree [11]"*.

5.1.4 New Open Questions

Our above investigation reveals new open questions. It seems that the criteria four (separate building blocks), five (axiomatic method), and 12 (self-containment) are strongly connected: Which building blocks occur in PBM seems to depend on the axiomatic method. The **Observation** 3 shows us that the axiomatic method interacts with whether or not a mathematical text is *"self-contained"*. In particular, we have to address the following questions:

1. Since not all experts count axioms as building blocks of PBM, we have to elaborate our answer to this question.
2. Is PBM a purely deductive language so that we need only building blocks for the axiomatic method, or do we also need something else?
3. What does it mean in PBM to be *"self-contained"*?

In the next subsection 5.2, we will address the first two issues. The third one is more complex, and we need a lot of preparation, and we will be able to address it not before 5.16.

5.2 Building Blocks of PBM in Literature

5.2.1 Objective

We saw above that many sources describe PBM texts and publications as highly structured texts. Besides optional prose texts, they often include particular typeset blocks of texts, especially definitions, theorems, and proofs. In this subsection, we will be trying to answer the following questions:

- Which are the building blocks of the language of PBM?
- What is the purpose of these building blocks?

5.2.2 Analysis

The phenomenon of structuring PBM in building blocks (4th criterion in 5.1.2) was described in [71], [9], [32] and [23]. In the **Table** 2 we present the building blocks mentioned in these sources:

1. **Definitions** are consistently recognized as building blocks of PBM with similar explanations in all four sources. Accordingly, definitions provide *unambiguous meaning* and *notation* to mathematical terms[43]. One source notes that their order in the text is important.[44]

[43] [71] p. 4; [9] p. 5; [32] p. 51, [23] p. 1
[44] [9] p. 5: *"We can only refer to a term [...] after the appropriate definition has been encountered."*

Building Blocks	[71]	[9]	[23]	[32]
Definitions	p. 4	pp. 5, 33	p. 51	p. 1
Theorems	*"Theorem-Proof Block"* pp. 4, 6-7	*"Results"* p. 33	p. 51	p. 2
Lemmas			p. 51	p. 2
Propositions			p. 51	p. 2
Corollaries			p. 51	p. 2
Proofs	*"Theorem-Proof Block"* pp. 4, 6-7	p. 33	p. 51	p. 2
Axiom				p. 2
Conjecture				p. 2

Table 2: Possible Building Blocks of PBM

2. **Theorems** are recognized as building blocks of PBM in all four sources. However, only [32] explains the difference between *"theorems"* and *"propositions"*, *"lemmas"*, and *"corollaries"*. Accordingly, while all these four building blocks are *"true statements relating the notions defined via definitions"*, theorems are the most significant ones.[45] [71] summarizes *"theorems"*, *"lemmas"*, *"corollaries"* and *"proofs"* to what they call a *"theorem-proof block"*, while *"propositions"* and *"corollaries"* are omitted.[46]; [9] summarizes *"theorems"*, *"lemmas"*, *"propositions"*, and *"corollaries"* to the term *"results"*; results can be named *"if they are famous or important"*.[47]

3. Three of four sources list **propositions** as building blocks. According to [32], propositions are *"of lesser significance"* as compared to theorems but *"of interest of its own"*[48]. In [71], the term *"proposition"* has another meaning, referring to a logical statement.[49]

4. **Lemmas** are consistently recognized as building blocks of PBM in all four sources. Accoriding to [32], a lemma *"is a statement that is often a technical step towards proving a theorem, and is often of questionable interest of its own."*[50].

5. Except of [71], three of four sources account **corollaries** for building blocks of PBM. Only two sources require a proof for a corollary ([9] and [23]), while [32]

[45] [32] p. 51
[46] [71] pp. 4, 6-7
[47] [9] p. 33
[48] [32] p. 51
[49] [71] p. 18: *"If the definiendum and definiens represent propositions, they are equated by a biimplication ('iff' or 'if and only if')."*
[50] [32] p. 51

considers a proof for a corollary optional since it is *"an easy consequence of the previous statement"*[51].

6. All four sources agree that a **proof** is a building block that cannot occur alone and that it occurs immediately *after* a theorem (respectively a proposition, a lemma, or a corollary). A proof consists of *"clear, convincing arguments showing the unquestionable validity"*[52] of the proceeding statement (i.e., a theorem, a proposition, a lemma, or a corollary).

7. Only one of four sources ([23]) mentions **axioms** when introducing the building blocks of PBM. Accordingly, axioms are logical statements that are *asserted* (i.e., lack a proof) and that *complement the meaning* of some basic mathematical objects that definitions cannot describe.[53]

8. Only one of four sources ([23]) mentions **conjectures** when introducing the building blocks of PBM. According to this source, a conjecture is a theorem whose proof has not yet been established[54].

These are surprising results. Once again, we can observe that different sources we picked because they explicitly aim to *describe the structure* of the language of PBM disagree in some key issues about how this language is structured. One major surprise is that three of four sources ([71], [9], and [32]) do not mention axioms when presenting the building blocks of the language of PBM. They still recognize the importance of axioms for PBM when talking about them in other contexts.[55] Another key issue is that all four sources use key terminology differently when talking about PBM as a language, for instance:

1. [71] uses the term *"sentence"* in contexts which other sources would use the term *"statement"*[56] as well as in its general, linguistic sence[57]. In [23], the term *"statement"* is used in the logical sense of propositional or predicate logics, i.e. a statement can be assigned a truth value[58]. Although it is not possible to assign a truth

[51] [32] p. 51
[52] [32] p. 51
[53] [23] p. 2: Axioms are *"statements that you accept as the guiding rules for how your mathematical objects behave and go beyond the definitions to describe and make precise just what the definitions are talking about."*
[54] [23] p. 2: A conjecture is *"a statement whose truth has not yet been determined."*
[55] In [71] p. 7: *"texts and text fragments without definitions and theorem-proof blocks [... may contain a ...] concatenation of sentences [...] usually rendered by conjoining their respective translations with ∧. A special case are axioms and local assumptions."*; in [9], axioms are mentioned the first time on p. 182: *"This is troubling from an epistemological perspective, as one does not want to expand the collection of axioms which underpins mathematics."*; in [32] axioms are mentioned on p. 2 *"to give meaning to primitive notions"*. They are, however (for some unclear reason) omitted on p. 51, on which the author introduces all building blocks of PBM.
[56] for instance [71] p. 8: *"This **sentence** can be read as the conjunction of 'A does not contain an integer k such that $k^2 > 4$' and 'A is finite'"*
[57] for instance [71] p. 7: *"texts and text fragments without definitions and theorem-proof blocks [... may contain a ...] concatenation of **sentences**."*
[58] [23] p. 1: *"A **statement** in Mathematics is just a sentence which could be designated as true or false."*

value to mathematical definitions, the author uses the term *"statement"* also when talking about *"definitions"*[59]. Similarly, [9] introduces the term *"definition statement"* to denote definitions that are stated inline, as opposed to *"definition blocks"* denoting definitions that are explicitly typeset.[60]

2. In [32], the term *"proposition"* is used in the logical sense (like *"statement"* was used in [23])[61]. With this respect, a *"proposition"* is used as an umbrella term for all building blocks which we can assign a truth value (e.g. *"theorem"*, *"lemma"*, *"corollary"*, *"proposition"* or even *"axiom"* in other sources). In some sources and contexts, the term *"proposition"* is used as a name of a special building block (like for instance in [23]). [9] uses the term *"result"* as an umbrella term for statements we can assign a truth value.[62]

3. [32] uses the term *"law"* synonymously to what *logical equivalence* denotes in propositional logic[63]. In [23], a *"law"* is used synonymously to *"theorems"*, *"lemmas"*, *"corollaries"*, or *"propositions"*.[64]

4. In [23], the term *"conjecture"* is a sepate building block of PBM; in [71] this word denotes any to be proven statement we can encounter in mathematical texts[65].

5.2.3 Conclusions

The above analysis results lead us to the following observation:

Observation 4 (Inconsistent Descriptions of the Structure of PBM)

> *Sources dealing with PBM as a language consistently recognize its structure blocks (like definitions, theorems, proofs.). However, some significant differences might exist regarding which blocks constitute the linguistic structure of PBM and which do not, or regarding crucial terminology.*

Unfortunately, while we were to some extent able to answer the question about the purpose of some building blocks, we still do not have a definitive answer to the question *"which are these building blocks"*?

[59] [23] p. 1: *"Mathematicians know from experience that [...] you better start with **definitions**, that is, you better make same clear **statements** about the objects you are about to study [...]"*
[60] [9] p. 34: *"We will, however, refer to **definition blocks** and **definition statements** where there is a risk of confusion."*
[61] [32] p. 4: *"A **proposition** is a statement with a well-defined truth value, either true (T) or false (F)."*
[62] [9] 33ff.: *"[...] a **result** explicitly presents an important mathematical fact without actually proving it [...]"*
[63] [32] p. 6: *"A **law** of propositional logic is an equivalence of propositional terms."*
[64] [23] p. 2: *"Once it is known that a statement has a proof, it is known as a theorem, lemma, corollary, proposition, or **law**."*
[65] [71] p. 14: *"Given a set of premises Γ and a **conjecture** ϕ, an ATP tries to find either a proof that the Γ logically implies ϕ, or [...]"*

5.3 The 8-Building-Blocks Hypothesis of PBM

5.3.1 Objective

In all four sources we used in 5.2, we could identify in total 8 (modulo synonymous terminology) building blocks of PBM: definitions, theorems, propositions, lemmas, corollaries, axioms, proofs, and conjectures. This leads us to what we will call below the *"8-Building-Blocks Hypothesis of PBM"*. We have to scientifically verify this Hypothesis because of the partially different descriptions we found in these four sources. In the following, we will define a sample of real PBM publications and *count* the building blocks we can identify in the sample. To our best knowledge, this type of data-driven investigation of PBM building blocks is novel. It will enable us both to verify the above Hypothesis and to gain some new insights.

5.3.2 Analysis

In order to make our sample to some extend representative, we choose mathematical textbooks from different mathematical disciplines, written by different authors, being at least 200 pages long, and in which building blocks are numbered or at least easily identifiable due to the typesetting of the textbook. Moreover, we will include two mathematical online resources, which are devoted to PBM results from different mathematical disciplines, and in which it is easy to count building blocks of PBM. Our sample contains the following publications:

1. [43] Translation of Thomas L. Heath: *"Euclid's Elements" Green Lion Press*, 2013
2. [36] Charles C. Pinter's *"A Book of Set Theory"*, *Dover Publications Inc.*, 2017
3. [35] G. H. Hardy, E. M. Wright, *"An Introduction to the Theory of Numbers"*, *Oxford at the Clarendon Press*, 1975
4. [18] Harro Heuser, *"Lehrbuch der Analysis, Teil 1"*, *B. G. Teubner*, 1994
5. [27] Serge Lang, *"Algebra"*, *Springer*, 2002
6. [25] Richard Kohar, *"Basic Discrete Mathematics, Logic, Set Theory, & Probability"*, *World Scientific*, 2016
7. [26] Sheldon Axler, *Linear Algebra Done Right*, *Springer*, 2015
8. [64] Online resource `https://www.bookofproofs.org/` *"BoP is an open book dedicated to mathematics, physics, and computer science. Its goal is to broaden the public knowledge of the axiomatic method."*
9. [65] Online resource `https://proofwiki.org/` *"ProofWiki is an online compendium of mathematical proofs! Our goal is the collection, collaboration and classification of mathematical proofs."*

The 8-Building Block Hypothesis	[43]	[36]	[35]	[18]	[27]	[25]	[26]	[64]	[65]	Total
Pages	481	214	413	566	606	478	331	5,047	3,048	11,184
Definitions	131	106	65	83	558	107	177	629	861	2,717
Theorems		142	459	279	191	66	204	90	918	2,349
Propositions	465				151			597		1,213
Lemmas	15	40		34	51			111		251
Corollaries	22	25			92	3		126		268
Axioms	5	14		28	31	4	9	30		121
Proofs	480	174	434	267	435	57	181	965	918	3,911
Conjectures					5			4		9
Potential other building blocks	[43]	[36]	[35]	[18]	[27]	[25]	[26]	[64]	[65]	Total
Examples		35	81	217	154	220	154	71	774	1,706
Remarks	5	16	192	27	58	14	93	57		462
Exercises		415		588	347	810	554			2,714
Applications			11	166	8	3		28		216
Motivations				14				16		30
Introductions		57	260	108	100	99	103	649		1,376
Historical Notes		6	2	12		1		2,639	495	3,155
Interactive Widgets								94		94
Images	635	26	21	116	92	170	16	206		1,282
Tables				3		97				100

Table 3: Sample of PBM Building Blocks

We present the results of our counting exercise in the **Table** 3. The following methodology has to be applied to reproduce the results:

- The numbers of pages of textbooks were determined, omitting prefaces, forewords, appendixes, acknowledgments, indices, and bibliographies, i.e., count only those pages that are devoted to PBM and not diluted by other artifacts.
- The numbers of *"pages"* of the online resources [64] and [65] correspond to the number of identifiable online pages as of 09/12/20 and are for technical reasons to some extend approximations.
- Since we wanted our sample to cover *different* mathematical disciplines, we excluded one of four parts of [27] devoted to linear algebra since linear algebra is covered in [26].
- For the same reason, the counts of [64] and [65] have been corrected as not to double-count parts devoted to *"Euclid's Elements"*, which are covered in [43].
- The counts of definitions include not only designated structure blocks titled *"definitions"* but also definitions stated inline in the running text like, for instance, in [27], as far as this was recognizable in its text.
- If more than one proof for a theorem existed, count only one proof.
- Count theorems, propositions, lemmas, and corollaries without proofs separately. In such cases, authors decided to leave the proof for the reader *"as an exercise"* or considered the proof *"too trivial"*.
- The counts for *"remarks"* might actually include other building blocks denoted in the sources not as *"remarks"* but, for instance, as *"summaries"*, *"notes"*, *"warnings"*, or as *"comments"*.

- Count *"exercises"* separately: They deserve a special note. In the sample, exercises may cover almost all other categories of building blocks. For instance, there are exercises with examples, exercises with new definitions and theorems not mentioned in the main text, exercises with proofs *"left to the reader"*, or exercises with applications. Exercises may also contain tabular data, images, and remarks.

5.3.3 Conclusions

The **Table** 3 allow us to confirm the 8-Building-Block Hypothesis: PBM, as we defined it in 1, is structured by no other but the eight building blocks, namely definitions, theorems, propositions, lemmas, corollaries, axioms, proofs, and conjectures. These are precisely those building blocks that we encounter in the *deduction* of logical results, PBM is the language to express.

Requirement 4 (8 Building Blocks of FPL)

> *The FPL language* MUST *allow eight building blocks on a syntactical level: definitions, theorems, propositions, lemmas, corollaries, axioms, proofs, and conjectures.*

Still, we could observe different coverage of each building block in the sample, ranging from definitions and proofs occurring in all eight publications to conjectures occurring only in two. In particular, we could find axioms in 7 out of 9 publications of our sample. This leads us to the **Requirement** 4 that requires all eight building blocks, including axioms and conjectures in FPL. Moreover, **Table** 3 provides us with some additional new insights:

1. There are many other building blocks we encounter in mathematical publications, which do not belong to PBM like we have defined it, i.e., are not required for *deduction*. We could particularly identify examples, remarks, exercises, applications, motivations, introductions, historical notes, images, tables, and even - in online sources, interactive widgets. Consistently with our definition of PBM, these additional building blocks are also not mentioned in the sources [71], [9], [32], and [23]. If we put the numbers of all 10,839 pure PBM building blocks and all 11,135 non-PBM building blocks in relation to their total 21,974, we get the shares of 50.7% PBM vs. 49.3% non-PBM. The shares shift slightly if we exclude 3,155 of pure historical notes, yielding in the numbers 18,819 total, thereof 10,839 (57.6%) PBM and 7,980 (36.3%) non-PBM contents. This leads us to the following observation and requirement

   Observation 5 (Density of PBM in Publications)

   > *Nearly half of the volume of a "typical" mathematical textbook contains non-PBM contents that explain and complement the remaining PBM contents.*

Requirement 5 (Surrounding FPL by Non-deductive Building Blocks)

> *FPL* MUST *allow being embedded into surrounding text that might contain non-deductive contents, for instance introductions, motivations, examples, exercises, remarks, observations, applications, and others (e.g. media, visualizations, algorithms, and other data-driven possibilities).*

The **Requirement** 3 together with the **Requirement** 5 might negatively affect the readability of FPL, in contrast to our **Principle** 1. To address this issue, we should add some support to translate FPL code back into natural languages that would help unexperienced users to read the FPL code. We formulate a new requirement for this purpose:

Requirement 6 (Support for Localization)

> *FPL parsers* SHOULD *support localization to translate mathematical notions formulated in FPL code into other languages, in particular natural languages.*

2. There are 4,081 of theorem-like building blocks (theorems, propositions, lemmas, and corollaries). The sources [71], [9], [32], and [23] consistently stated that such theorem-like building blocks exist. This leads us to the following requirement:

   Requirement 7 (Four Theorem-like Building Blocks in PBM)

   > *Mathematical publications contain four theorem-like building blocks: theorems, propositions, lemmas, and corollaries. FPL* SHOULD *allow to distinguish between these four types.*

3. Finally, let us have a closer look at the ratio of theorem-like building blocks that are unproven and their semantical difference to conjectures. Our sample contains a total of 3,911 proofs. 95.8% of the total of 4,081 theorem-like statements in our sample were proved, and 4.2% (one of 24) remained unproven. Still, unproven theorem-like building blocks are semantically *different* to conjectures since theorem-like building blocks are known to be true, and they may only remain unproven because the author consciously omits a proof that is *known*. A conjecture is not known to be *true* or *false*. A proof (or disproof) of a conjecture is omitted for another reason: it is simply *unknown*. This leads us to the following requirement:

   Requirement 8 (Unproved Theorem-like Blocks vs. Conjectures)

   > *FPL* MUST *allow zero to many proofs for each theorem-like building block and* SHOULD *provide syntactic means to omit a proof for such a theorem-like building block explicitly. FPL* MUST *allow a clear syntactic distinction between a theorem-like building block without proof and a conjecture.*

5.4 Expressing PBM in Propositional and Predicate Logic

5.4.1 Objective

After we have identified the 8 building blocks of PBM in **Requirement** 4, it is now time to investigate PBM from the perspective of *logic*. Examples of logics are *propositional logic* (denoted by **PL0**) and *predicate logics* (denoted by **PLm** for $m \geq 1$). We are going to dedicate this subsection to them. We will address the following questions:

1. What are propositions in PL0 and how are PL0's syntax and semantics defined?
2. What are predicates in PL1 and how are PL1's syntax and semantics defined?
3. What are predicates, syntax, and semantics in PL2?
4. What are higher-order logics PLm?

5.4.2 Syntax and Semantics of PL0

As we mentioned already in 5.2, the term *"proposition"* may have two totally different meanings depending on the context: First, a proposition refers to one of the four theorem-like building blocks of PBM we have identified (next to the terms *"theorem"*, *"lemma"*, and *"corollary"*). In the context of PL0, a **proposition** is a logical formula p whose syntax we can define recursively[66]:

Definition 1 (Syntax of Propositions (PL0))

*The syntax a PL0 consists of a tuple (B, V), called its **syntax signature** with the set $B := \{\top, \bot\}$ of truth symbols and a set $V := \{v_1, v_2, \ldots\}$ of variable symbols such that:*

- *$\top$ and $\bot$ are propositions.*
- *Every variable in V is a proposition.*
- *If p and q are propositions, then also*

$$(\neg p), (p \wedge q), (p \vee q), (p \Rightarrow q), (p \Leftrightarrow q)$$

are propositions.

We see that the most simple *syntax* of propositions may consist of two distinguished symbols we call the **truth symbols**: $\top$ (*true*) and $\bot$ (*false*), or of variables from a stocks of symbols V. We will call these simple propositions **prime propositions**[67]. On the other hand, the syntax may involve additional symbols like $\neg$ **negation**, $\wedge$ **conjunction**,

[66] [22] p. 87
[67] [25] p. 6

$\vee$ **disjunction**, $\Rightarrow$ **implication**, and $\Leftrightarrow$ **equivalence**. We will call propositions involving these additional symbols $\neg, \wedge, \vee, \Rightarrow$, and $\Leftrightarrow$ **compound propositions**[68]. Finally, when used with these symbols, propositions have to be put into parantheses in order to disambiguate formulas like $\neg p \vee q$ by having to write either $(\neg(p \vee q))$ or $((\neg p) \vee q)$[69].

When we talk about *"disambiguation"* via parantheses, we only address disambiguation of propositions on a pure syntactical level. Parentheses only fix the exact order a given proposition has to be read (by a human) or parsed (by a computer) but do not explain the meaning, i.e. the *semantics* of a proposition. In order to precisely define the semantics of propositions, we need to introduce the term *interpretation*[70]:

Definition 2 (Interpretation of PL0)

> *Given a proposition p with the variables $v_1, v_2, \ldots, v_n$, its **interpretation** $I(p)$ is any* function *of the form $I : V^n \to B$.*

In other words, given a proposition p, an interpretation $I(p)$ assigns one of the values *true* ($\top$) or *false* ($\bot$) to each of its n variables and returns exactly one truth value from B. We also agree to write $I \models p$ in case I returns $\top$ (i.e. we interpret p as being *"true"*) under this variable assignment, or else we agree to write $I \not\models p$ in case I returns $\bot$ (i.e. we interpret p as being *"false"*) under this variable assignment. These technical prerequisites help us to define the semantics of propositions in the following way:

Definition 3 (Semantics of PL0)

> *Given an interpretation I, the **semantics** of propositions is defined recursively by:*
>
> - *For prime propositions:*
> - *Syntax:* $\top$
> *Semantics:* $I \models \top$.
> - *Syntax:* $\bot$
> *Semantics:* $I \not\models \bot$.
> - *Syntax: some variable v*
> *Semantics:* $I \models v$ *if and only if* $I(v) = \top$.
> - *For compound propositions:*
> - *Syntax:* $(\neg p)$
> *Semantics:* $I \models (\neg p)$ *if and only if* $I \not\models p$.
> - *Syntax:* $(p \wedge q)$
> *Semantics:* $I \models (p \wedge q)$ *if and only if* $I \models p$ *and* $I \models q$.

[68] [25] p. 6
[69] By defining further notational rules and conventions, we could get rid of parantheses in some precise cases. We do not introduce these rules and conventions since they are irrelevant to understand the following text.
[70] [22] p. 88

- *Syntax:* $(p \vee q)$
  Semantics: $I \models (p \vee q)$ *if and only if* $I \models p$ *or* $I \models q$.
- *Syntax:* $(p \Rightarrow q)$
  Semantics: $I \models (p \Rightarrow q)$ *if and only if* $I \not\models p$ *or* $I \models q$.
- *Syntax:* $(p \Leftrightarrow q)$
  Semantics: $I \models (p \Leftrightarrow q)$ *if and only if* $I \models (p \Rightarrow q)$ *and* $I \models (q \Rightarrow p)$.

One important thing to understand is that the semantics of a proposition is strictly separated from its syntax. For instance, a proposition like $(\neg(p \vee q))$ may have two different interpretations I_1 and I_2 such that $I_1 \models (\neg(p \wedge q))$ but $I_2 \not\models (\neg(p \wedge q))$, although its syntax does not change under these two interpretations:

- First interpretation $I_1(p) = T$, $I_1(q) = F$

$$\underbrace{(\neg(\underbrace{\underbrace{p}_{\models p} \wedge \underbrace{q}_{\not\models q}}_{\not\models (p \wedge q)}))}_{\models (\neg(p \wedge q))}$$

- Second interpretation $I_2(p) = T$, $I_2(q) = T$

$$\underbrace{(\neg(\underbrace{\underbrace{p}_{\models p} \wedge \underbrace{q}_{\models q}}_{\models (p \wedge q)}))}_{\not\models (\neg(p \wedge q))}$$

Concrete interpretations are means of how we can *disambiguate* PBM on a *semantical level*, as opposed to parentheses used in compound propositions that helped us to disambiguate expressions on their syntactical level. It leads us to some new important requirements for FPL:

Requirement 9 (Parantheses in FPL)

The syntax of FPL MUST *avoid ambiguous formulas (disambiguation on syntactical level) by appropriate parantheses rules.*

Requirement 10 (Deterministic Interpreters of FPL)

Only a ***deterministic interpreter*** SHALL *be allowed for FPL to ensure that its formulas always have the same unambiguous interpretation (disambiguation on semantical level).*

The existence of complex formulas that can be either true or false without changing their syntax brings us to the important terms *model* or *model relation*:

Definition 4 (Model Relation on PL0)

*Given a proposition p and an interpretation I, we say that I **models** (or **fulfills**, or **is a model of**) p if and only if $I \models p$. The sign $\models$ is a relation between all propositions (on a syntactical level) and all of their possible interpretations.*

We may use the simplified notation $\models p$ if p is true and $\not\models p$ if p is false under a specific interpretation I, given that I is clear from the context. The purpose of PL0 is to be able to decide whether $\models p$ holds, given a proposition p and an interpretation I.

Sometimes, it is useful to think of a proposition p with n variables $v_1, \ldots, v_n$, given an interpretation I, as a **Boolean function** $f_I : B^n \to B$ taking the value $\top$ exactly in the case if I induces a model of p[71]:

$$f_I(v_1, \ldots, v_n) := \begin{cases} \top & \text{if and only if } I \models p, \\ \bot & \text{if and only if } I \not\models p. \end{cases}$$

A Boolean function can be visualized using a so-called **truth table**. The **Table** 4 shows the truth table of some most important propositions

p	q	$(\neg p)$	$(\neg q)$	$(p \wedge q)$	$(p \vee q)$	$(p \Rightarrow q)$	$(p \Leftrightarrow q)$
$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$

Table 4: The truth tables of $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$

A Boolean function is named after George Boole (1815 - 1864). He was one of English mathematicians[72] who 1847 created a pioneer system of manipulating propositions on their pure syntactical level, we call today a **Boolean algebra**. Symbolic manipulation rules of a Boolean algebra can be found in appendix 9.1. A Boolean algebra makes do without the symbols $\Rightarrow$ and $\Leftrightarrow$. In fact, based on **Table** 4, we can easily verify that the symbols $\Rightarrow$ and $\Leftrightarrow$ are syntactical abbreviations for more complex expressions based on the symbols $''\neg, \wedge, \vee''$ alone[73]:

$$\begin{aligned} (p \Rightarrow q) &:= ((\neg p) \vee q), \\ (p \Leftrightarrow q) &:= (((\neg p) \vee q) \wedge ((\neg q) \vee p)). \end{aligned}$$

For centuries until the 20th century, mathematicians deeply believed that the truth and provability of mathematical statements are the same concepts. Therefore, Boole's PL0[74]

[71] [22] p. 89

[72] Boole's famous friend, Augustus De Morgan (1806 - 1871), published a similar work in the same year and month as Boole!

[73] In fact, it is possible to reduce the set of symbols $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ of PL0 like we have introduced it in **Definition** 1 even further to the sets $\{\neg, \wedge\}$, $\{\neg, \vee, \}$, or $\{\neg, \Rightarrow\}$ ([22] p. 92), making the syntax of PL0 less complicated but also less readable. For the sake of **Principle** 1, we are not further interested in such simplifications.

[74] compare 5.4

seemed to be quite a discovery of a *"Characteristica universalis"* in Leibniz's sense! Boole claimed that his system *"does not depend on the interpretation of the symbols which are employed, but solely upon the laws of their combination"*[75]. PL0 seemed to enable mathematicians to derive mathematical truth by simply manipulating symbols. In particular, some propositions seemed to be true or false in an absolute way, i.e. under *any* interpretation. Trivially, this applies for the symbols $\top$ and $\bot$ (because by definition $\models \top$ and $\not\models \bot$). More interestingly, it also applies for many compond propositions like $\models (p \vee (\neg p))$ and $\not\models (p \wedge (\neg p))$. A proposition that is true under any interpretation, is called a **tautology** or a **valid statement** while a proposition that is false under any interpretation, is called a **contradiction**[76] or **unsatisfiable**[77]. Combining what we have already learned about models, a proposition is **satisfiable** if and only if it has a model. Of course, there are satisfiable PL0 formulas that are neither tautologies nor contradictions.

5.4.3 Syntax and Semantics of PL1

The syntax of PL0 is too poor to express PBM. For instance, it is not even possible to express simple symbols like numbers $0, 1, -\frac{1}{2}$, etc., or to state the *existence* of mathematical objects, or to express *how many* such objects are involved. *Predicates* aim to close this gap since they allow a significantly richer syntax than propositions do, while still being expressed *formally*, i.e. without using natural languages. We will call such a richer symbolic language a **first-order predicate logic** (denoted by PL1). The syntax of PL1 can be defined recursively. We simplify and slightly modify the introduction to predicate logic given in [22]:[78]

Definition 5 (Syntax of First-Order Predicate Logic (PL1))

*The syntax of PL1 consists of a tuple (V, F, P, Q), called its **syntax signature**, that involves the following symbols and grammar rules:*

- *V: A set of **variable symbols**, for instance $V = \{x, y, z, \ldots\}$.*
- *F: A set of **functional terms**, for instance $F = \{0, 1, 2, +, -, \sin, \exp, \ldots\}$. A functional term has an **arity** $n \geq 0$, i.e. involves $n \geq 0$ symbols, called its **parameters**. The parameters may themselves be variables or other functional terms. Functional terms with zero parameters are called **constants**.*
- *P: A set of **predicate symbols**, for instance $P = \{\top, \bot, >, <, =, \ldots\}$[79]. Like functional terms, predicates have an **arity** $n \geq 0$ i.e. involve zero, one, or more functional terms, called their **parameters**.*

[75] [25] pp. 2,3
[76] [25] p. 22
[77] [22] p. 90
[78] 102ff.
[79] $\{\top, \bot\}$ are the same truth symbols we encountered already in PL0.

- *Q: A set of **quantor symbols**, for instance $Q = \{\forall, \exists, \exists_n, \ldots\}$. A quantor is supposed to **bind** some variables inside a predicate. If all variables are bound in a predicate by a quantor, we call it a **closed predicate**. If it has **unbound** variables, we call it an **open predicate**.*

- *We will call predicates **prime predicates**, if their parameters consist only of functional terms or other predicates and involve neither quantor symbols, nor the expressions $\neg, \wedge, \vee, \Rightarrow$, and $\Leftrightarrow$. If a predicate involves quantor symbols or one or more predicates combined via these expressions, we will call it a **compound predicate** and put it into a paratheses, like we did for propositions. In particular, if ϕ, ψ are closed predicates with the variables $\xi \in V^m$, $m \geq 1$ and setting $n \geq 0$, the following are (compound) predicates:*

$$\forall\xi(\phi), \exists\xi(\phi), \exists_n\xi(\phi), (\neg\phi), (\phi \wedge \psi), (\phi \vee \psi), (\phi \Rightarrow \psi), (\phi \Leftrightarrow \psi).$$

As an example, we want to express a possible mathematical statement using the syntax of PL1.

Example 1 (A Predicate in PL1)

- *We are going to express the mathematical statement*

  "For all x with $x \neq 0$ there is an y such that $x \cdot y = 1$".

  Note that it uses natural language and usual mathematical notation.

- *We choose the following syntax signature:*
  - $V = \{x, y\}$,
  - $F = \{0, 1$ *(constants)*$, Multiply$ *(binary functional symbol)*$\}$,
  - $P = \{Equals$ *(binary predicate)*$\}$, *and*
  - $Q = \{\forall, \exists\}$.

- *Now, we can express the statement as a (compound) predicate:*

$$\forall x((\neg Equals(x, 0)) \Rightarrow \exists y(Equals(Multiply(x, y), 1)))$$

To define the *semantics* of PL1, we will need two additional concepts. The first one is the concept of the *domain of discourse*:

Definition 6 (Domain of Discourse)

*An arbitrary, non-empty set $\mathbb{D}$ is called the **domain of discourse**.*

A *domain of discourse* $\mathbb{D}$ is simply the set of all mathematical objects the variables and constants of a PL1 may range. For instance, $\mathbb{D}$ may be the set of all real numbers or all integers. The second concept we are going to introduce is similar to PL0: It is the concept of *interpretation*. However, an interpretation of a predicate in PL1 is defined a little bit more complicated than it was the case in PL0:

Definition 7 (Interpretation of First-Order Predicate Logic (PL1))

*Let $\mathbb{D}$ be a domain of discourse and let $\Sigma := (V, F, P, Q)$ be the syntax signature of PL1. $I_{\mathbb{D}}$ is called the **interpretation** of Σ under $\mathbb{D}$, if and only if:*

- *For every n-ary ($n \geq 1$) functional term $f \in F$, $I_{\mathbb{D}}(\phi)$ replaces the n parameters of f by a tuple of arguments being elements of $\mathbb{D}^n$ and maps them to a single element of $\mathbb{D}$. In other words, $I_{\mathbb{D}}(f)$ represents a **map** in mathematical sense $I_{\mathbb{D}}(f) : \mathbb{D}^n \to \mathbb{D}$.*
- *In case of a functional term $f \in F$ with no parameters, ($n = 0$), $I_{\mathbb{D}}$ replaces f by a single element of $\mathbb{D}$. In other words, $I_{\mathbb{D}}(f)$ represents a **constant** in mathematical sense.*
- *For every m-ary ($m \geq 1$) predicate $\phi \in P$, $I_{\mathbb{D}}$ replaces ϕ by is a fixed (!) subset $I_{\mathbb{D}}(\phi) \subseteq \mathbb{D}^n$. In other words, $I_{\mathbb{D}}(\phi)$ is a **relation** in mathematical sense[80]. This relation enables us to tranform the predicate into a PL0 proposition as follows: The predicate ϕ becomes a true proposition ($\top$), if the m arguments of ϕ equal some tuple of values in $\mathbb{D}^m$ that* belongs *(!) to the subset $I_{\mathbb{D}}(\phi)$. Otherwise, the predicate ϕ becomes a false proposition ($\bot$).*
- *In case of a predicate $\phi \in P$ with no arguments ($n = 0$), the interpretation $I_{\mathbb{D}}(\phi)$ corresponds to a subset of $\mathbb{D}^0$. Since $\mathbb{D}^0$ is a **singleton** $\{d\}$ for some $d \in \mathbb{D}$, $I_{\mathbb{D}}(\phi)$ is identical[81] to the interpretation of variables in PL0: ϕ becomes like a variable that is either truth $\top$ if $\phi \in \{d\}$ (equivalently $\phi = d$), or false $\bot$ if $\phi \notin \{d\}$ (equivalently $\phi \neq d$). Whether or not ϕ is true or false, depends on the concret intepretation $I_{\mathbb{D}}$. Note that the predicate ϕ becomes a constant in $\mathbb{D}$, like it was the case for functional terms with no arguments. In contrast to constants in functional terms, ϕ is interpreted as being true (if it equals the contant) or being false (if it does not equals the constant).*

Thus, PL1 and PL0 are related to each other in the following way: While the interpretation $I_{\mathbb{D}}$ of functional symbols of PL1 replaces them by some constants in $\mathbb{D}$ or functions $\mathbb{D}^n \to \mathbb{D}$, the interpretation of predicates of PL1 turns them into propositions of PL0. If $I_{\mathbb{D}}$ is a **model** of ϕ (notated via $I_{\mathbb{D}} \models \phi$), ϕ turns into a proposition with the truth value $\top$. If $I_{\mathbb{D}}$ is not a model of ϕ (notated $I_{\mathbb{D}} \not\models \phi$), ϕ becomes a proposition with the truth value $\bot$. Now, we can precisely define what the semantics of PL1 predicates is:

Definition 8 (Semantics of PL1 Predicates)

*Given a domain of discourse $\mathbb{D}$ and an interpretation $I_{\mathbb{D}}$, the **semantics** of PL1 predicates is defined recursively by:*

[80] Note that we use the terms *"map"*, *"constant"*, and *"relation"* referring to their modern mathematical definitions that can be found, for instance in [37]. In order to define the interpretation of PL1, we use mathematics as a metalanguage!

[81] [19] p. 120

- *Variables, constants and other functional terms:*
  - *Syntax:* x *(*x *being a variable)*
    Semantics: $I_{\mathbb{D}}(x) \in \mathbb{D}$.
  - *Syntax:* f *(*f *being a constant)*
    Semantics: $I_{\mathbb{D}}(f) \in \mathbb{D}$.
  - *Syntax:* $f(g_1, \ldots, g_n)$ *(*f *being an* n*-ary functional term with* $n \geq 1$ *arguments being variables, constants, or other functional terms)*
    Semantics: $I_{\mathbb{D}}(\sigma)$ *is some mathematical function*

$$I_{\mathbb{D}} := \begin{cases} \mathbb{D}^n & \to \mathbb{D} \\ (I_{\mathbb{D}}(g_1) \ldots, I_{\mathbb{D}}(g_n)) & \to I_{\mathbb{D}}(f) \end{cases}.$$

- *For prime predicates:*
  - *Syntax:* $\top$
    Semantics: $I_{\mathbb{D}} \models \top$.
  - *Syntax:* $\bot$
    Semantics: $I_{\mathbb{D}} \not\models \bot$.
  - *Syntax:* ϕ *(no arguments)*
    Semantics: $I_{\mathbb{D}} \models \phi$ *if and only if* $\phi = d$ *for some fixed (!)* $d \in \mathbb{D}$ *depending on* $I_{\mathbb{D}}$.
  - *Syntax:* $\phi(f_1, \ldots, f_n)$, $f_1, \ldots, f_n$ *being functional terms*
    Semantics: $I_{\mathbb{D}} \models \phi(f_1, \ldots, f_n)$ *if and only if* $(I(f_1), \ldots, I(f_n)) \in I(\phi)$.
- *For compound predicates, where* ϕ, ψ *are closed predicates with the variables* $\xi \in V^m$, $m \geq 1$ *and for* $n \geq 0$*:*
  - *Syntax:* $\forall \xi(\phi)$
    Semantics: $I_{\mathbb{D}} \models \phi$ *for all tuples* $d \in \mathbb{D}^m$ *with* $I(\xi) = d$.
  - *Syntax:* $\exists \xi(\phi)$
    Semantics: There exists some tuple $d \in \mathbb{D}^m$ *such that* $I(\xi) = d$ *and* $I_{\mathbb{D}} \models \phi$.
  - *Syntax:* $\exists_n \xi(\phi)$
    Semantics: There exist exactly n *tuples* $d \in \mathbb{D}^m$ *such that* $I(\xi) = d$ *and* $I_{\mathbb{D}} \models \phi$. *If* $n = 0$*, none exists.*
  - *Syntax:* $(\neg\phi)$
    Semantics: $I_{\mathbb{D}} \models (\neg\phi)$ *if and only if* $I_{\mathbb{D}} \not\models \phi$.
  - *Syntax:* $(\phi \wedge \psi)$
    Semantics: $I_{\mathbb{D}} \models (\phi \wedge \psi)$ *if and only if* $I_{\mathbb{D}} \models \phi$ *and* $I_{\mathbb{D}} \models \psi$.
  - *Syntax:* $(\phi \vee \psi)$
    Semantics: $I_{\mathbb{D}} \models (\phi \wedge \psi)$ *if and only if* $I_{\mathbb{D}} \models \phi$ *or* $I_{\mathbb{D}} \models \psi$.

- *Syntax:* $(\phi \Rightarrow \psi)$
  Semantics: $I_{\mathbb{D}} \models (\phi \Rightarrow \psi)$ *if and only if* $I_{\mathbb{D}} \not\models \phi$ *or* $I_{\mathbb{D}} \models \psi$.
- *Syntax:* $(\phi \Leftrightarrow \psi)$
  Semantics: $I_{\mathbb{D}} \models (\phi \Leftrightarrow \psi)$ *if and only if* $I_{\mathbb{D}} \models (\phi \Rightarrow \psi)$ *and* $I_{\mathbb{D}} \models (\psi \Rightarrow \phi)$.

Like it was the case in PL0, a predicate ϕ in PL1 having the same syntactical representation may have *different* interpretations with $I_{\mathbb{D}} \models \phi$ and $I'_{\mathbb{D}'} \not\models \phi$. Those interpretations depend on the domains of discourses $\mathbb{D}$ and $\mathbb{D}'$. We are going to demonstrate this phenomenon using the above **Example** 1:

- If $\mathbb{D}$ ranges over all real numbers $\mathbb{R}$, $I_{\mathbb{R}}(0)$ and $I_{\mathbb{R}}(1)$ are interpretations of the real numbers 0 and 1. $I_{\mathbb{R}}(Multiply)$ is an interpretation of the usual real function of multiplying two real numbers. $I_{\mathbb{R}}(Equals)$ is an interpretation of the equality relation of two real numbers. With $I_{\mathbb{R}}$, we get a first interpretation of the predicate:

  $$I_{\mathbb{R}} \models \forall x((\neg Equals(x, 0)) \Rightarrow \exists y(Equals(Multiply(x, y), 1))).$$

  $I_{\mathbb{R}}$ is a model for the predicate since every real number that is unequal to 0 has a multiplicative inverse that is also a real number, for instance $2 \cdot \frac{1}{2} = 1$.

- If $\mathbb{D}$ ranges over all integers $\mathbb{Z}$, $I_{\mathbb{Z}}(0)$ and $I_{\mathbb{Z}}(1)$ are interpretations of the integers 0 and 1. $I_{\mathbb{Z}}(Multiply)$ is an interpretation of the multiplication of integers as well as $I_{\mathbb{Z}}(Equals)$ is an interpretation of the equality relation of two integers. $I_{\mathbb{Z}}$ provides us with another interpretation of the predicate:

  $$I_{\mathbb{Z}} \not\models \forall x((\neg Equals(x, 0)) \Rightarrow \exists y(Equals(Multiply(x, y), 1))).$$

  $I_{\mathbb{Z}}$ is not a model for the predicate since not all (but only two) integers x that are uniqual the integer 0 have multiplicative inverses: $1 \cdot 1 = 1$ and $-1 \cdot -1 = 1$.

Note the distinction between the meaningless symbols 0 and 1, $Multiply$ and $Equals$, and their concrete interpretations $I_{\mathbb{R}}$ and $I_{\mathbb{Z}}$! It demonstrates how the syntax and the semantics of PL1 are separated from each other. There is no unique interpretation of symbols like 0 and 1 unless we fix it with some *domain of discourse* $\mathbb{D}$ and the interpretation $I_{\mathbb{D}}$. Both exist outside and independently from the syntax of PL1! The domains of discourses and interpretations of PL1 are always defined externally from its syntax.

5.4.4 Syntax and Semantics of PL2

Modern PBM texts make extensive use of symbols like "=". One important fact about PL1 is that despite its richness of syntax compared to PL0, it is impossible to express "=" or similar concepts on a purely syntactic level. Thus, PL1's syntax is not sufficiently rich. We cannot express all of PBM using it. If we try to formalize different statements in PL1, in which we exclude the predicate $"="$, some statements will become very cumbersome or even impossible to state. In [22][82], the following example is given to illustrate the

[82] 111ff.

problem: Imagine we want to formalize the terms of a binary relation R being *left-total* and *right-unique*[83] in PL1 without having the symbol "=" at hand: We can easily formalize left-totality by requiring that R fulfills the following expression:

$$\forall x(\exists y(R(x,y))).$$

However, we will experience great difficulties in expressing the right uniqueness of R. We have to state formally: There is no such x such that there are two *different* elements y and z with $R(x,y)$ and $R(x,z)$. In other words, if both, $R(x,y)$ and $R(x,z)$, are fulfilled, then y and z must be *equal*. In order to state this formally, we lack a symbol like "=" to express our *intuitive* notion of the *equality* or a symbol like "$\neq$" to express our *intuitive* notion of the *difference* of two mathematical objects. One way to close the gap is to enrich the syntax of PL1 by introducing a new symbol, say "=," creating a so-called **PL1 with equality**[84]. Then, we can easily formalize right-uniqueness by requiring R to fulfill the following expression

$$\forall x(\forall y(\forall z(R(x,y) \wedge R(x,z) \Rightarrow y = z))).$$

Introducing equality either in the notation "=" (or in the notation of *"Equals*$(\cdot,\cdot)$*"* like we did in **Example** 1) is not a matter of abbreviating things that would otherwise be cumbersome to state in PL1 alone. It is possible to show that it is fundamentally *impossible* to express equality in PL1[85]. Things change fundamentally, however, if we allow the quantors $\forall, \exists$ to bound not only variables, like we did in PL1, but also functional terms and predicates. Loosely speaking, we enrich the syntax of PL1 to be able to express the following:

- $\forall\phi(\ldots)$ should mean *"For all predicates it holds ..."*.
- $\exists\phi(\ldots)$ should mean *"There exists a predicate ϕ such that ..."*.
- $\forall f(\ldots)$ should mean *"For all functional terms it holds ..."*.
- $\exists f(\ldots)$ should mean *"There exists a functional term f such that ..."*.

Given the signature $\Sigma := (V, F, P, Q)$, in the syntax of the so-called **second-order predicate logic** (denoted by PL2), we are going to allow the stocks of available variables V to represent also individual instances of functional terms $f \in F$ as well as predicates $\phi \in P$. We will also allow bounding these variables by the quantors from Q like $\forall$ and $\exists$. Thus, the syntax of PL2 covers the syntax of PL1, extending it to formulas

[83] compare [37] p. 69. Left-totality and right-uniqueness are basic properties of binary relations characterizing all mathematical functions.

[84] [22] 146ff.

[85] A proof can be found in [22] 114ff. The proof demonstrates that for every interpretation $I_{\mathbb{D}}$ of a formula in PL1 containing the symbol "=" and interpreting it as equality, another interpretation $I'_{\mathbb{D}'}$ can be constructed in which the symbol loses its first interpretation. In other words, no absolute interpretation of "=" can exist in PL1!

in which variables can denote functional terms and predicates.

Similarly as we did for PL1 for all formulas, we set $\mathbb{D}$ to be a domain of discourse, and call $I_{\mathbb{D}}$ the **interpretation** of Σ under $\mathbb{D}$. Now, we extend the semantics from PL1 to PL2 as follows:

Definition 9 (Semantics of PL2 (Extension of PL1))

- *If ξ is a variable denoting a functional term, $I_{\mathbb{D}}(\xi)$ assigns ξ a single function $\mathbb{D}^n \to \mathbb{D}$. Correspondingly and for a formula ϕ in which ξ is bound:*
  - *$\forall\xi(\phi)$ means: $I_{\mathbb{D}} \models \phi$ for all $f : \mathbb{D}^n \to \mathbb{D}$*
  - *$\exists\xi(\phi)$ means: $I_{\mathbb{D}} \models \phi$ for some $f : \mathbb{D}^n \to \mathbb{D}$*
- *If ξ is a variable denoting a predicate, $I_{\mathbb{D}}(\xi)$ assigns ξ a subset of $\mathbb{D}^n$ (a mathematical relation on $\mathbb{D}$). Correspondingly and for a formula ϕ in which ξ is bound:*
  - *$\forall\xi(\phi)$ means: $I_{\mathbb{D}} \models \phi$ for all relations $R \subseteq \mathbb{D}^n$.*
  - *$\exists\xi(\phi)$ means: $I_{\mathbb{D}} \models \phi$ for some relation $R \subseteq \mathbb{D}^n$.*

Not surprisingly, and by analogy to what we have learned already about PL0 and PL1, formulas in PL2 may have *different* interpretations without changing their syntax. Nevertheless, within PL2, it is now easy to state what a symbol like "=" or like *"$Equals(\cdot,\cdot)$"* means on a syntactical level, in an absolute way that is independent from any interpretation. This can be performed in the following way[86]:

$$\forall x(\forall y((Equals(x,y) \Leftrightarrow \forall\xi(\xi(x) \Leftrightarrow \xi(y))))).$$

In this formula, the expression $\forall\xi$ iterates through all predicates (i.e. variables representing $\xi \in P$) in the concrete underlying signature (V, F, P, Q).

5.4.5 Examples of Expressing Foundations of PBM in PL2

PL2 is much more than just *"PL1 with equality"*. In particular, we can express in PL2 some of the modern foundations of mathematics. In 5.8, we will be talking about the foundations of PBM in more detail. For the time being, we are going to present two examples of expressing PBM foundations in PL2. According to modern foundations, we can explain natural numbers using only five axioms, called the **Peano axioms**, named after Giuseppe Peano (1858 - 1932). Expressed in natural language, we can find them, for instance, in [48][87] in the following form:

Definition 10 (Natural Numbers)

*The set $\mathbb{N}$ of **natural numbers** is characterized by the following axioms:*

[86] [25] p. 119
[87] p. 11, (own translation from German)

- ***Axiom 1**: $\mathbb{N}$ is not empty, in particular $0 \in \mathbb{N}$.*
- ***Axiom 2**: For every $n \in \mathbb{N}$ there is a unique element $n^* \in \mathbb{N}$ such that $n^* \neq n$, called the **successor** of n.*
- ***Axiom 3**: No element of $\mathbb{N}$ has 0 as its successor.*
- ***Axiom 4**: If $n^* = m^*$ for any two elements of $n, m \in \mathbb{N}$, then $n = m$ (being the successor is injective).*
- ***Axiom 5 (complete induction)**: If T is a subset of $\mathbb{N}$ with the property that $0 \in T$ (base case) and that from $t \in T$ it follows that $t^* \in T$ (induction step), then $T = \mathbb{N}$.*

Some of the Peano axioms cannot be expressed using PL1 alone. This is because they make use of symbols like "=" or "≠". For another reason, the fifth axiom - the *complete induction* - cannot be expressed in PL1, either. By introducing for n^* the functional term $Succ(n)$[88], we can express the fifth Peano axiom in PL2 as follows[89]:

$$\forall\xi((\xi(0) \wedge \forall n(\xi(n) \Rightarrow \xi(Succ(n)))) \Rightarrow \forall n(\xi(n))).$$

Note that we are able to express the axiom without referring to set-theoretical concepts like *"T being a subset"* of natural numbers. This is possible since, instead of saying *"subset of natural numbers"*, we require all predicates ξ to fulfill some condition, namely $\forall n(\xi(n))$ shall be concluded in case $(\xi(0) \wedge \forall n(\xi(n) \Rightarrow \xi(Succ(n))))$ is fulfilled.

The Peano axioms are not the only modern foundation of mathematics that requires PL2. As the **Definition** 10 demonstrates, some of the Peano axioms use set-theoretic notions, for instance *"non-empty"*, *"is subset of"*, etc. The *set theory* itself is another modern foundation that is based on the so-called **Zermelo-Fraenkel (ZF) axioms**, named after Ernst Zermelo (1871 - 1953) and Abraham Fraenkel (1891 - 1965). All ZF axioms can be found in the Appendix 9.2. A prominent example ZF-axiom we cannot express in PL1 but can express in PL2, is the **axiom of separation**. It states that if x is a set and ϕ is *any* predicate, then there exists a set y consisting exactly of those elements of x that fulfill the predicate ϕ. In other words, this holds for *all* predicates ϕ and we can state the axiom in PL2 as follows:

$$\forall x(\exists y(\forall\phi(\forall u(u \in y \Leftrightarrow (u \in x \wedge \phi(u)))))).$$

Only few undergraduate mathematicians are aware of the fact that it is this ZF axiom that justifies the *set-builder notation*[90]

$$y := \{u \in x \mid \phi(u)\}$$

[88] This functional term could have the interpretation of a function assigning the value $n + 1$ to n, but this interpretation is irrelevant for the example.
[89] compare [19] p. 148
[90] [25] p. 76

that can be used with some arbitrary (but fixed) predicate ϕ, in which the variable u is bound. The set-builder notation is very common in PBM.

5.4.6 Higher-order Logics PLm

In PL0, there are no quantors at all. In PL1, quantors can bound variables of individual elements of a domain of discourse $\mathbb{D}$. In PL2, we allow quantors to bound also variables denoting PL1 functional terms and PL1 predicates. Are there any higher-order predicate logics, for example, PL3 or PL4? The answer is *yes*. First of all, in a predicate logic PLm for some $m \geq 3$, we allow quantors to bound functional terms and predicates of PL(m-1). The semantics of such expressions becomes, however, hard to imagine, even for versed mathematicians. We can formalize it set-theoretically as follows: Recall that the interpretation $I_{\mathbb{D}}(\phi)$ of a PL1 predicate ϕ with n arguments is a mathematical relation, i.e., a subset of $\mathbb{D}^n$. By analogy, if ϕ is a PL2 predicate with n arguments, $I_{\mathbb{D}}(\phi)$ denotes a subset or subsets of $\mathbb{D}^n$[91], i.e. a relation on the *power set* $I_{\mathbb{D}}(\phi) \subseteq \mathcal{P}(\mathbb{D}^n)$. In the general case, in which ϕ is a PLm predicate ($m \geq 3$) with $n \geq 0$ arguments, $I_{\mathbb{D}}(\phi)$ a relation on the $m-1$-th repetition of building the power set $I_{\mathbb{D}}(\phi) \subseteq \mathcal{P}^{m-1}(\mathbb{D}^n)$. Keep in mind that despite a theoretical possibility of higher-order predicate logics, PL2 is sufficient to express most predicates in modern PBM.

5.4.7 Conclusions

Propositional logic PL0 and predicate logics PLm (for $m \geq 1$) are artificial languages developed in the 19th and 20th century, allowing to express logical statements in PBM formally. PL0 allows us to express logical statements, while PL2 allows us to express some foundational mathematical axioms using PL2 in an unambiguous, absolute way. The importance of predicate logics PLn lies in the fact that they allow much richer syntax than PL0. The importance of PL0 lies, on the other hand, in the fact that it is the constituent of all higher predicate logics: Loosely speaking, once we have an interpretation of a predicate, we can treat it like it was a proposition. On a symbolical level, we can create compound predicates out of prime predicates, just like we can create compound propositions out of prime propositions. For these reasons and their importance for being able to express PBM, we decide that predicate logics (containing propositional logic) has to be integral parts of FPL:

Requirement 11 (Higher-Order Predicate Logic as Part of FPL's Syntax)

> *The language FPL* MUST *incorporate propositional and predicate logic (at least PL2) and distinguish* ***variables***, ***constants***, ***functional terms***, ***predicates***, ***quantors***, *and logic-specific keywords for the truth values* ***true*** *and* ***false***, *as well as logical expressions to build compound propositions and predicates, especially* ***conjunction***, ***disjunction***, ***implication***, ***equivalence***.

[91] [19] p. 147

5.5 PL0 and PLn vs. Building Blocks of PBM

5.5.1 Objective and Approach

We mentioned in 5.4 that PL0 is far too simple to express PBM and that we can express in PL2 significant parts of PBM. Therefore, we decided to formulate the **Requirement** 11. In this subsection, we will address the question, whether PL0 and PLn are rich enough to express all eight building blocks of PBM we have identified in 5.3.

The key to answering this question will be *semantics*. In natural languages, semantics is either *"the study of the meanings of words and phrases"* or the *"the meaning of words, phrases, or systems"*[92]. In PL0, PLn, and PBM, semantics boils down to answering the question of assigning them a clear truth value $\top$ or $\bot$.

5.5.2 The Truth of PBM Axioms and Theorem-like Building Blocks

As we have seen in 5.2, mathematicians presume the truth of *axioms*. We can assign axioms a truth value, namely $\top$. They are propositions in the PL0 sense. In 5.2 we also showed that the purpose of providing a proof for a theorem-like building block, i.e., a *theorem*, *proposition*, *lemma*, or *corollary* is to establish the truth of a corresponding building block. Thus, they are, like axioms, propositions in PL0 sense. According to our **Requirement** 8, this also holds in case a mathematician decides to omit a proof in an FPL publication. We are, therefore, able to make the following observation:

Observation 6 (PBM Building Blocks Being Propositions in PL0 Sense)

> *The building blocks axioms, theorems, lemmas, corollaries, and propositions are "propostions" in the PL0 sense since we can assign them a truth value. Usually, in PBM, they have the value $\top$ (true) since mathematicians claim to have a model for them.*

Of the eight building blocks of PBM, only three building blocks remain, for which we cannot answer the question quickly, if they are propositions in the PL0 sense: *conjectures*, *definitions*, and *proofs*.

5.5.3 PBM Conjectures and Conditional Propositions in PL0

As we have learned in 5.2 and 5.3, a conjecture is different from valid theorem-like blocks, for which authors occasionally omit known proofs in a text. In contrast, a conjecture is an expression for which a proof is unknown. Thus, it could be true or be false or none. We do not know. We could, therefore, conclude that conjectures are *not* propositions in PL0 sense. However, things are more complicated than that. Occasionally, mathematicians *assume* the truth of unproven conjectures to prove statements that would become true if the underlying conjecture was true. For instance, there are many implications of a positive answer to the famous conjecture known as the *Riemann hypothesis*[93] (RM). It

[92] according to the Oxford Learners Dictionaries [87]
[93] [47] p. 226

draws us back from what we hoped to have established already in the **Observation** 6. Such *"conditionally proven statements"* may also occur in real-life PBM publications, and they cover theorem-like building blocks that are not, in the strict sense, propositions. We can establish their truth only *given* some conjecture (like RM) is valid. According to our goal of creating FPL which would facilitate expressing real PBM contents, we have to require that computers will be able to deal with constellations allowing conjectures to be propositions in PL0 sense. It brings us to the following observation and requirement:

Observation 7 (Conjectures as Conditional PL0 Propositions)

> *The building block of a conjecture can be considered propositions in PL0 sense since they are either* true *or* false.

Requirement 12 (PL0 Propositions in FPL Based on Conjectures)

> *FPL* MUST *allow referring to conjectures (or their negations) in the proofs of theorem-like building blocks as if they were true propositions in the PL0 sense. Those derived theorem-like building blocks become then* ***conditional*** *propositions in PL0 sense, i.e., their truth or falseness holds only given the assumed conjecture (or its assumed negation). Every FPL interpreter* MUST *clearly distinguish between propositions in PL0 sense and conditional propositions that depend on the truth of unproven conjectures.*

5.5.4 PBM Proofs Are Propositions in PL0 Sense

On a micro level, each proof in PBM consists of single logical arguments that are created by applying so-called **inference rules**. Inference rules are deductive laws that most mathematicians accept as valid because they are either tautologies in the PL0 sense or for other reasons[94].

On a macro level, proofs can have very different forms. A discussion of typical forms, including **direct proofs**, proofs **by contraposition**, **by contradiction**, **by induction**, or proofs of **biconditionals** $\Leftrightarrow$ can be found, for instance, in [32]. For the time being, we only state that a mathematical proof is, on a macro level, a (compound) proposition in the PL0 sense. We conclude this in the following observation:

Observation 8 (PBM Proofs Are Propositions in PL0 Sense)

> *The proof building block is a "propostion" in the PL0 sense. From a formal point of view, a PBM proof is a conjunction of steps referring to proceeding propositions (e.g., axioms, proven theorems, or inference rules of the proof).*

A key feature of mathematical proofs that is key for the specification of FPL is nesting and linking. Nesting occurs, for instance, if we include a proof by contraposition to prove

[94] They can be just further *"common sense"* conventions acceptable by mathematicians as valid logical arguments. Later, we will present examples of inference rules that are not tautologies in the strict PL0 sense.

something that is only a single step in the scope o a proof by induction, or vice versa[95] Linking occurs if we refer to one or more of previously proven steps in a proof in a logical step in a proof.

Nesting and linking rules of reference on a micro level and proofs on a macro level create many possible mathematical proofs. The grammar of FPL has to support it, and we formulate a new requirement:

Requirement 13 (Nesting and Linking of Proofs)

> *FPL's grammar* MUST *support nesting and linking different logical steps and proofs into more complex ones.*

5.5.5 PBM Definitions Are Not Propositions in PL0 Sense

As we have seen in 5.2, the purpose of *definitions* in PBM is to introduce new mathematical terms and fix their meaning and notation unambiguously. Therefore, definitions are building blocks of PBM that *enhance* its syntax rather than are formulated in its syntax. Note that predicates in PLn and propositions in PL0 have a predefined, fixed syntax and remain meaningless. Neither a domain of discourse $\mathbb{D}$ nor an interpretation $I_{\mathbb{D}}$ are ever *part* of their respective syntax, without which the syntax remains meaningless. Therefore, definitions are a meta-language of PBM, and they are exactly the means to fix the domain of discourses and interpretations of the predicates that we express using the other seven building blocks. Altogether, it does not make any sense to assign a truth value to a definition. Otherwise, it would be an array of meaningless symbols, and we would need a domain of discourse to interpret it. However, PBM definitions are never meaningless without an interpretation. They rather provide this interpretation. We observe:

Observation 9 (PBM Definitions Are Not Propositions in PL0 Sense)

> *The PBM building block of a* ***definition*** *is not a "proposition" in the PL0 sense since we cannot assign it a truth value. Definitions are a meta-language that allows increasing the syntax and semantics of PBM be defining and fixing new domains of discourse and new interpretations.*

If we could express all PBM in some higher-order logic PLn, we could also express mathematical definitions. However, it follows from **Observation** 9 that we cannot express definitions in the syntax of PL0, PL1, PL2, or any other higher-order logic. If we could, it would contradict the observation. A language cannot enhance its syntax out of its meaningless symbols. We need a meta-language, in which we can state the interpretations of the symbols and variables that are otherwise meaningless.

Some books suggest the opposite. Ebbinhaus gives the following example[96]:

[95] compare [32] p. 56, discussing another example of nesting proofs, namely how to prove a biconditional using two proofs for a conditional.
[96] [7] pp. 22-25

- A new predicate $\phi_T(x, y)$ abbreviating that a set x is a subset of another set y:

$$\phi_T(x, y) := \forall z(z \in x \Rightarrow z \in y).$$

- A new symbol $x \subset y$ to abbreviate the predicate $\phi_T(x, y)$

$$x \subset y := \phi_T(x, y).$$

- A new functional term $\phi_P(x, y)$ abbreviating that a set x is a *power set* of another set y:

$$\phi_P(x, y) := \forall z(z \in y \Rightarrow z \subset x).$$

- An new symbol $Pot(x)$ to abbreviate $\phi_P(x, y)$:

$$Pot(x) = y := \phi_P(x, y).$$

This way, we can use the new symbols, predicates, or functional terms as abbreviations of previous ones. For instance, instead of writing

$$\forall z(z \in b \Rightarrow (z \in c \wedge \forall x(x \in c \Rightarrow \forall y(y \in x \Rightarrow y \in a))))$$

we can simply state

$$Pot(a) \subset b.$$

The misleading idea is to conclude that if definitions, like in this case, abbreviate predicates or functional terms, they are like predicates or functional terms. Thus, we could wrongly conclude that the syntax of PLn supports expressing definitions, which is not the case! Taking a closer look at the above example, definitions introduce four new symbols: $\phi_T(x, y)$, $\phi_T(x, y)$, Pot and $\subset$. They were not part of the original vocabulary. Their meaning is fixed by the symbol $:=$, which is not part of PLn syntax. Even if we replaced $:=$ by $\Leftrightarrow$, a computer interpreting the text would get problems at the latest when it tried to interpret the variable symbols x, y, z, a, b, c or the symbol $\in$ that is not part of the PLn syntax. They are meaningless. We will need other definitions stating what $\in$ and a *"set"* are and somehow tell the computer that the variables denote sets. We cannot define $\in$ be itself inside the syntax of set theory. We need a meta-language of definitions to accomplish that. From the above analysis, we can formulate another requirement for FPL:

Requirement 14 (Metasyntax vs. New Syntax and Domains of Discourses)

> *FPL* SHALL *use the bulding blocks of definitions as a* ***meta syntax*** *to introduce new syntax and to introduce new domains of discourses, in which end-users can interpret the remaining FPL code.*

5.5.6 Feasibility and Possible Practical Issues

In PBM, the purpose of introducing and fixing the syntax and semantics *on demand* is met by the construct of definition blocks. Syntactically, definitions are part of a meta-language being *"outside"* the syntax of any given PBM theory. Otherwise, they would not increase its syntax each time they occur! For this reason, we are tempted to meet **Requirement** 14 by simply allowing definition blocks in FPL. Unfortunately, allowing definition blocks in FPL alone is not a sufficient solution. There are potential major difficulties we may encounter if trying to design FPL to fulfill **Requirement** 14. To demonstrate these difficulties, let's consider how we could accomplish the above example of defining natural numbers in FPL:

- The stock of the symbols required to express all natural numbers is potentially infinite. We have to be able to introduce infinitely many (meaningless) expressions like *"0"*, *"1"*, *"2"*, etc. telling the FPL interpreter to accept them not only as allowed symbols but also to parse exactly those symbols as *constants* in the sense of PL1.
- For each of the infinitely many constants *"0"*, *"1"*, *"2"*, etc., we have to fix their semantics by referring to a domain of discourse of natural numbers, fixing the interpretation of the symbols 0, 1, 2 as consecutive natural numbers.

The first difficulty alone seems hard to accomplish, but things get even worse for the second difficulty. Hold on for a second to grasp why this is a big problem: Are we not running into a **vicious circle**? In order to introduce in FPL what natural numbers *are*, we have to fix their syntax and semantics. However, fixing the semantics of natural numbers in FPL requires us to refer to an external concept - the domain of discourse $\mathbb{D}$, *being* the set of natural numbers we are still trying to define in FPL! In 5.11, we are going to accept this kind of vicious circle as a fundamental limitation in designing FPL and will investigate ways to address it in FPL. At this stage, let us only observe how this problem is handled in real PBM books. In [48] 11ff., natural numbers are introduced starting with the above-mentioned *Peano axioms* (**Definition** 10 being numbered in this source as *"Definition 1.1"*). Note that the functional term of the successor of a natural number $Succ(n)$ is denoted by n^*. Then, the author states[97]:

> *"Note 1.2. Successively, by Definition 1.1, we determine the symbols we are well-acquainted with:"*
>
> $$1 := 0^*, 2 := 1^* = 0^{**}, 3 := 2^* = 1^{**} = 0^{***}, \ldots, \tag{1}$$

If you do not notice the difficulty yet, imagine building a computer that can parse this natural language text and interpret it. It has to create a clear internal representation of the *domain of discourse* fixed in **Definition** 10, so it will finally *"know"* all the symbols allowed for expressing natural numbers. The fact that the source [48] uses a definition

[97] own transl. from German

block to define natural numbers doesn't solve the following technical problems when we try to write this definition down in FPL and be parseable and interpretable by both: humans *and* computers:

1. As we have seen, the **Definition** 10 is, like any other definition in PBM, not expressible in PL2.
2. **Definition** 10 nests the building blocks of axioms inside the building block of a definition, a phenomenon quite common in mathematics but new to what we have already seen in our analysis of the structure of PBM language performed in 5.1, 5.2, and 5.3.
3. The **Definition** 10 fixes the fundamental mathematical concept of natural numbers while using another fundamental mathematical notion: *"sets"* and their language.
4. Taking a closer look at the **Definition** 10, it does fix the semantics and syntax, but only of two new functional symbols: "0" and "n^*," while making use of the symbols "=" and "$\in$" that are presumed to be known. The *"Note 1.2"* quoted above is stated *outside* the definition block (it is not part of any of the eight building blocks of PBM we have identified in 5.3). It suddenly introduces infinitely many new symbols "1", "2", "3", etc., using the assignment symbol ":=" (presumed to be known by the reader) as well as the notation of dots "$,\ldots,$" in the sequence 1, indicating that these assignments continue ad infinitum. *"Note 1.2"* justifies the introduction of infinitely many new symbols relying on readers being *"well-acquainted with"* these symbols. A human reader might accept this, but building a computer accepting this explanation might not be feasible.

We will show how we address all these practical difficulties in FPL in 7. For the time being, let us draft possible solutions, some of them resulting in additional requirements for FPL:

Ad (1) Obviously, FPL has not only to incorporate predicates and functional terms of PLm but also to allow formulating definitions for them. In technical terms, instead of defining the *" natural numbers"*, we could rather define a new mathematical *type* `Nat` of a natural number. Expressing that a variable in FPL is a natural number would be comparable to how we declare typed variables in some modern programming languages like C# or java. A new requirement follows from that:

Requirement 15 (Definitions to Introduce New Types in FPL)

> *FPL definitions* MUST *introduce new types to be used to declare variables of this type.*

Ad (2) In FPL, we will have to allow nesting axioms inside definitions, as we encounter this phenomenon in real-world PBM publications. As we have seen above, axioms are building blocks that are completely expressible as predicates in some PLm logic.

Moreover, as we can see in the case of **Definition** 10, axioms that are nested inside a definition might refer to the same mathematical type that is yet to be defined. For instance, we may encounter the axiom *"0 is a natural number"* stated inside the definition describing what a natural number is. This kind of self-reference is common in mathematical definitions nesting axioms of this kind. It is also common in modern object-oriented programming languages. We will elaborate on this more in 5.13 and define corresponding requirements for FPL definitions.

Ad (3) This difficulty seems to be manageable: As we have demonstrated in the case of the 5th Peano axiom, PL2 allows us to express the Peano axioms without making any references to set theory. We might be able to express Peano axioms in FPL without referring to set-theory while using PL2. Nevertheless, one difficulty remains: How can we express in PL2 an axiom like *"0 is a natural number"* without referring to set-theoretical notation like "$0 \in \mathbb{N}$"? We lack a meta-language to express this kind of predicates in PL2. This is a new requirement:

Requirement 16 (`is` Operator in FPL)

> *FPL* MUST *allow stating that (or checking if) a variable is of a given type. This* MAY *be accomplished using an operator like* `is`*. Given a variable x and based on the context, an expression like $is(x, < type >)$ would either check if x is of a given $< type >$ or state that it is this $< type >$. The first context might occur in a premise block of an implication*
>
> $$(< premisse > \Rightarrow < conclusion >),$$
>
> *the second context might occur in the conclusion block of a logical implication, or the context of an axiom.*

Taking **Requirement** 15 as well as **Requirement** 16 into consideration, we could express the axiom $0 \in \mathbb{N}$ inside the **Definition** 10 for instance as follows:

Example 2 (Possible Formulation of the 1st Peano Axiom in FPL)

```
def Nat()
{
   axiom is(0, Nat);
   // ...  further axioms
}
```

Ad (4) The notion *"well-acquainted with"* appeals to the reader's previous knowledge of notation of (potentially *infinitely* many) natural numbers in decimal system (for instance 1000 means in this notation the number *"tausend"*, but not the binary number *"eight"*). Obviously, **Definition** 10 combined with *"Note 1.2"* provide an informal *interpretation* $I_{\mathbb{D}}$ for human readers, who are now willing to accept the

axioms inside **Definition** 10 (for instance the expression $0 \in \mathbb{N}$) as well as expressions like $199 = 198^*$ after reading *"Note 1.2"* as *true* statements and identify expression like $20 = 17^*$ as *false* statements. Those expressions would otherwise be meaningless arrays of symbols. In terms of theoretical computer science, we obviously need a formal language FPL that is capable to dynamically increase the syntax accepted by its own *parser*[98] and to dynamically increase the interpretation of the introduced symbols by its own *interpreter*[99].

In the **Example** 3, the expression `axiom is(0, Nat)` is basically not parseable, because the symbol `0` is not (yet) known for the FPL parser. What we need is a possibility to fix the notation (for instance in the decimal system) of natural numbers that would should be accepted. This leads us to the following requirement:

Requirement 17 (Flexible Notation Inside FPL Definitions)

> *FPL* MUST *allow fixing a new notation inside a definition for objects of the type defined inside that definition.*

Taking **Requirement** 17 into consideration, **Example** 3 has to be extended to

Example 3 (Possible New Notation in FPL)

```
def Nat()
{
   extension /\d+/;
   axiom is(0, Nat);
   // ...  further axioms
}
```

The `extension` keyword SHOULD have the following effect on the overall FPL syntax:

- Everything that is written between `extension` and `;` is a new notation of symbols that the FPL parser is going to accept when parsing FPL code.
- If the FPL parser encounters a sequence of symbols that it cannot consume as native FPL keywords or symbols (like the symbol `0` in the expression `axiom is(0, Nat)`), it has to apply a *"dictionary"* of user-defined types and their notations. In this case the type `Nat` corresponds to the notation `/\d+/`, which is a regular expression accepting positive integers written in the decimal system.
- If the FPL parser accepts the new sequence of symbols applying a notation, it will tokenize it as an instance of the corresponding type.

[98] A **parser** is a computer program performing a syntactical analysis to confirm if a given input code belongs to a specific language ([39] p. 708).

[99] In contrast to a compiler translating a given code into an executable computer language, an **interpreter** analyses every statement and declaration in a code and executes it immediately ([39] p. 322).

This approach seems to solve the problem of introducing potentially infinitely many symbols that the FPL parser has to accept as instances of the newly introduced type `Nat`. But how about telling the FPL interpreter to interpret them as consecutive applications of a the functional symbol

```
Succ(n):=Succ(...(Succ(0))...)?
```

In PoC, we will take a slightly modified approach using a preprocessor directive and not a keyword to meet **Requirement** 17. *"Definition 1.1"* in combination with *"Note 1.2"* in [48] create an interpretation of these symbols and a model of natural numbers, making the axioms inside the definition or expressions like $199 = 198^*$ true statements for human readers.

5.5.7 Conclusions

We learned that we can express axioms, theorems, propositions, lemmas, corollaries, proofs, and conjectures in PLn (see **Observation** 6, **Observation** 8, and **Requirement** 12) but we cannot express definitions in the PLn (see **Observation** 9). Because definitions fix, among others, the notation of mathematical objects[100], they *define new* syntax and interpretation of PBM rather than are part of that syntax.

5.6 Formal Systems, Axiomatic Method, and Provability

5.6.1 Objective

In the proceeding two subsections 5.4 and 5.5 we investigated how PBM can be expressed using propositional PL0 and predicate logics PLn, $n \geq 1$. In this subsection, we will demonstrate how we can prove theorem-like PBM building blocks mentioned in the **Observation** 6 using *inference rules*. We will model the proving process using the so-called *formal systems* and learn an example of a formal system that is imporant for PBM: the *Hilbert calculus*. We will also introduce and explain the concept of *provability*.

The work in this subsection is crucial for designing FPL since proving mathematical statements is the heart of what PBM is all about. Moreover, we will draft the fundamental difference between the provability and the truth of mathematical statements.

5.6.2 Boole's PL0 and Its Limitations

Given a mathematical statement that was to be proven, Boole's idea was to use tautologies to mechanically transform the PL0 formula of the original mathematical statement into a simpler one. Finally reaching precisely one of the most simple symbols $\top$ (*true*) or $\bot$ (*false*). Since tautologies preserve the original mathematical statement's truth, one can decide if the original statement is true if and only if one reaches $\top$. Otherwise, the original statement must have been a contradiction. This was a big deal for mathematical

[100] compare 5.2

proofs. Of course, tautologies and contradictions were known before Boole, and mathematicians had proved mathematical statements intuitively for centuries. However, it was now the first time in history mathematical statements could be debunked as valid statements or as contradictions on a purely syntactic level, i.e., in some absolute, precise way independent from subjective interpretations, mathematical genius, and intuition. Leibniz's dream of a *"Characteristica universalis"* seemed within reach.

Still, the promise of PL0 to come closer to a *"Characteristica universalis"* had to be quickly revised for many reasons. First of all, the syntax of PL0 was far too simple to express real PBM, and that is why logicians invented PL1, PL2, and higher-order logics after Boole. Second, the stocks of applicable tautologies to simplify a mathematical statement are potentially *infinite*. We either have to limit it radically or find some other way to decide whether a formula is a tautology for feasibility reasons. It took more than 130 years after Boole's publication when it turned out that it is fundamentally hard to answer even the seemingly simpler question: Is a given PL0 formula satisfiable? This decision problem became later known in theoretical computer science as SAT. Stephen Cook, a Canadian computer scientist, proved in 1971 that SAT is *NP-complete*. NP-complete problems are particularly hard to be solved, even if using powerful computers, because the number of calculation steps grows exponentially with the length of the SAT input formula. All attempts to find a fast computer program that would solve SAT (or any other NP-complete problem) for longer formulas found in praxis have remained unsuccessful for another 60 years until now. Today, many mathematicians and computer scientists believe that NP-complete problems (like SAT) are fundamentally not tractable. Final proof for SAT being fundamentally intractable (or otherwise finding a computer program solving SAT efficiently) remains one of the great still open questions of modern mathematics and computer science, known as the $P \stackrel{?}{=} NP$ problem.[101]

The above analysis leads us to a first possible fundamental limitation to the design of FPL: We cannot expect being able to build computers that help us efficiently prove or disprove that a given mathematical theory expressed in FPL is free of contradictions. Such a computer would have to solve SAT efficiently, which is NP-complete. We formulate this in a separate limitation:

Limitation 1 (SAT Is (Probably) Not Tractable)

> *The problem of deciding with a computer's aid if a given proposition in PL0 is satisfiable (not a contradiction) is NP-complete and, to date, broadly believed to be not tractable by computers.*

5.6.3 Formal Systems and Provability

A related problem to SAT in PL0 is the problem of deciding if a given formula is a tautology. Some promissing techniques have been developed in the 20th century to solve

[101] compare [47] 1ff.

it, like the *Hilbert calculus*, the *resolution calculus*, and the *tableau calculus*[102]. All these methods require that we first limit the potentially infinite stocks of possible inference rules to a *finite* set R so that a computer can apply them to simplify a given input formula. The inference rules in R are formulas in the form $\texttt{premise} \vdash \texttt{conclusion}$. For instance, R could consists of the following four inference rules:

1. **Modus ponens**: $(p \wedge (p \Rightarrow q)) \vdash q$, based on the tautology $((p \wedge (p \Rightarrow q)) \Rightarrow q)$.. An example of such a logical argument is *"If it is raining and the streets are wet whenever it is raining, then the streets are wet."*.

2. **Modus tollens**: $(\neg(q) \wedge (p \Rightarrow q)) \vdash \not(p)$, based on the tautology $((\neg(q) \wedge (p \Rightarrow q)) \Rightarrow \neg(p))$. An example of such a logical argument is *"If the streets are not wet, and streets are wet whenever it is raining, then it is not raining."*

3. **Hypothetical syllogism**: $((p \Rightarrow q) \wedge (q \Rightarrow r)) \vdash (p \Rightarrow r)$, based on the corresponding tautology. An example is *"If the streets are wet whenever it is raining, and if the streets get slippery whenever they are wet, then whenever it is raining, the streets get slippery."*

4. **Disjunctive syllogism**: $(\neg(p) \wedge (p \vee q)) \vdash q$. For instance, *"If (either it is raining or the sun is shining (or both)) and if it is not raining, then the sun is shining."*

There are many other examples of inference rules, some of which are described in [101], [103], [36], [25], or [22]. A system with a finite set R of inference rules is called a **formal system**, or a **(logical) calculus**. *Formal systems* come close to Leibniz's vision of a *"Characteristica universalis"* since they *"allow us to code mathematical statements and to derive from the logical conclusions"*[103]. We will present an example of the Hilbert calculus since it is technically the most simple and inspired by how mathematicians do their proofs. Mathematical theories are much more complex than just collections of tautologies like R. When we formalize a mathematical axiom as a predicate in some higher-order logic, we will still need a *domain of discourse* and an interpretation to turn the meaningless predicate's formula into a PL0 proposition. Mathematicians assert the truth of such an axiom. In the Hilbert calculus, the formulas we get from the axioms complement the finite set of inference rules we choose for our formal system. Before we present the example for the Hilbert calculus, let us formalize what should happen to prove a mathematical statement using this calculus. For this purpose, we need the *provability relation*[104].

Definition 11 (Provability Relation)

*Let $\Sigma := (A, R)$ with A being the set of all axiom formulas and R being the set of all inference rules. We say that the formula q is a **direct consequence** of a formula p (denoted by $p \to_\Sigma q$) if there is an inference rule $\rho \in R$ such that $p = \rho(q)$. We*

[102] [19] pp. 96-111, 124-138
[103] [22] p. 2, transl. from German
[104] [57] 28ff.

say that q is ***derivable*** *in Σ (denoted by $\Sigma \vdash q$), if there a sequence of formulas $p_1, \ldots, p_n$ with $q = p_n$ and*

- *$p_i \in A$ for $i = 1, \ldots, n$, or*
- *there is a formula p_{i-1} with $p_{i-1} \rightarrow_\Sigma p_i$ for $i = 2, \ldots, n$.*

We will denote the set of all formulas that are derivable in Σ by $\mathrm{Der}(\Sigma)$. Thus, $\Sigma \vdash p$ if and only if $p \in \mathrm{Der}(\Sigma)$.

Now, we are ready to present an example of the *Hilbert calculus* in work[105]. We will demonstrate how we can derive a PBM theorem mechanically from a set of axioms and inference rules expressed in PL1. We will be able to do so on a purely syntactical level, without really *"knowing"* what the symbols we are manipulating mean. We want to mechanically prove the following statement *"There exists a thing that is greater than any two other given things."* To do so, we will have to formalize the statement in the following way:

- Let *Greater* be a binary proposition (i.e., taking two arguments).
- We formalize the word *"thing"* by applying variables from the list a, b, c, x, y, z.
- Let $\exists x(p(x))$ denote: *"There exists an x such that a the predicate $p(x)$ holds."*

Based on these formalizations, we can state the above sentence expressed in natural language as a formula in an artificial language of a *formal system*, let us call it T. In terms of building blocks of PBM, this would be a theorem-like building block, for instance a *lemma*.

Example 4 (A Lemma in T)

$$\exists x(Greater(x, y) \wedge Greater(x, z)).$$

Now, let us prove this lemma *"mechanically"*. In order to do so, we will have to assume the truth of the following axioms:

Example 5 (Axioms of T)

1. *A1:* $Greater(a, b)$.
2. *A2:* $Greater(b, c)$.
3. *A3:* $((Greater(x, y) \wedge Greater(y, z)) \Rightarrow Greater(x, z))$.

We will be also using the following inference rules:

Example 6 (Inference Rules of T)

1. *R1: Modus Ponens*

[105] [36] 8ff gives a similar example.

2. *R2: Given p, q being propositions or proceeding logical steps, then $(p \wedge q)$.*
3. *R3: If $p(c)$ then $\exists x(p(x))$.*

We are now able to prove the lemma mechanically:

Example 7 (Mechanical Proof Lemma in T)

1. $Greater(a, b)$	(A1)
2. $Greater(b, c)$	(A2)
3. $(Greater(a, b) \wedge Greater(b, c))$	(1, 2, R2)
4. $((Greater(a, b) \wedge Greater(b, c)) \Rightarrow Greater(a, c))$	(3, A3)
5. $Greater(a, c)$	(4, R1)
6. $(Greater(a, c) \wedge Greater(a, b))$	(5, A1, R2)
7. $\exists x(Greater(x, y) \wedge Greater(x, z))$	(6, R3)

Pinter states that we can regard a mechanical proof like this as *"meaningless arrays of symbols; the fact that they have a meaning* to us *is irrelevant to the task of carrying out the proof. Thus, intuition is absent from a formal mathematical proof"*[106]. We can observe that each logical step in a formal proof has the syntax of a predicate. The rule of inference R2 is a generally accepted convention in mathematics: If we accept more than one logical argument to be accurate, we also accept their conjunction via $\wedge$ to be true. By this convention, we can therefore conclude that the whole proof is a conjunction of true statements.

5.6.4 Axiomatic Systems and Their Quality Requirements

The *Hilbert calculus*, example of which we saw in 5.6.3, is a formalization what in PBM is called the **axiomatic method** (sometimes also called the *deductive method*). Because this method will accompany us from now on very often in this document, we introduce the abbreviation AM. AM has been a method used in mathematics for millenia to deductively derive all statements formulated in PBM from *"a priori formulated basic assumptions"*[107]. It is the *"vital nerve of mathematics by what mathematics becomes a science"*[108].

A system consisting of axioms and rules of inference, from which we can derive (prove) theorems, is called an **axiomatic system**. Axiomatic systems and their quality requirements are described, for instance, in [15], [11], [20], and [36]. The following *metamathematical* features of axiomatic systems are desirable:

[106] [36] p. 8
[107] [20] p. 14
[108] [18] p. 33 (own transl. from German: *"Axiomatische Methode [...] ist der Lebensnerv der Mathematik, das, wodurch die Mathematik zur Wissenschaft wird."*)

1. **Consistency** (being free of contradictions).
   An axiomatic system is **consistent** if it never allows to derive a formula g and its negation $\neg g$. Equivalently, the conjunction of all of its axioms should never allow to derive a contradiction: $a_1 \wedge a_2 \wedge \ldots \wedge a_n \nvdash \bot$. Consistency is a syntactical feature. It is related only to the derivability of formulas, independently from their truth.

2. **Soundness** (ability to prove only statements that are true).
   Similarly, an axiomatic system is **sound**, if it has a model. In other words, all axioms should indeed be interpretable as being true. They should be *"plausible"*. Soundness is a semantical feature and creates a connection between the derivability of formulas and their truth.

3. **Completeness** (ability to prove all true statements).
   The axiomatic system is said to be **complete** if we can derive in them all true statements (i.e., all existing tautologies). Like soundness, completeness is a semantical feature.

4. **Independence** (ability to negate axioms without causing contradictions).
   An axiomatic system is **independent** if we cannot deduce one of its axioms from the remaining axioms.[109]. It is another, not crucial characteristic of an axiomatic system, and it is a *"mere question of aesthetics"*[110]. An independent axiomatic system allows the negation of an axiom, leading to a new, consistent theory. This way, we also can prove the independence of a given axiomatic system containing the axioms $a_1, a_2, \ldots, a_n$. By finding two models proving the truth of the formulas $a_1 \wedge \ldots a_i, \ldots, a_n$ and $a_1 \wedge \ldots \neg a_i, \ldots, a_n$, we can prove that the axiom a_i is independent from the remaining axioms. We have to repeat this procedure for all axioms[111]. Attempts to prove the independence of the *parallel axiom* from the remaining four axioms of Euclid's *"Elements"* led to the discovery of consistent non-Euclidean geometries. In the 19th century and independently from each other, János Bolyai (1802 - 1860) and Nikolai Lobachevsky (1792 - 1856) replaced the parallel axiom by an assumption that contradicted it[112]. Independence is a syntactical feature, like consistency.

5. **Negation-completeness** (ability to prove either a statement or its negation).
   We should formulate axioms in an axiomatic system in such a way that we can prove either a statement or its negation based on this axiomatic system. Negation-completeness is a syntactical feature.

6. **Completeness of the set of axioms** (no more axioms can be added).
   The completeness of axioms in the axiomatic system should not be confused with the completeness of the system. The axioms $a_1, a_2, \ldots, a_n$ in the axiomatic system are **complete**, if by adding a new axiom a_{n+1} to would make it inconsistent.

[109] [15] p. 197
[110] [?] p. 268)
[111] compare [15] p. 204
[112] [36] p. 5

Complete axiomatic systems are in this sense *"maximal"*.[113] However, unlike consistency and soundness, the completeness of an axiomatic system is not crucial.[114]. It is a syntactical feature because it is closely related to consistency.

5.6.5 Theorems About Quality Requirements of Axiomatic Systems

In 20th century, formalists were able to use AM as a method to prove many metaresults about itself, a selection of which we present below. We formulate them as an observation to better refer to them when we specify FPL. Proofs can be found for instance in [22], [19], or [41].

Observation 10 (Metaresults about AM)

1. *The soundness of an axiomatic system implies its correctness, but not vice versa.*
2. *The completeness of an axiomatic system together with the condition that for all formulas we have either* $\models g$ *or* $\not\models g$ *imply its negation-completeness, but not all negation-complete systems are complete.*
3. *A* algorithm *exists to derive a formula* ($\vdash g$) *inside an axiomatic system, if it is consistent and negation-complete.*
4. *An algorithm exists to find a model of a formula* ($\models g$) *inside an axiomatic system, only if it is sound and complete. Because soundness and completeness imply* $\vdash g \Leftrightarrow \models g$*, this follows immediately from 3.*

Theorems 3 and 4 are originally formulated as so-called *solvable decision problems*[115]. We choose the formulation *"an algorithm exists"* by which we mean a *deterministic terminating* algorithm. The computer-scientific theory of decidability, computability, or determinism goes far beyond the scope of this document. For more information, please refer, for instance, to [19]. What we mean is that we can (in principle) write a computer program that would solve 3 or 4 for any given formula and any given axiomatic system with the required properties in a *finite* number of steps. The theorems do not provide any information whatsoever about the efficiency of such algorithms or how to find them. They state only that such algorithms exist. In contrast, theorems 3 and 4 do not apply the non-existence of such algorithms for concrete cases of axiomatic systems that do not fulfill the required properties. They only state that in those cases, existence is not ensured.

5.6.6 Provability and Satisfiability Are Independent of Syntax

In analogy to the model relation $\models p$ introduced above, the provability relation $\vdash p$ is completely separated from the formal system's syntax. Note that $p \rightarrow_\Sigma q$ has nothing to

[113] [15] p. 198
[114] [?] p. 268)
[115] [22] 83ff.

do with the implication $p \Rightarrow q$[116]. The latter is part of the syntax of the formula we are going to prove. The first is a meta-expression about our ability to prove it. The direct consequence $p \rightarrow_\Sigma q$ of two formulas p and q and the derivability $\Sigma \vdash q$ of the formula q express meta-qualities of formulas. Just like it was possible to choose different and *replaceable* interpretations I_1, I_2 with $I_1 \models q$ and $I_2 \not\models q$, it is now possible to replace Σ_1 by a Σ_2 to get different provability relations $\Sigma_1 \vdash q$ and $\Sigma_2 \nvdash q$, respectively to decide if q is a direct consequence of p ($p \rightarrow_{\Sigma_1} q$ or $p \not\rightarrow_{\Sigma_2} q$) without even changing the concrete syntax of p and q! The justification of transforming one formula into another (like it is done intuitively by mathematicians in real-life PBM proofs) can be explained as a purely mechanical application of inference rules, bare of any intuition! This insight might embarrass many mathematicians!

5.6.7 Conclusions

In this subsection, we covered a lot of theoretical stuff related to *formal systems*. We approached very rapidly a key insight still to be uncovered later: The *truth* and the *provability* of mathematical statements are different concepts fundamentally independent from each other. For formalists, a statement's truth is not necessarily the same as its provability and vice versa! It makes Leibniz's vision of a *"Characteristica universalis"* a dream that will never come true and will pose further serious limitations to FPL later in this document. However, we already know that the syntax of predicates in PLn is independent of the model relation $\models$ and the provability relation $\vdash$. This principal independence leads us to the following MUST requirement for FPL since it will prevent FPL from running into major conceptual problems when being applied to real PBM:

Requirement 18 (Configurable Axioms and Inference Rules in FPL)

> *FPL's syntax* MUST *allow end-users to freely configure the set of axioms and the set of inference rules on which they want to base a PBM theory.*

5.7 What is the 'Beauty of PBM'?

5.7.1 Objective

Many mathematicians consider mathematics as *"beautiful"*. In this last subsection of our analysis part, we want to ask what constitutes this beauty and how we can design FPL so that things mathematicians consider beautiful will stay so when expressed in the syntax FPL.

5.7.2 Analysis

In [112], there is a pretty concise list of quotes of more or less famous mathematicians about what they think makes mathematics beautiful. It is sufficient for our purposes to refer to this list and show how mathematicians try to describe mathematics's beauty.

[116] Nevertheless, there is a strong connection between the two, known as the *deduction theorem* ([25] p. 95, [19] p. 100).

1. According to Aristotle (384 B.C.-322 B.C.), beauty (in general) *"depends on size as well as symmetry"*.

2. Due to Bertrand Russell (1872-1970), mathematics *"constructs an ideal world where everything is perfect but true."*

3. Due to G. H. Hardy (1877 - 1947), the *"mathematician's patterns [...] must fit together harmoniously. [...] There is no permanent place [...] for ugly mathematics."*

4. Isaac Newton wrote about Leibniz's method for obtaining convergent series as *"very elegant"*.

5. When Serge Lang (1927 - 2005) asked his audience *""What does mathematics mean to you?""*, he received the answer *"The manipulation of numbers, the manipulation of structures."* He asked back: *"And if I had asked what music means to you, would you have answered: 'The manipulation of notes?'"*

5.7.3 Criteria for Beauty?

The above examples show that it is even for mathematicians hard to describe what makes mathematics *"beautiful"*. The problem is that *"you cannot recognize the beauty of mathematics unless you delve into it."*[117] It is, therefore, illusory to look for objective criteria we could use to *"measure"* the beauty of a given mathematical theorem. However, we can define our quality criteria that help avoid hindering the perception of beauty by mathematicians expressing their thoughts in FPL. One such criterion is flexible symbolic notation. Mathematicians always tend to write things possibly short, still expressing the same content. Many mathematicians would prefer to be able to write the Euler's equation in a concise way

$$e^{i\pi} + 1 = 0$$

and would perceive it more beautiful as any attempt to express the equation in prose, for instance,

> *"The Euler constant* e *to the power the imaginary unit* i *times the number* π *plus* 1 *equals* 0*,"*

or writing this equation in, say, some prefixed notation, like:

```
Equals(Add(Exp(Mul(ImaginaryUnit,Pi), 1), 0)
```

Because of that, we make the the following design decision:

Requirement 19 (Flexible Symbolic Notation)

> *The look&feel of FPL* SHOULD *be possibly similar to mathematical symbols and keywords. Ideally, it should exploit the possibilities of LaTeX.*

[117] [29] p. 11 (own transl. from German)

Another criterion we should address is an *"adequate"* depth of detail in mathematical proofs written in FPL. In his plea to write structured proofs instead of proofs written in prose[118], Leslie Lamport lists three common objections he encountered when students or mathematicians are confronted with *"structured proofs"*:

1. *"They become too complicated. [...] Being rigorous requires filling in missing details, which makes the proof longer."*
2. *"They don't explain why the proof works. [...] Mathematicians sometimes precede a proof with a proof sketch."* A purely structured proof would not allow explanations of the steps.
3. *"A proof should be great literature."* However, according to Lamport, *"Its beauty lies in its logical structure, not in its prose."*

No matter if these objections are justified or not, we have to accept them if we take user-friendliness of FPL seriously. Depending on the particular knowledge, some FPL users will need more details than others when writing or reading the same proof. Usually, mathematicians perceive a theorem and its proof as *"beautiful"* if it reveals some unexpected deep insights or hidden symmetries. In principle, every mathematician should be able to ask the *"why"* question, providing more and more details for every single logical step until the level of unprovable mathematical axioms is reached[119], from which a deep theorem follows. On the other hand, this level of detail is not necessary to understand the idea of mathematical proof and grasp the theorem's insight. A too extensive level of detail could even cramp the understanding of the theorem's proof and finally hinder the perception of its *"beauty"*.

So what is the *"adequate"* depth of detail we should seek in FPL? The following design decision specifies this fuzzy criterion more concretely:

Requirement 20 (Variable Depth of Details in Proofs)

> *For the authors of proofs, FPL* SHOULD *allow a wide range of how many or how few details they have to provide to allow an automatical validation of the proof by the FPL interpreter. It should still be possible for the less experienced mathematicians to get more details of the original proof by asking the FPL interpreter (or similar computer-aided tools) why a verified logical step is correct.*

5.8 Foundations of PBM

5.8.1 Objective

In this subsection, we will investigate what the foundations of PBM are and if the FPL specification should take any foundational system for granted or rather allow to formulate foundational systems inside FPL.

[118] [82], 17ff.
[119] compare 5.6

5.8.2 What are mathematical foundations?

Every theory starts with the abstraction process from the material content of the real-world objects it deals with.[120] Abstraction helps to get rid of special cases. Mathematical theories use abstraction to get rid of special cases of real-world objects. At a higher level (i.e., abstractions of abstractions), they create logical systems in which we do not investigate the objects of the real world but rather the abstract terms and their properties. If we continue the abstraction process even further, we will develop foundational mathematical theories. We call a mathematical theory *foundational* if we re-use it to define the syntax and semantics of related theories. On the other hand, the foundational theory defines *its* syntax and semantics. In this sense, a foundational theory is self-contained and is not a special case of a more general theory.

5.8.3 Historical Drift of Foundations in PBM

We present the historical evolution of mathematical foundations shortly. A more detailed analysis can be found, for instance, in [13], [102], [14], [53], [54], or any other treatise of the history of mathematics.

1. **Everything is a number**: In Ancient Greece, Pythagoreans believed that the *"everything is a number"*. With *"number"* they meant whole positive numbers and their ratios, i.e., what we would nowadays call the *rational numbers*.[121] They discovered that the sounds of chords were harmonic if their lengths behaved to each other like the ratios of consecutive natural numbers $1:2$ (octave), $2:3$ (quint), $3:4$ (fourth), or in general like $n:(n+1)$. Also, Pythagoreans discovered *perfect numbers*. These are natural numbers whose sum of divisors equals themselves: like $6=1+2+3$, $28=1+2+4+7+14$.[122]

2. **Crisis** The discovery of *incommensurable* lengths shook the Pythagorean foundations. Incommensurable lengths are what we would nowadays call *irrational numbers*. These lengths correspond to geometrical segments with no common dividing measure (like the length of the side of a square and the length of its diagonal). Pythagoreans could maintain the principle *"everything is a number"* if there were incommensurable numbers.[123], unless they extended their notion of a *"number"* to irrational numbers. However, this was technically not feasible in ancient mathematics[124].

3. **Everything is geometry**: The Pythagoreans chose another way out of the crisis: They revised their foundations: From now on, not *"everything was a number"*, but everything was geometrical. The Ancient Greeks managed to provide a geometrical

[120] [102] p. 96-97
[121] [106]
[122] [102] pp. 152-153
[123] [102] pp. 152-160
[124] [102] p. 155

interpretation of every kind of number known to them. They started with the whole positive numbers, up to their ratios and the geometrical constructions of irrational square roots, and including the geometrical equivalent of the *infinitesimally* small measures of areas and volumes of planar and spatial geometrical figures.[125]. Euclid laid out these foundations in his *"Elements"*. The success of providing new foundations of mathematics - changing from a purely arithmetical toward a purely geometrical view - was now more sustainable and lasted for centuries until the 18th century.[126] The scientific problems of this time stemmed mostly from solving practical construction problems and mechanics. They were solvable using smooth, *analytical functions* expressible as a power series (i.e., being a sum of terms like $f(x) = Ax^{\alpha} + Bx^{\beta} + Cx^{\gamma} + \cdots$). Therefore, mathematicians of the 18th century did not have any practical reason to believe that some problems or functions could not be written and treated like this. Thus, they assumed associativity and commutativity for infinite series like they did for finite sums[127]. Whenever mathematicians proved a result in one branch, mathematicians felt free to use this result in another branch without questioning possible boundaries of its validity. In [102][128], many examples of this attitude that was characteristic for this epoch, even among the best contemporary mathematicians, among others Daniel Bernoulli, Leibniz, and Euler. Does a convergent sequence mean that it comes arbitrarily close to a limit, or does it mean that it reaches it? Questions like this did not play any role.

4. **Crisis** The careless application of arithmetics and geometry to all other new branches of mathematics caused in the 17th and 18th century a new foundational crisis of mathemematics, because more and more paradoxes and mathematical curiosities occured that could not be explained by mathematicians, some example of which [129] we give here:
   - Mathematicians had doubts about the truth of the equation $\frac{+1}{-1} = \frac{-1}{+1}$ since they questioned how it was possible to get the same result if one divided a smaller measure by a greater one and vice versa.
   - How it is possible that $(1-1)+(1-1)+\cdots = 0$ but $1-(1-1)-(1-1)-\cdots = 1$?
   - *"Paradoxes"* connected to the discovery of *improper integrals*, for which previously unchallenged rules like the equation
   $$\int_a^b (f(x) - g(x))dx = \int_a^b f(x)dx - \int_a^b g(x)dx$$
   seemed not to work.
   - Using arithmetical sign rules for the division, J. Bernoulli deduced for the *infinitesimal* magnitude dx (used for integration and derivative purposes) that

[125] [102] p. 156
[126] [52] pp. 11-25
[127] [102] pp. 182-185
[128] pp. 161-166
[129] compare [102] pp. 169-201

since for $x > 0$ we have $d(-x)/-x = dx/x$, it must hold also for the integrals $\ln(-x) = \ln(x)$. Leibniz countered using another argumentation that a negative real number $-x$ has *"infinitely many logarithm values"*, all of which are complex numbers[130].

- Euler confirmed Leibniz's opinion[131], but was against his another conclusion: Leibniz claimed that a derivative formula valid for $\ln(x)$ does not have to be valid for $\ln(-x)$. For Euler, this would shake the *"foundations of calculus"* which implied, among others, that a *"correct application of laws and operations does not depend on the nature of objects"*.[132]
- Euler's view on the *"foundations of calculus"* were related to the contemporary view of a what a function is: A function in the 18th century was understood as a single analytical expression[133]. So if the logarithm $\ln(x)$ was a function, a distinction of cases like

$$\ln(x) := \begin{cases} \ln(x) & \text{for } x > 0, \\ \ln(|x|) + i\pi & \text{for } x < 0, \\ \text{undefined} & \text{for } x = 0 \end{cases}$$

  had simply to be ruled out. Thus, the possibility of the existence of functions that today we would call *incontinuous* was simply not considered.
- Because only smooth and continuous functions were allowed, and because only geometrical proofs were used, it was broadly believed that if a function is positive for all values $x < S$ and negative for all values $x > S$ then it must zero at $x = S$ (i.e. geometrically cut the horizontal axis at this point.)[134].
- In connection with heat conduction, incontinuous functions were discovered, so that the concept of a (always smoth) function did not work any more.
- The Euclidean notion of a *"number"*, in which a number was a total of units, was still present in the 18th century. However, mathematicians extended it to everything that can be measured and compared to a measure taken for a unit. Because this did not cover complex numbers known in the 18th century, scholars broadly believed that complex numbers had no real-world applications[135]. They were therefore perceived as curiosities and called *"unreal"* or *"imaginary"*.

5. **New ways out of the crisis** These and many other curiosities and seemingly paradoxes disappeared between the 18th and 19th centuries. Laws whose applicability had been taken for granted for all kinds of objects became more and more applicable only for a limited number of objects. New criteria of applicability were

[130] [102], p. 163
[131] Today we know that all three mathematicians were wrong since $\ln(-x) = \ln(x) + i\pi$ for $x > 0$.
[132] [102] p. 163
[133] [102] p. 164
[134] [102], p. 164
[135] [102] p. 170

discovered and sharpened the existing mathematical definitions. There were many examples of revolutionary discoveries in this time that impacted and widened the content, interpretation, and semantics of basic mathematical notions. We provide only some of them[136]:

- Bolzano's provided a first modern definition of what *continuous functions* are.
- Weierstraß found that there exist continuous functions, which have no derivative. His discovery helped to sharpen the distinction between *continuity* and *differentiability*.
- Fourier proved that his trigonometrical series (consisting of continuous terms) were able to converge against incontinuous functions.
- Cauchy was the first to recognize that it was necessary to elaborate *existence criteria* ensuring that limits of *indefinite integrals* can be calculated. He also revised the notion of a *limit*. In particular, Cauchy developed a concept we call today a *Cauchy sequence* and proved that a sequence of values $x_1, x_n, \ldots$ is convergent if and only if for every $\epsilon > 0$ there is an index N such that the distance $|x_m - x_n| < \epsilon$ for all indexes $n, m > N$. This criterion was essential because, unlike earlier concepts of convergence, it was now independent of how the sequence approached its limit: Earlier, only monotonic, smooth changes of the sequence members were allowed, now also oscillating or nonmonotonic ones.
- Riemann discovered criteria for what we call today *absolutely convergent series*. This discovery showed, for instance, that the commutativity law of addition was indeed limited. It was only applicable if a series was absolutely convergent. Otherwise, changing the order of addition could lead to different values of the resulting series. That explained some of the previous paradoxes.
- Mathematicians discovered that complex numbers could not be ordered using relations like $<$ or $\leq$ like it was possible for real numbers.
- The discovery of *quaternions* revealed that there are a number of types for which arithmetical laws like the commutativity of multiplication believed to be valid for all kind of numbers were not applicable.
- N. Lobachevsky (1772-1856) and J. Bolyai (1802-1860) discovered non-Euclidean geometries. It showed that Euclidean geometry that had been regarded so far as an absolute, only valid geometry, is not the only one that is possible.
- In his attempt to solve algebraic equations of n-th degree, E. Galois (1811-1832) create a new theory, in which the notion of a *group* played a central role. It turned out groups occur in many mathematical disciplines. It laid the foundations for what we today call *group theory*.
- Cauchy introduced the new concepts of uniform continuity and uniform convergence of sequences of functions that substantially helped understand the

[136] compare [102] pp. 199, 202-211

limits of applicability of arithmetical laws when calculating improper integrals, thus avoiding supposed paradoxes.

6. **The dispute about infinity**. Originally, the concept of infinity was invented in astronomy and developed independently by different ancient peoples, particularly Egypt, Babylon, China, India, Chaldea, and Greece. For Anaxagoras (ca. 500 to 428 BC)[137], only the *potential infinity* existed. Ancient Greeks believed that the *"whole is greater than its part"*.[138] Zeno of Elea (ca. 495-430 BC) *"proved"* that the *actual infinity* could not exist since it would cause contradictions of time, movement, and space. He provided 45 antinomies (contradictions) out of which only nine preserved to nowadays, the famous three of which are the *"Achilles and the tortoise"*, the *"dichotomy"*, and the *"being of many"*[139]. In 19th century, Anaxogoras' and Zeno's views about infinity were reborn in the movement of *intuitionism*. Intuitionists tried to explicitly *construct* all mathematical objects. They rejected the *principle of the excluded middle* $\phi \vee \neg\phi$ and its equivalent forms, e.g. the *double negation* $\neg\neg\phi = \phi$. This limitation led to a significant *"revision of mathematical knowledge"* and a *"fairly complicated theory"*[140] as compared to a non-intuitionistic one. Since the individual construction process is always finite, intuitionists only accept *potential infinity*. For instance, we can construct a new mathematical object (like the successor $n+1$ of a given natural number n) no matter how big it is. Famous supporters of this view were Carl Friedrich Gauß (1777 - 1855), later Luitzen Brouwer (1881 - 1966), and Leopold Kronecker (1823 - 1891), who supported in with dogged determination, and who is also famous for his sentence *"God made the natural numbers, all else is the work of man."*[141]. Other famous supporters were Cauchy and Leibniz[142].

7. **Everything is a set (first attempt)** The *actual infinity* occurs when mathematicians accept the possibility of *collecting* all mathematical objects that can be potentially constructed (e.g. natural numbers) and forming from this collection a separate new mathematical entity (e.g., $\mathbb{N}$ as the *"set of all natural numbers"*)[143]. This view was propagated by a student of Kronecker: Gregor Cantor (1845 - 1918). In his *set theory*, he discovered many pioneering new concepts (including *uncountable* sets, *ordinal numbers*, *cardinal numbers*). Cantor also formulated famous conjectures like the *continuum hypothesis* that have inspired mathematicians to develop new methods.[144] He could prove that the *cardinality* (number of elements) of any set X (even if X is infinite) is always less than the cardinality of its *power set*

[137] [102] pp. 156-157
[138] [43] Common Notion 1.5; today, with set-theoretic foundations of mathematics we will discuss below, we consider this as true only for finite sets.
[139] [102] pp. 157-158
[140] [78] 2.2 Intuitionism
[141] [20] pp. 33-34
[142] [102] p. 239
[143] [21] pp. 11-12
[144] [21] pp. 12-24

$\mathcal{P}(X)$. Starting with the *countable* infinity (for instance of the irrational numbers), he denoted by $\aleph_0$ he could prove that real numbers are uncountable, what he called the continuum $\aleph_1$. Due to Cantor, we can repeat the process of building the power sets of a given infinite set infinitely many times. It transfers a set with the cardinality $\aleph_i$ into a new set with the cardinality $\aleph_{i+1}$.[145] Cantor's set theory was very successful at the end of the 19th and the beginning of the 20th century, because it *"delivered a unified method to provide a foundation for almost all mathematical theories."*[146]. In the meantime, Dedekind and Peano formalized arithmetics.

8. **Everything is logic - a dead end**. *Logicism* is the idea of reducing mathematics to logic. The idea went back to Leibniz and was further developed in the 19th century, in which Gottlob Frege (1848 - 1925) spent his lifetime in formalizing logic in his *"Foundations of Arithmetic"*, published 1884[147]. Frege used Cantor's notion of a set and postulated some axioms, in particular the so-called *axiom of comprehension*. It states that we can always construct some set y if its elements $x \in y$ fulfill some *arbitrary* logical formula $\phi(x)$.

9. **Russell's paradox and a new crisis**. Russell discovered (1902) that Frege's *axiom of comprehension* was too strong and involved a contradiction[148]. Russell showed that $\phi(x)$ could also describe a property like $(x \notin x)$ which led to the contradiction $(y \in y) \Leftrightarrow (y \notin y)$ that became later known as *Russell's paradox*. This discovery shocked not only Frege because it seemed to destroy his life work but also all mathematicians trying to create logical foundations of mathematics[149].

10. **Everything is a typed class**. In *"Principia Mathematica"*, Russell!Principia Mathematica and Whitehead (1861-1947) proposed their own reduction of mathematics to logic that circumvented the discovered paradox by the so-called *type theory*. In the type theory, *"only properties of mathematical objects that have already been shown to exist, determine classes. Predicates that implicitly refer to the class that they were to determine if such a class existed do not determine a class."*[150]. The type theory did not stand the test of time because it restricted the concept of Cantor's sets too much so that even harmless sets cannot be easily constructed[151]. Moreover, it implied that *"there exists an infinite collection of ground objects [which could] hardly be regarded as a logical principle"*[152]. Logicism was a dead end due to its complicated and cumbersome language. Frege used his own complicated notation that never prevailed. Russell and used a more modern notation, but in their monumental work *"Principia Mathematica"*, published 1910, several hundred pages proceed the proof of $1 + 1 = 2$.

[145] [21] pp. 18-20
[146] [102] p. 222
[147] [78] 2.1 Logicism
[148] [22] p. 37
[149] [22] pp. 36-41
[150] [78] 2.1 Logicism
[151] [22] p. 41
[152] [78] 2.1 Logicism

11. **20th century: Everything is a set - a new attempt**. Ernst Zermelo took 1908 another approach to resolve Russell's paradox. He replaced some of the assumptions of Frege and Cantor with more cautious versions of axioms, which did not lead to paradoxes.[153]. In particular he preplaced Frege's *axiom of comrehension* by the *axiom of separation* that limits the formula $\phi(x)$ to some elements of an already existing set u, from which we separate elements that *in addition* to being elements of u fulfill the formula $\phi(x)$. This way, we circumvent Russell's paradox because the separated set y becomes simply a subset of the set u we already have shown to exist. His axioms (Z) were then complemented by one additional axiom by Abraham Fraenkel, leading to the so-called ZF axioms.

12. **Doubts about Zermelo's Axiom of Choice** One axiom of Zermelo - the *axiom of choice*, is sometimes excluded from the remaining ZF axioms. While the original notation ZF denoted all axioms, it currently denotes the original axioms without excluding the axiom of choice. ZFC denotes the complete set of axioms. Some mathematicians refuse the correctness of the axiom of choice. The reasons for this are, for instance:
    - The axiom of choice implies the existence of actual infinity, still rejected by mathematicians who are intuitionists.
    - It implies that every set can be *well-ordered* i.e., every non-empty subset of such a set contains a minimal element. This implication is not so obvious. For instance, we cannot write down the minimal element of the open interval $(0, 1)$, even though the axiom of choice implies it exists.
    - The axiom of choice allows to define sets that are not definable in the original sense of Cantor[154].
    - The axiom of choice allows decomposing measurable sets into non-measurable components, causing (seemingly) paradoxical results: Stefan Banach (1892 - 1945) and Alfred Tarski (1901 - 1983) proved using this axiom that it was possible to decompose a sphere into a *finite* number of non-overlapping parts and put them again together to create a sphere with double the radius of the original sphere; F. Hausdorff managed to decomposed a sphere into three parts such that each of them covered the sum of the other two (except only countably many exception points).[155]

13. **Structuralism and other doubts** Due to the philosopher Paul Benacerraf's article *"What Numbers Could Not Be"* published in 1965, the challenge to the set-theoretic foundations[156] is that there are *"infinitely many ways of identifying the*

[153] [102] p. 244
[154] [102] pp. 246-247 provides an example of such a set
[155] [102] p. 248
[156] [78] 4.1

natural numbers with pure sets" two of which are

I :			II :		
0	$=$	$\emptyset$	0	$=$	$\emptyset$
1	$=$	$\{\emptyset\}$	1	$=$	$\{\emptyset\}$
2	$=$	$\{\{\emptyset\}\}$	2	$=$	$\{\emptyset, \{\emptyset\}\}$
3	$=$	$\{\{\{\emptyset\}\}\}$	3	$=$	$\{\emptyset, \{\emptyset, \{\emptyset\}\}\}$
$\vdots$			$\vdots$		

From the set-theoretical point of view, they are *structurally incompatible*. Benacerraf proposed that natural numbers are not sets at all. Accepting his proposal would solve the problem since there are no reasons why one account is superior to the other, and both accounts cannot be correct at once. Benacerraf concludes a similar argument for all other number systems, like the integers, the rational numbers, the real numbers, etc. Also, in [9] argues that Benacerraf's challenge causes many ambiguities when interpreting the semantics of mathematical concepts. One of other examples instanced in [9] is that mathematicians often *identify* the set of natural numbers $\mathbb{N}$ with a *proper subset* of the set of integers $\mathbb{Z}$ ($\mathbb{N} \subset \mathbb{Z}$)[157], while other mathematicians define natural numbers as the set of *equivalence classes* formed from *ordered pairs* of natural numbers having the same difference[158]. Thus, mathematicians believing that everything is a set propose semantically incompatible interpretations for many mathematical concepts.

14. **Other Modern Foundational PBM Theories** Besides ZFC, there are concurrent foundational theories, like the Neumann-Bernays-Gödel (NBG) set theory (NBG)[159], involving the notion of a *class*, or the *category theory* developed in the 20th century, which also has the potential to play a fundamental role to mathematics.[160].

5.8.4 Conclusions on Foundations

Although in contemporary literature, we can find a broadly accepted view that we can found *"all of PBM"* on ZFC[161], the doubts about the axiom of choice or Benacerraf's problem show us that there are no foundations of PBM that could ever be considered as the ultimate and correct ones.

Mathematicians have considered *many* theories as foundational throughout centuries. The question arises if FPL should prescribe a syntax (respectively semantics) of FPL that takes any notation (respective ideas) from existing foundational systems of mathematics,

[157] Such an identification can be found, for instance in [18] p. 18
[158] such a construction can be found, for instance in [48] p. 72
[159] [21] p. 44
[160] see [33] p. 3, [10] pp. 165-167, or [3], p. 2
[161] examples can be found both, in purely mathematical sources, e.g., [22] 145ff., [7] 5ff., [32] 21ff., but also in linguistic and philosophical ones, e.g., [9] 175ff.

like the ZFC set theory, or the Peano axioms for granted. The above analysis gives us a clear answer is to this question is *no*. First of all, the foundations of mathematics underwent significant shifts and challenges in the past. There is no reason to believe that today's foundational theories like ZFC, NBG, and the category are the final answer. What we learn from the history of foundations in mathematics is that these theories are only current snapshots of possible foundations of PBM. Moreover, the history of mathematics shows that the foundations of mathematics have always had their supporters and opponents. Therefore, building any foundations of mathematics into the language of FPL (for instance, inbuilt set-theoretic concepts like the $\in$ operator) and its notation would put the language FPL at risk, that it would become outdated in the future. We formulate, therefore, a new important requirement:

Requirement 21 (Independence of Foundations in FPL)

> *The FPL specification* MUST NOT *presume any foundations of PBM. No axiomatic system* MAY *be built into the syntax of FPL.*

Moreover, the above analysis leads us to the following high-level design decision of FPL:

Requirement 22 (Optional Intuitionism in FPL)

> *The FPL specification* SHOULD *not put itself constraints on whether the principle of excluded middle is valid, or on whether the actual infinity exists, or whether proofs by contradiction are allowed. How mathematics is done and interpreted should be up to the individual user (human or computer program). However, FPL* SHOULD *allow technical means* recognizing *if non-intuitionistic logical arguments or mathematical concepts are being used in an FPL theory.*

5.9 Semantics of PBM and Metaphysics

5.9.1 Objective

We learned a lot about the separability syntax and semantics in formal languages in previous subsections. Therefore, we have to address in FPL the problem of how we are going to encode the semantics of mathematical objects in the syntax of FPL. Before we can do that, we have to understand better what *semantics* is and which possible problems can occur if we deal with this concept too carelessly in the design of FPL. We can only interpret objects based on what we know or have observed about them so far. In psychology, semantics raises at least two metaphysical issues: the ontological *"issue or real existence"* and the epistemological issue of *"how we can ever know anything"*[162]. The ontological problem of mathematics is how real the mathematical objects are we deal with in PBM. The epistemological issue about mathematics lies in the AM. As a deductive method, it only *preserves* knowledge, but it does not *create* new knowledge. Ironically, to interpret a *formal system* based on axioms, we need knowledge about the domain of discourse

[162] [44] Part 5: *"Three Enduring 'Isms' – Empiricism, Rationalism, Materialism"*

to interpret the axioms as true statements and, consequently, the formal system as a consistent one. However, while proving more and more theorems inside using Hilbert's calculus in a formal system, we do not increase our knowledge because it is already there - it is required a priori to interpret the axioms, based on which we prove the theorems. So where does that knowledge come from?

In this subsection, we can only shortly explore answers to these big questions of mathematics which have occupied the minds of philosophers, mathematicians, and scientists throughout centuries. Our goal is to identify all problems that we have to consider when designing how we encode semantics in the syntax of FPL.

5.9.2 Are Mathematical Objects Real?

Most philosophers postulate that reality is separated from the human mind because we can only observe reality, creating an individual interpretation. The empirical observation of the world around us creates new psychological problems caused by the limits of perception and sensation[163]. We will not address these psychological problems and simplify our investigation by trying to answer the following questions:

- Are mathematical objects real-world objects?
- If not, what ensures the existence of objects in mathematics?

There are many different approaches to answer these questions. We present only some of them:

1. In **Platonism**, PBM is about *abstract entities*[164] that exist independently from subjects (e.g., humans or computers) but are not necessarily the same as real-world objects. Due to Kurt Gödel (1906-1978), who was a platonist, there are similarities and differences between abstract and physical worlds, shown in the **Table** 5.

Mathematical Objects	Physical Objects
Parallels	
Not constructed by humans	
Objective	
Postulated to achieve satisfactory theory of experience	
Not fool-proof, can be corrected	Our perception is fallible and can be corrected
Differences	
Do not exist in space in time	Exist in space and time

Table 5: Abstract and physical worlds due to Gödel

[163] e.g., the laws of Weber and Fechner, [44], *"Sensation and Perception"*
[164] see [78] 3. Platonism

2. Tegmark takes can find an even more radical position in [55]. For him, the Platonic abstract world and the real world are, in fact, the same entities. He postulates its existence in what he calls the *External Reality Hypothesis* (ERH), which is *"accepted by most but not all physicists"*. In Tegmark words, *"It is crucial not to conflate the language of mathematics (which we invent) with the structure of mathematics (which we discover)"*[165].

3. Using abstract mathematics to predict real-world phenomena goes back to Galileo Galilei (1564 - 1642). There are multiple examples of physical phenomena, which mathematics predicts, including planets or antimatter. For more details and discussion, refer, for instance, to [13], [55], [45], [97], or [102].

4. We have discussed already in 5.8 the dispute between formalists and intuitionists whether or not the *actual infinity* existed or not. Today, most mathematicians believe that it does, although nobody can observe actual infinity in the real world. The number of atoms in the observable universe is estimated between 10^{78} to 10^{82}[166], a huge, but still finite number. Mathematicians base their belief on the results achieved by *assuming* that such an infinity existed. The following list contains examples of such results:

   - Already Ancient Greeks proved the existence of irrational numbers like $\sqrt{2}$, whose discovery (including the numbers $\sqrt{3}$ to $\sqrt{17}$) is attributed to Theodorus of Cyrene (5th century BC).[167] Irrational numbers are magnitudes that are *incommensurable*, i.e. (for instance $\sqrt{2}$), no *finite* (or potentially *infinite*) decimal precision is enough to represent such numbers.
   - Cantor discovered in the 19th century that there are only countably many irrational numbers being solutions of algebraic equations with rational coefficients (like $\sqrt{2}$ is the solution of the equation $x^2 - 2 = 0$). Since he also proved that real numbers are uncountably many, he could not find any examples of real numbers that would fill the gap.
   - This succeeded in finding the first so-called transcendent numbers, i.e., real numbers being not solutions of any algebraic equation of rational coefficients. A prominent example of such a number is the number $\pi \approx 3.1415\ldots$. Its transcendence was proved 1882 by Ferdinand von Lindemann (1852 - 1939). By proving the transcendence of π, von Lindemann solved a two thousand years old geometrical problem of finding a ruler-and-compass construction of a square with the same area as a given circle. The transcendence of π showed that such a construction does not exist.[168]

   This historical development shows that the assertion of the existence of *actual infinity*, which we *cannot observe* in the physical world, implies the existence of

[165] [55] pp. 243-271
[166] [98]
[167] [13] p. 77
[168] [22] p. 12

other abstract mathematical objects, like the transcendental numbers. We need some of them, like the number π, to describe many real-world phenomena we *can observe*, like oscillations, waves, or relative frequencies of statistical events.

5. In some mathematical theorems, mathematicians want to *"prove the existence"* of some mathematical objects. Most frequently, such objects are the solutions to numerical problems. The existence of such solutions justifies searching for them and if a solution to a given problem does not exist, then developing algorithms or methods to find it doesn't make sense. Even if mathematicians can prove the existence of a solution and have found such a solution, engineers or natural scientists will consider the solution as an existing one if they can apply it to a concrete real-world problem. Mathematicians are satisfied only with the abstract proof of existence, sometimes besides proving that no better solution exists. The concrete implementation of the solution in the real world is of secondary interest for mathematicians, if not of no interest.

6. Notwithstanding the success of mathematics in predicting the physical world, in *"pure mathematics"*, the ontological issue existence reduces to a formal one. On a pure syntactical level, the *quantor* $\exists x(p(x))$ is used to state that *"there exists at least one x fulfilling $p(x)$"*[169], as we saw in 5.4. Recall that quantors like $\exists$ are used to *bound* variables used in a predicate. We can observe examples of using the $\exists$ quantor to solve the issue of existence in mathematics in some ZFC axioms, especially the axioms asserting the existence of a set, a power set, and an infinite set.

7. Cantor, who supported the existence of actual infinity in his set theory, distinguished himself between two kinds of the reality of mathematical objects. First, ideas and concepts have an *immanent reality* if they only exist in our minds. But those ideas and concepts which reflect processes or objects of the outer world have a *transient reality*. For Cantor, mathematics was a *"free"* science because it had only to consider ideas with immanent reality. However, Cantor still restricted the freedom of ideas and concepts in mathematics: The (inherent) existence of only those ideas should be allowed, which were consistent (i.e., free of contradictions) and could be (at least somehow) related to already established mathematics.[170] Many contemporary mathematicians follow this *"free"* view on existence: Many of them believe in the existence of ideas and concepts, not because they reflect real-world objects, but because they are logically consistent. Only contradictory objects do not exist.

According to the above analysis, mathematicians seem to neglect whether mathematical objects exist in the real world. Instead, they care about the consistency of their theories and deny the existence of mathematical objects that would lead to a contradiction. On the other hand, they seem to accept every abstract entity inside a given axiomatic system

[169] [22] p. 133
[170] [102] p. 229

that does not cause a contradiction. Therefore, we are tempted to conclude that the existence of mathematical objects is equivalent to the lack of contradictions they might cause. However, things are not as simple as this. There are at least some exceptions to this rule, and we want to explore these exceptions:

1. In some cases, mathematicians do not shy away from *defining* mathematical objects *using* contradictions. For instance, Ebbinghaus defines the *empty set* $\emptyset$ by the expression $\emptyset := \{x \mid x \neq x\}$ i.e. as the set of all elements that do not equal themselves.[171] In the context of a mathematical proof, mathematicians would treat the expression $x \neq x$ like a contradiction, i.e., an expression that is *false* under any interpretation. On the other hand, mathematicians would dispute the existence of a set like $Z := \{x \mid x \notin x\}$. For if Z is a set, then by definition $x \in x \Leftrightarrow x \notin x$, which is a contradiction, known as Russell's paradox[172].

2. In 5.8 we noted that the intuitionists among mathematicians reject the *double-negation*, i.e. $\neg\neg\phi$ for any expression ϕ. Because $\neg(x \neq x) \Rightarrow x = x$ involves double negation, the practical question arises how (if at all) an intuitionist would be able or willing to negate an expression like $x \neq x$.

3. Despite its name, Euclid's *parallel postulate* omits the notion of a parallel line. If Euclid used that notion in the postulate, he would need to explain what it means to extend straight lines *infinitely* (which he didn't). According to [56], the parallel postulate states: *"If a straight line l falling on two straight lines n and m makes angles α and β less than $180°$, then those straight lines meet on the side, on which α and β occur [...] Euclid's axiom about non-parallel lines implies that parallel lines exist"*[173]. Only by *assuming the opposite* (contradicting!) of the postulate, the existence of parallel lines in the Euclidean Geometry follows. For if $\alpha + \beta \geq 180°$ then we have either the case $\alpha + \beta > 180°$, which means that the two straight lines n and m make some angles α' and β' less than $180°$ on the opposite site of l (being just a reformulation of the axiom for the opposite side) or that $\alpha + \beta = 180°$, which means that the two straight lines n and m *do not meet*.

Note the subtle difference in the definitions of $\emptyset$ and Z above: $\emptyset$ *"exists"* in the mathematical sense and its definition *uses* a contradiction as a defining property of its elements, while Z *"does not exist"* because the defining property of its elements *leads* to a contradiction. Is it possible to create a computer that would accept the definition $\emptyset$ but reject the definition of Z? Are mathematicians rigorous enough in the way they deal with contradictions concerning questions of existence in mathematics? How could a computer accept this kind of formal language? Fortunately, there are ways of avoiding contradictions in definitions, and these ways promise a better way for a formulation in FPL that is both human- and computer-friendly. For instance:

[171] [7] p. 33
[172] [32] p. 22
[173] [56] p. 3

1. To define $\emptyset$ without using a contradiction, we could define $\emptyset$ as a set fulfilling the axiom that no other set is its element. Formalizing this axiom would require the property that $\emptyset$ is a set x such that for all sets $z \neq x$ we have $z \notin x$. Note that we are using a quantor *"for all"* and also excluding the case $z = x$ to avoid running into Russell's paradox.

2. Hilbert chose a better formulation of the parallel postulate than Euclid. His formulation neither creates the problem of negating an axiom to derive parallel lines nor requires the notion of extending a straight line infinitely. All that is needed is counting unique straight lines through a point. Hilbert's formulation of the axiom is[174] this:

   > *"In a plane, there can be drawn through any point* A*, lying outside of a straight line* a*, one and only one straight line* l *which does not intersect the line* a*. This straight line is called the parallel to* a *through the given point* A*."*

   Note that this formulation is not equivalent to Euclid's formulation. If we negate this statement, we allow multiple lines through A that do not intersect a, leading to non-Euclidean geometries discovered independently by Lobachevsky and Bolyai.

5.9.3 How do we gain new knowledge in mathematics?

When characterising *inference*, i.e. an abstract method to describe human thinking, [4] defines it as a relation R such that $(K_1, K_2) \in R$ if we generate new knowledge K_2 given a knowledge K_1[175]. In 1931, C. S. Peirce proposed tree types of inference[176]:

- **Deduction**: If it is raining, we could *conclude* that the streets get wet, based on the general rule that dry streets never occur when it is not raining. We derive our knowledge from a general case to a special case.

- **Induction**: If we observe a wet street, we could *conclude* that it is raining because we know from observation that streets become wet when it is raining. We derive our knowledge from many special cases.

- **Abduction**: If we observe a wet street, we could also *make a hypothesis* that there is a sprinkler installation nearby. We derive our knowledge from an active search for possible new explanations for our observations.

In 5.1, and later in 5.6, we saw that PBM is solely based on the deductive (axiomatic) method, we abbreviated with *"AM"*. According to [15], AM it is the only type of inference that preserves the truth[177]. Both, *induction* (in the terminology of Peirce)[178] and

[174] [77] p. 7
[175] [4] p. 20
[176] [34], we use examples similar to those given in [4] 23ff.
[177] [15] p. 194
[178] The term *"induction"* should not be confused with the mathematical method of proving *"by induction"*, which is a misleading terminology, because actually this is a *deductive* form of reasoning.[15] p. 195

abduction are not admitted in mathematical proofs, because they are types of so-called *"non-monotonic, or defeasible reasoning"*[179]. Theorems for which mathematicians find deductive proofs cannot be revoked or revised, even if additional knowledge is available, while the collection of true statements in a mathematical theory that can be proven deductively is *"monotonically increasing"*[180].

According to this analysis based on Peirce's model, if mathematicians only had AM as a method at their disposal, they could never increase their knowledge; they could only increase the number of true statements inside their axiomatic system(s). As we saw in 5.6, mathematical *"knowledge"* we can gain in the Hilbert calculus would consist of a collection of true theorems that we proved in a chain of *tautologies* derived from axioms, the truth of which we have postulated. Mathematics would never be a creative mental process, let alone increase our capability to predict real-world phenomena. We would reduce it to a mechanical finding of proofs for theorems we can derive from a given, static set of axioms. Only the number of proven conjectures could *"monotonically increase"*, but mathematicians could never formulate new conjectures. This is another argument that reconfirms our **Requirement** 21.

With this respect, only *abduction* can increase knowledge because it is the only form of human inference that actively formulates a *new* hypothesis about a possible reason for what we observe. In contrast, *induction* can help to verify this hypothesis[181]. The interaction between induction and abduction constitutes the scientific method, the *Novum Organum* described by Francis Bacon in 1620, and adopted in modern sciences, especially the natural sciences like physics, chemistry, and biology.[182]

Not surprisingly, working mathematicians apply the scientific method, as other scientists do. Only this method (and not AM) enables them to formulate new conjectures in mathematics and potentially increase their knowledge. However, there is a significant difference between mathematics and other sciences. Even if mathematicians (in an *abductive* process) formulate a hypothesis that turns out to be a perfect one because it has overwhelming evidence[183], they would never accept this evidence as proof for that conjecture because it is an *inductive* argument. Mathematicians still need some *deductive* argument that finally strengthens their faith that the conjecture is true.

The attempts to prove new conjectures lead to theories consisting of many auxiliary theorems helping to prove the original conjectures. Only when the whole theory is big

[179] [4] p. 26
[180] [4] p. 27
[181] [15] p. 195, referring to Peirce [34]
[182] [44] Part 4. *"The Emergence of Modern Science"*, Bacon establishes the triumph of method over authority (ex. the authority of Aristotle).
[183] As an example, consider the *Goldbach hypothesis*: *"Every natural number $n > 2$ can be written as a sum of two prime numbers."* Numerical evidence shows that not only is there such a sum, but also that the number of possible sums for a fixed n seems to grow as n grows.[22] p. 4

enough can mathematicians try to reduce the theory to axioms of an existing foundational system (like ZFC) or formulate possibly new axioms as a foundation of the emerging new mathematical theory. In contrast to how mathematicians finally present and publish their theories, they seldom create them *starting* from axiomatic systems. The actual chronology of creating a mathematical theory starts with conjectures and their proofs and ends with axioms to create its foundations. Only the endproduct appears as if it was created from scratch, starting with axioms.

Thus, mathematicians are not limited to AM but use abduction and induction, as other scientists do. But how about the latter? Do they use AM as a method, or is that method reserved for mathematicians? At least, physicists use AM, too. It brings us back to the ontological issue of existence and real-world relevance of mathematical objects. There are multiple examples of axioms in physics, starting from classical mechanics like in Newton's laws of motion[184], relativistic mechanics like in Einstein's (1879 - 1955) principles of *relativity*axiom!in theory of relativity or of the *constancy of light speed*[185] to axioms of *quantum mechanics*[186], and the respective theories follow from them deductively. Like mathematicians, physicists care about avoiding contradictions and do not prove their axioms but rather assume their truth. However, in contrast to mathematicians, physicists do not accept axioms just because they ensure consistent theories. This can be temporarily the case like for instance, the *string theory* is accepted as a *"leading candidate"* to *"describe space and its contents"*[187]. However, physicists will probably discard string theory in the long run if they cannot produce experimental evidence supporting it or even produce experimental evidence contradicting it. Thus, physicists address the issue of existence requiring both: the logical consistency of a theory and its ability to predict observations correctly. On the other hand, mathematicians address existence by solely requiring a theory to be consistent.

5.9.4 Conclusions

Although they base PBM on the deduction, mathematicians use abduction and induction in their creative process to generate new knowledge, and PBM is only the language to express this knowledge and give it semantics in the form of a consistent theory, free of contradictions.

Based on the above analysis, we can add this observation to our list of observations:

Observation 11 (Ontological Questions Are Out Of Scope in PBM)

> *While mathematics can describe and even predict many real-world phenomena, the issue of actual existence doesn't play an essential role in PBM. Mathematicians (unlike physicians) intentionally do not address the ontological question of the actual*

[184] [45] pp. 129, 135, 140
[185] [45] pp. 1242,1243
[186] compare [109]
[187] [55] p. 135

existence of their abstract objects. All that matters is the logical consistency of predicates involving bound and unbound variables using quantors like ∃ or ∀. Philosophical or physical questions are out of scope in PBM.

5.10 Axiomatic Method vs. Semantics in PBM

5.10.1 Objective

Transmitting the meaning of words from a sender to a receiver in any language is a complex phenomenon. It also holds for the language of PBM. Unsurprisingly, there are contradicting experts views on this topic:

- In 5.1 and 5.2, we observed incoherent views on the purpose and the necessity of axioms to express semantics. Although all sources claimed to *describe* the language of PBM, some of them recognized definitions as the sole means to *"fix the semantics and notation"* of mathematical objects.
- Some other sources (for instance [23] or [42]) attested that we could not express the semantics of some basic mathematical objects and that need axioms to *complement their meaning*.
- In 5.5.5, we learned that while formalists claim to be able to *precisely express* the PBM in predicate logic, they do so without definitions at all. Even worse: For formalists, a formalized PBM and its axioms consist of meaningless arrays of symbols, unless we have an external *interpretation* of these symbols, under which all formulas become either *true* or *false*. The *provability* of a theorem does not depend on its meaning or truth. Only the (meaningless) syntax of *axioms* and *inference rules* determine the provability of a theorem. In formalized PBM, a proved theorem can be both: false or true, depending on its interpretation.

These are three contradicting expert views regarding by which means semantics is encoded in PBM. In this subsection, we will analyze how exactly *semantics* is encoded in PBM and provide our own explanation. It will be one of the main results of this document. Some insights we gain in this subsection are novel. In particular, we will discover new limits to a possible FPL specification that we consider fundamental to every language, not only the language of mathematics.

5.10.2 The Infinite Regress Problem in PBM

The **Infinite Regress Problem** (IRP) is a philosophical problem of being fundamentally unable to explain something. According to [66], IRP can occur in logical arguments. In fact, it can occur in different domains, depending on the type of questions we are trying to answer. Logical IRP corresponds to the *"Why?"* question, but in principle, all *Five Ws*[188] (*"Who"*, *"What"*, *"When"*, *"Where"*, *"Why"*) can induce different types of IRP

[188] compare [67]

problems. In mathematics, we can observe IRP in two different domains: in *logical arguments* (i.e. mathematical theorems and their proofs, answering the question *"Why"*)[189] and in *mathematical definitions* (answering the question *"What"*) [190]. We will call these two kinds of infinite regress problems the Logical IRP (LIRP) and the Semantical IRP (SIRP).

LIRP and SIRP have only apparently nothing to do with each other. In PBM, they are strictly interconnected: We cannot address LIRP without addressing SIRP and vice versa. The reason for this is a characteristic of PBM we discussed in 5.5: If mathematicians define mathematical objects, they answer the *"What"* question. As soon as they relate these objects to each other, they produce either a *true* or a *false* statement. By proving this statement, they answer the *"Why"* question. Unfortunately, asking the *"What"* and *"Why"* questions ever and ever again, mathematicians finally face SIRP and LIRP and have to address both. The *axiomatic method* AM we learned about in 5.6.4 is exactly the method mathematicians intuitively use to address SIRP and LIRP at once. In [42], we find a mainstream view about how AM addresses SIRP and LIRP at once:

1. Addressing LIRP: Mathematicians *"base their different mathematical disciplines on **axioms**, i.e. on statements that are just **postulated to be 'true'** and which can then be referenced in all subsequent proofs."*[191],

2. Addressing SIRP: *"[...] the basic notions are not **defined explicitly**. Instead, they have to be **defined implicitly**, describing their relationships that are formulated in appropriate axiomatic systems."*[192]

Thus, modern AM addresses both, SIRP and LIRP in axioms. Definitions are omitted for basic notions because they are meaningless on their own. Instead, axioms give them meaning by relating them to each other in a logical statement. It seems that, at least at the level of basic mathematical notions, we do not need definitions anymore to answer the *"What"* question because axioms answer both: the *"Why"* and the *"What"* questions.

In the following text, we will contradict this mainstream view and show that axioms never fully answer the *"What"* question (i.e., address SIRP). Still, they successfully address LIRP. With this respect, SIRP is more fundamental than LIRP and this insight is novel.

5.10.3 References of Building Blocks in PBM

Before we continue our investigation, we first make the notion of *"source"* more precise. Building blocks in mathematics (e.g., definitions, axioms, theorems) always have identi-

[189] *"If we keep reducing more complicated theorems to simpler ones, we finally encounter theorems that cannot be further derived from other laws."* [42] Vol 1. p. 21 (own transl. from German)
[190] *"[When defining mathematical concepts,] it is inevitable that we finally come upon basic notions that cannot be defined [...]"*[42] Vol 1. p. 21 (own transl. from German)
[191] [42] Vol 1. p. 21 (own transl. from German), see also 5.6.4
[192] [42] Vol 1. p. 21 (own transl. from German)

fiers that help to distinguish them inside a given theory. We say that a building block y of a mathematical theory $\mathcal{T}$ **refers to** another building block x, denoted by $x \to y$ if and only the string expressing y contains the identifier of x. Note that if x is a theorem-like building block, or a conjecture, or an axiom, the reference $x \to y$ will be a logical one since we can assign x a truth value, which you can recall from 5.5. Such a text reference will be used in the context of a logical argument, for instance, in a mathematical proof. However, if x is a definition, $x \to y$ must be a semantical reference, because y references to a building block x that we cannot assign a truth value (compare 5.5.5).

Now, we build the **transitive hull** of all such text references, i.e. the set $H(\mathcal{T})$ of all pairs (x, y) such that x and y are building blocks of $\mathcal{T}$ and $x \to y$. In such a transitive hull, we can define the set of **source building blocks**:

$$S(\mathcal{T}) := \{x \in \mathcal{T} \mid x \neq z \textbf{ for all } (y, z) \in H(\mathcal{T})\}.$$

Thus, $S(\mathcal{T})$ contains exactly those building blocks of $\mathcal{T}$ that contain no further references. S might be empty. In this case, all building blocks inside the theory $\mathcal{T}$ either refer to some building blocks in some external mathematical theory, or the references inside $\mathcal{T}$ contain a circular reference. We could also say that there is a *vicious circle* inside $\mathcal{T}$, and mathematicians will usually try to avoid it. In the usual case, S will be non-empty for a real-case mathematical theory. The interesting question then is, which kind of building blocks of $\mathcal{T}$ will $S(\mathcal{T})$ contain when it is non-empty? Definitions? Axioms? Both? The answer is: It depends on the kind of *axiomatic method* mathematicians use. We will demonstrate this by studying the evolution AM underwent throughout centuries.

5.10.4 Intrinsic Semantics in AM Until the 19th Century

The AM is a very old method of philosophical reasoning, reaching back to Eudoxus of Cnidus (c. 390 - 337 BC), Aristotle (c. 384–322 BC), or even back to Plato (c. 438 - 348 BC)[193] before it even was used in mathematics. In mathematics, it first emerged in Euclid's epochal work *"Elements"*[194] 230 BC. Euclid of Alexandria compiled the mathematical knowledge of his time in 13 books. Among definitions, theorems[195], and their proofs, which we can find in almost every one of the 13 books, only the first book also contains five *"axioms"* and five so-called *"common notions"*. Axioms and common notions are the only theorems in the 13 books that remain unproven. They are just *a priori* assumed as true, and Euclid uses assumed axioms to derive every theorem deductively. This is how Euclid addresses LIRP in the *"Elements"*.

Now, let us have a look how Euclid addresses SIRP: The subtle characteristics of Euclid's AM is that he defines basic notions by the *"essence of their meaning"*[196]. For

[193] [20] p. 14
[194] [60]
[195] Euclid called all of his theorems either *"propositions"*, or *"lemmas"*, or *"porisms"* (today we would say corollaries)
[196] [20] pp. 18-19

instance, Book 1 starts with many definitions, the first three of which are[197]:

- **Def 1.01**: *"A **point** is that of which there is no part."*
- **Def 1.02**: *"And a **line** is a length without breadth."*
- **Def 1.03**: *"The **extremities** of lines are points."*

Note that in this example, Def 1.03 uses the notions *"line"* and *"point"*, referring to the definitions 1.01 and 1.02. However, Euclid uses in these two definitions the terms *"part"*, *"length"*, and *"breadth"* and does not defined them in the *"Elements"*. Euclid presumes that every reader of his work knows the semantics of these terms. Thus, Euclid defines the **basic notions** *"line"* and *"point"* *"by their essence"*[198]. Their semantics is subjective. Every reader of *"Elements"* has an own notion of *"part"*, *"length"*, and *"breadth"* and creates an own understanding of *what* a *"line"* or *"point"* are. This is how Euclid addresses SIRP: The final answer to the *"What"* question remains with the reader. We will call such subjective semantics of mathematical notions **intrinsic semantics** of basic mathematical notions.

This kind of AM introduced by Euclid in the *"Elements"* 2300 years ago remained *"virtually unchanged until the end of the 19th century"*[199]. The **Figure** 1 illustrates how SIRP and LIRP had been addressed in ancient AM. The building blocks of the *source* $S(\mathcal{T})$ of a theory like that described in Euclid's *"Elements"* consists of definitions of basic notions that are defined explicitely using *intrinsic semantics*.

We should note that the frameworks proposed in [9] and [71] to program computers in such a way that they could derive semantics of mathematical objects from mathematical texts requires precisely this kind of AM. These frameworks propose to use adaptivity[200], in which the computer finally arrives at the source of a theory consisting of definitions with intrinsic semantics (we will call such definitions **intrinsic definitions**. These frameworks make do without axioms because adaptivity, according to them, does not need axioms to derive semantics[201]. Unfortunately, these frameworks would fail in the modern form of AM introduced by Hilbert at the end of the 19th century. We will now look at what changed to AM since then and how mathematicians address SIRP and LIRP in modern PBM.

5.10.5 Relative Semantics in Hilbert's Modern Form of AM

David Hilbert (1862 - 1943) published in 1899 his new *"Foundations of Geometry"* [77], in which he chose a very different approach to encoding a *"meaning"* into the theory. Although he never used the term *infinite regress problem* in this context like we do, Hilbert was convinced that definitions based on the *intrinsic meaning* of things would cause

[197] see [60]
[198] [20] p. 18
[199] [20] p. 16
[200] compare 11 in 5.1
[201] compare [9] pp. 66-67, [71] p. 18

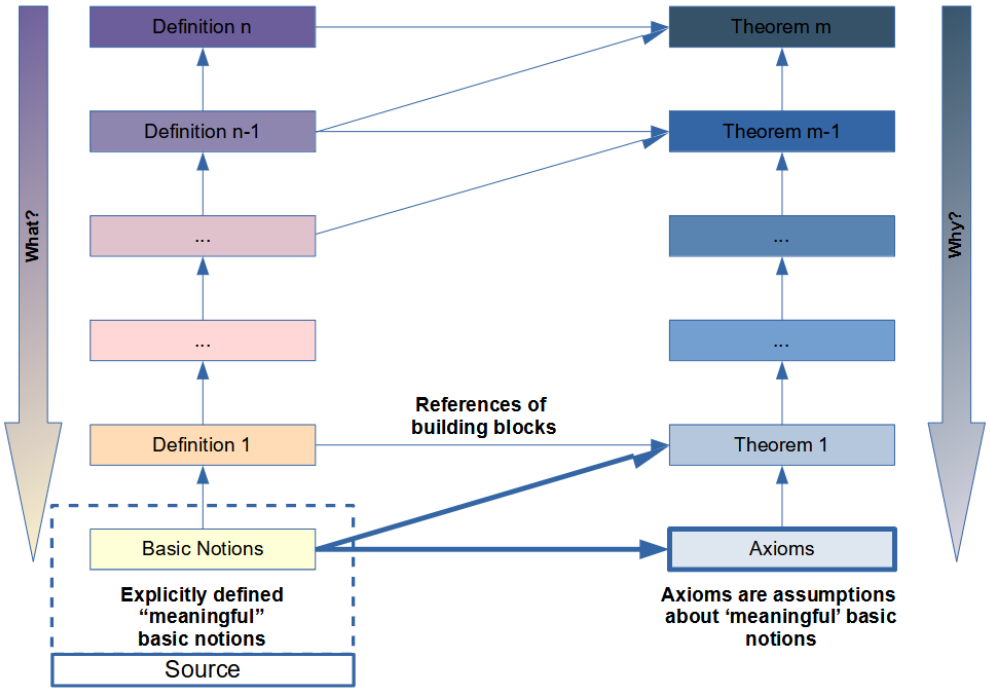


Figure 1: Addressing SIRP and LIRP in the AM from Euclid until the 19th century

semantical problems, in our vocabulary, they would inevitably lead to SIRP. Hilbert's approach to solving this problem was to avoid defining mathematical objects *"by their essence"* as Euclid did. Instead, he decided to define them by their relationships with each other. In Hilbert's *"Foundations of Geometry"*, we can find no separate definitions of a *"point"* or a *"line"*. In his own words[202]:

> *"Let us consider three distinct systems of things. The things composing the first system, we will call* ***points*** *and designate them by the letters* $A, B, C, \ldots$*; those of the second, we will call* ***straight lines*** *and designate them by the letters* $a, b, c, \ldots$*; and those of the third system, we will call* ***planes*** *and designate them by the Greek letters* $\alpha, \beta, \gamma, \ldots$*. [...] We think of these points, straight lines, and planes as having certain mutual relations, which we indicate utilizing such words as 'are situated,' 'between,' 'parallel,' 'congruent,' 'continuous,' etc. The complete and exact description of these relations follows because of the axioms of geometry. These axioms may be arranged in five groups. [...]"*

Hilbert's conscious renunciation of defining basic notions explicitly and giving them meaning by relating them with each other in axioms is a subtle but major change in the AM as compared to Euclid's AM. It's Hilbert's way to address SIRP: The final an-

[202] [77] p. 2

swer to the *"What"* question shifts from the domain of definitions into the domain of axioms. He replaces the definitions by axioms in the source $S(\mathcal{T})$ of a theory $\mathcal{T}$. Basic notions become meaningless. Only axioms give them meaning by relating them with each other. Now, axioms address both SIRP and LIRP since they answer the final *"What"* and the final *"Why"* questions. We will call such semantics of basic mathematical notions **relative semantics**. We illustrate Hilbert's AM in **Figure** 2.

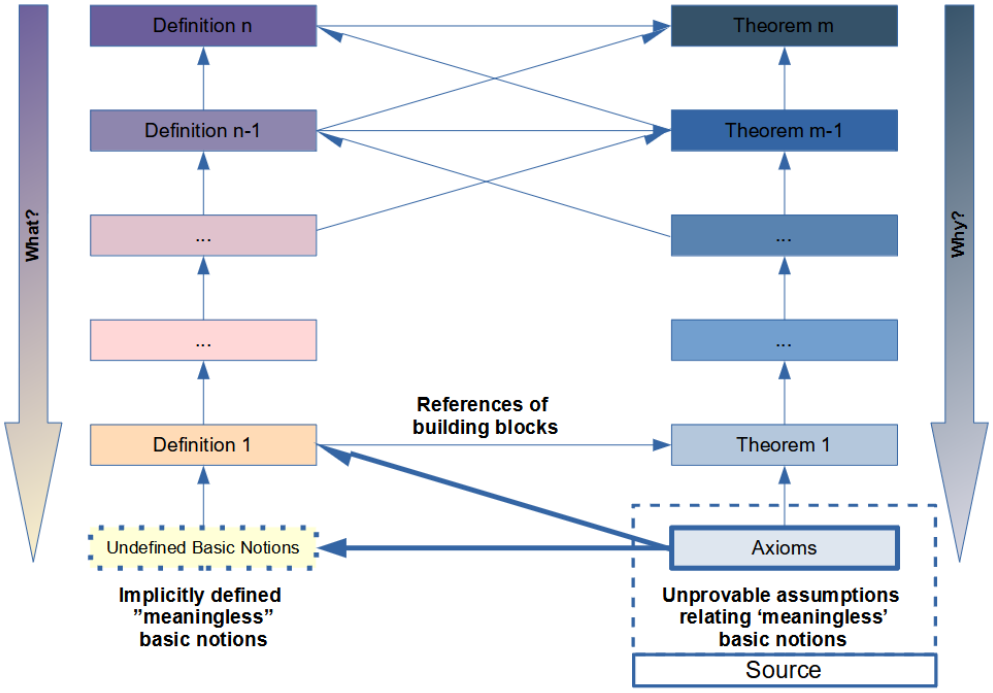


Figure 2: Addressing SIRP and LIRP in the modern AM since Hilbert

In a sense, relative semantics in Hilbert's AM is superior to intrinsic semantics in Euclid's AM: While intrinsic semantics depends on the *subjective* interpretation of a reader (or a computer), relative semantics allows an unambiguous interpretation. According to our **Observation** 6, we can express axioms explicitly using predicate logic PLm ($m \geq 1$). The opposite was the case for definitions: We were unable to express definitions in predicate logic (compare **Observation** 9).

Hilbert formulated 20 axioms[203] arranged in five groups, in which he, in addition to the changed approach to definitions, stated a more precise version of Euclid's geometry.[204] Hilbert's approach to the AM has penetrated the contemporary mathematics until nowadays.

[203] some authors like in [63] p. 17, count 21, maybe depending on the translation. In [77] there are 20 axioms.

[204] [20] pp. 17-18 *"It would be wrong if we considered Hilbert's axioms to be merely a more precise version of Euclid's postulates. [...] Hilbert was [also] convinced that all attempts to define mathematical*

5.10.6 Are Basic Notions Never Defined in Modern AM?

Some sources misleadingly describe contemporary mathematics as never defining basic notions explicitly. For instance, in [42], we find the following misleading description of contemporary AM:

> *"A [mathematical] **definition** is generally a clear demarcation of a concept [definiendum] in a broader context, using another term [definiens] or more other terms [more definientia]. However, this approach causes similar problems as in [mathematical] proofs: It is inevitable that we finally come upon basic notions that cannot be defined this way. [...] Many concepts, particularly the basic notions natural number, point, straight line, distance, area, etc., are then not **defined explicitly**. Instead, they have to be **defined implicitly**, describing their relationships formulated in appropriate axiomatic systems."*[205]

Indeed, we can find examples in contemporary mathematics in which authors omit defining basic explicitely (like Hilbert did for points, lines, and planes in [77] or Ebbinghaus does for sets in [7]). However, there are counterexamples to this approach: In many modern PBM texts, we *do find* explicit definitions of basic notions. A first example of this kind of definition we saw in the definition of natural numbers in [48] (**Definition** 10). We find many other examples, for instance the definition of *vector space* in [51], the definition of *topological space* in [46], or the definition of *probability* in [40]. We quote these examples for later reference:

Definition 12 (Vector Space)

A non-empty set V, together with two maps

$$\begin{array}{ll} (x,y) \to x+y & \text{of } V \times V \text{ to } V, \text{ the } \textbf{\textit{addition}} \\ (\alpha, y) \to \alpha x & \text{of } K \times V \text{ to } V, \text{ the } \textbf{\textit{scalar multiplication}} \end{array}$$

*is called a **vector space over a field** K, if the following properties hold:*

- *V is together with the addition an Abelian group.*
- *For $\alpha, \beta \in K$ and $x, y \in V$ the following axioms are fulfilled:*
  1. $(\alpha + \beta)x = \alpha x + \beta x$,
  2. $\alpha(x + y) = \alpha x + \alpha y$,
  3. $(\alpha\beta)x = \alpha(\beta x)$,
  4. $1x = x$, *where* $1 = 1_K$.[206]

---

objects by their essence were doomed to failure." (own transl. from German)

[205] [42] Vol 1. p. 21 (own transl. from German)

[206] [51] p. 2 (own transl. from German)

Definition 13 (Topological Space)

*A **topological space** is a pair $(X, \mathcal{O})$ consisting of a set X and a set $\mathcal{O}$ of subsets of X (called **open** sets) such that*

- *Axiom 1: Arbitrary unions of open sets are open.*
- *Axiom 2: The intersection of any two open sets is open.*
- *Axiom 3: $\emptyset$ and X are open.*[207]

Definition 14 (Probability)

*A function P defined on a system of events is called **probability**, if it fulfils the following axioms:*

1. *The probability $P(A)$ of an event A is a uniquely determined, non-negative real number with $0 \leq P(A) \leq 1$.*
2. *The probability of the certain event equals one: $P(\Omega) = 1$.*
3. *The probability of any two distinct events $A \cap B = \emptyset$ is or $P(A \cup B) = P(A) + P(B)$.*[208]

These multiple examples show that explicit definitions of basic notions might indeed occur in modern PBM, but only in combination with axioms stated *inside the definition block*. We will call such definitions **relative definitions**. It appears, not only the *lack of definitions* of basic notions is characteristic to modern PBM (like sets are never defined explicitly in [7]), but also the occurrence of explicit relative definitions of basic notions. A modern PBM author can choose between two options: either she omits defining basic notions explicitly and states separate axioms relating these basic notions to each other or defines the basic notions explicitly by using relative definitions containing the corresponding axioms. It leads us to an important requirement for the correct design of FPL:

Requirement 23 (Relative Definitions in FPL)

FPL MUST *support relative definitions (i.e. definitions containing axioms the defined objects must fulfill).*

5.10.7 Does the modern AM really solve SIRP?

A closer look at the examples **Definition** 10, **Definition** 12, **Definition** 13, and **Definition** 14 reveals another phenomenon that creates new doubts if modern AM indeed solves SIRP. Note that in all these examples, the axioms reference to basic notions that are not defined in the corresponding theory:

[207] [46] p. 7 (own transl. from German)
[208] [40] pp. 8-9 (own transl. from German)

- In the axioms inside **Definition** 10 we find references to the notions *"set"*, *"empty"*, *"$\in$"*, *"$=$"*, *"subset"*, being defined outside the theory of natural numbers.
- The axioms inside **Definition** 12 refer to the notions *"Abelian group"*, *"field"*, being defined outside the theory of vector spaces.
- In the axioms inside **Definition** 13 we find the notions *"set"*, *"union"*, *"intersection"*, *"subset"* being defined outside the theory of topological spaces.
- The axioms inside **Definition** 14 refer to notions *"function"*, *"real number"*, *"0"*, *"1"*, *"$\cup$"*, *"$\cap$"*, *"$\emptyset$"*, being defined outside the theory of probability.

These examples demonstrate that if a modern PBM publication is dedicated to a specific mathematical theory, it will usually include relative definitions containing references to basic notions of some external theories, i.e., theories being located outside this publication. It leads us to the following observation:

Observation 12 (References to External Basic Notions)

> *In modern PBM texts, relative definitions of basic notions contain axioms stating their properties. These axioms usually refer to other basic notions that are not defined in the* domain of discourse *of the corresponding PBM text. Thus, definitions of basic notions in one PBM theory $\mathcal{T}_1$ might refer to external mathematical notions from other PBM theories $\mathcal{T}_2, \mathcal{T}_3, \ldots$.*

Schematically, this observation is illustrated in **Figure** 3. It shows that a modern PBM publication dedicated to one theory $\mathcal{T}_1$ may define basic notions of its own domain of discourse using *relative definitions* containing axioms stated inside the theory. However, these axioms are not sufficient to explain *all* semantics of these basic notions. To do so, they refer to other, *more fundamental* basic notions that are outside the domain of discourse of $\mathcal{T}_1$, i.e., in some other, external theory $\mathcal{T}_2$. This process may repeat in $\mathcal{T}_2$.

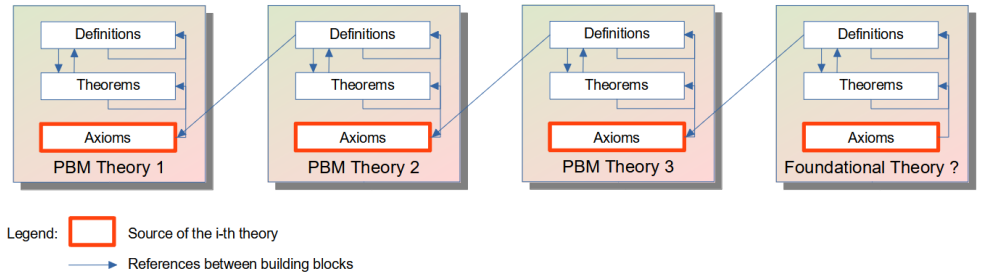


Figure 3: References and sources of PBM publications using Hilbert's AM

Note that modern AM addresses LIRP in each of the theories because the axioms in the theory $\mathcal{T}_i$ explain all *"Why"* questions inside this theory. However, for addressing SIRP, we obviously still need external theories, $\mathcal{T}_2, \mathcal{T}_3, \ldots$. This could create a (potentially long) chain of different PBM theories, from which we have to extract the semantics of their

basic notions. Successful addressing SIRP is again at stake unless we find some kind of foundational theory that successfully addresses both problems: LIRP and SIRP.

5.10.8 SIRP and Predicativism in Mathematics

In 5.8 We mentioned already several philosophical issues of PBM that raised during its historical development, including the dispute about the existence of *actual infinity* and the intuitionist approach to PBM, paradoxes that occurred during attempts to create a foundation of mathematics based on logic (Frege) or naive set theory (Cantor), to mention only a few.

Addressing SIRP continues the list of possible problems related to PBM. SIRP is closely related to the issues raised by **predicativists**. They ask if the language of PBM should allow *impredicative* definitions and properties of mathematical objects or only allow *predicative* ones. Due to [78][209], impredicative definitions contain a self-reference to entities that are being defined. For instance, a *minimal closure "of an operation on a set is ordinarily defined as the intersection of all sets that are closed under [...] the operation. But the minimal closure itself is one of the sets that are closed under applications of the operation."* Thus, to construct a minimal closure, we have to have constructed it already. In predicative definitions, this kind of self-reference is forbidden. An interesting philosophical question is whether we can create mathematics that is bare of any impredicative definitions while successfully avoiding SIRP.

5.10.9 Hilbert's Program and Formalistic AM

Hilbert witnessed all these philosophical problems emerging in the course of the 19th and at the beginning of the 20th century. There were competing philosophical movements in mathematics: the supporters of logicism (like Frege, Russell, and Whitehead), intuitionists and constructivists (like Brouwer and Kronecker), and their opponents who supported non-constructivist mathematics and believed in the existence of the *actual infinity* (like Hilbert himself and the set-theorists Cantor, Zermelo, Fraenkel).

Hilbert was a *"figurehead of formalists"*[210], and was enthusiastic about the AM after his achievements using a modified version of it in *"Foundations of Geometry"* 1899, as compared to Euclid's original 2300 years before. Hilbert believed to be able to modify AM in a formalistic manner even further. We have spoken already about AM in formalism in the context of *Hilbert calculus* we learned about in 5.6.3. Formalists discovered the difference between provability and satisfiability. That difference is illustrated schematically in the **Figure** 4. Thus, formalists strictly separate the syntax of formulas from their semantics, consequently also the derivability (provability using mechanical manipulations of meaningless symbols) of formulas from their satisfiability (i.e., interpretation of being true).

The discoveries of propositional logic PL0 and predicate logic PLm ($m \geq 1$) and the possibility of expressing formulas that are independent of any interpretation expressing

[209] in 2.4 Predicativism
[210] [20] p. 16

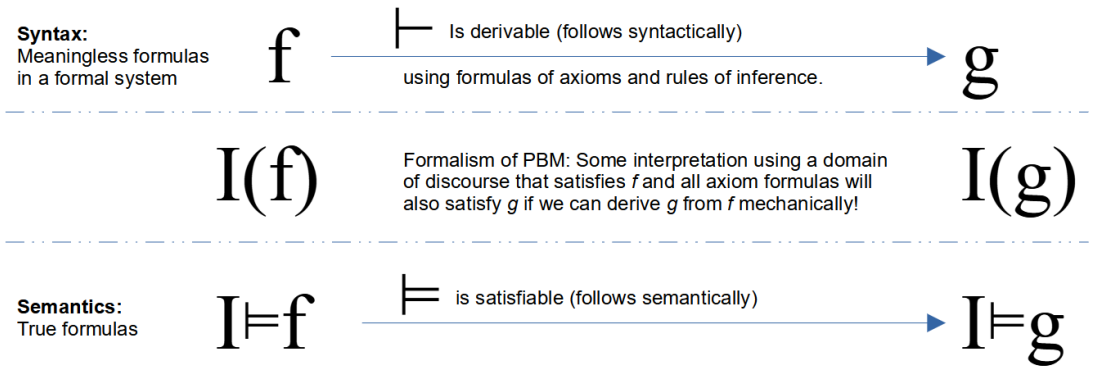


Figure 4: Difference between provable and satisfiable formulas according to [63]

the *Peano axioms* in PL2 or expressing equality in PL2 (compare 5.4) encouraged Hilbert to try to find an *axiomatic system* that would help to prove some meta results mechanically about itself, in particular, the following two of the five quality requirements of axiomatic systems we have introduced in 5.6.4:

1. **Consistency**: $\vdash g$ implies $\nvdash \neg g$ for all formulas g, i.e. we should never be able to prove both, the formula g and its negation $\neg g$.
2. **Completeness**: $\models g$ implies $\vdash g$ for all formulas g, i.e. every valid statement should be also provable.

By creating an axiomatic system like this, Hilbert aimed nothing less than to finish the dispute among the leading mathematicians of his time regarding the question of what are the *"correct"* foundations of mathematics once and for all. This became later known as *"Hilbert's Program"*[211]. At the same time, he would fulfill Leibniz's dream of a *"Characteristica univeralis"*: If Hilbert discovered such an axiomatic system, mathematicians could base all mathematics on it without being afraid of discovering new paradoxes or inconsistencies.

5.10.10 Proofs of Consistency and Completeness

Hilbert was not only enthusiastic in finding an ultimate foundational axiomatic system for PBM but also aimed to prove its consistency and completeness once he found it. As we saw in **Observation** 10, there are some theorems about consistency and completeness that formalists have achieved using AM. As such, they are meta results about axiomatic systems. In general, given an axiomatic system, there are two possibilities to prove such meta results about it:

[211] compare [78] 2.3 Formalism and [20] pp. 26-37

1. An **absolute proof** on a pure syntactical level: Absolute proofs (e.g., of consistency or completeness) are independent from interpretation, because they use only the syntactical features of inference rules and axioms inside a *formal system*. For feasibility reasons, only simple axiomatic systems with a limited number of inference rules and axioms allow such proofs. Consequently, axiomatic systems allowing absolute proofs about their properties have no practical significance for real PBM theories[212].

2. A **relative proof** on a semantical level: Relative proofs are dependent on interpretation, i.e., we need to construct a *domain of discourse* in which we can interpret the (otherwise meaningless) formulas of a formal system as true or false statements. Relative proofs provide the only method we can hope to successfully apply for PBM because of the syntactical complexity of its axiomatic systems. Taking proofs of consistency as an example, the idea is to find a *model* for the axiomatic system. From what we know about models (discussed in 5.4), this proves the *soundness* of the system. Now, its consistency follows immediately. This is because, if all of its axioms are true statements, we cannot derive both: a formula ϕ and its negation $\neg(\phi)$ because one of these formulas would be false, contradicting the fact that we found a model for the whole theory.

Gödel proved in 1929 in his doctoral thesis that the syntax of PL1 is complete. That theorem became later known as the *Gödel's Completeness Theorem*[213]. Albert Skolem (1887 - 1863) prepared 1923 the logical arguments needed to prove (later actually proved by Gödel again) that PL1 has a model, at the same time proving its consistency. This became later known as the *Gödel-Skolem Model Theorem*[214]. Thus, PL1 is an example of a *formal system* that is both: complete and consistent. Indeed, there are more axiomatic systems that are complete and consistent. We can find an example of a complete and consistent *Hilbert calculus* expressed in PL0 in [19][215].

Unfortunately, as we saw in 5.4, PL0's and PL1's syntax are far too simple to express PBM theories we encounter in praxis. Nevertheless, these results indicated that the solution to Hilbert's Program was within reach.

5.10.11 Gödel's Incompleteness Theorems and the End of Hilbert's Program

Indeed, Hilbert was right: the formalistic form of AM enabled deriving metamathematical conclusions about the AM as a method itself. Unfortunately, not like Hilbert expected it to do. Gödel showed 1930 in his famous *Incompleteness Theorems* that a sufficiently complex axiomatic system, i.e., one in which we could express non-trivial mathematics,

[212] An example of such an axiomatic system and a syntactical proof of its consistency and completeness can be found in [20] pp. 22-30, in [22] 71ff.
[213] compare [20] p. 325
[214] [22] p. 343
[215] pp. 98-104

that not complete. Even worse, a system that can prove this result cannot prove its own consistency[216]. We formulate these results as possible limitations to the design of FPL:

Limitation 2 (First Incompleteness Theorem)

> *Due to Gödel, if a consistent formal system is complex enough to formalize arithmetics, then it is not complete.*

Limitation 3 (Second Incompleteness Theorem)

> *Due to Gödel, a* formal system *that is complex enough to formalize the proof of the first incompleteness theorem cannot prove its own consistency.*

In other words, using a formalistic form of AM, Gödel was able to prove a metamathematical result about the method itself, proving that Hilbert's program must fail.

5.10.12 Consequences for Addressing SIRP in Axiomatic Systems

Gödel's incompleteness theorems show that it is impossible to create an axiomatic system of PBM that proves its own consistency. Moreover, there will always be some true statements that remain unprovable in a single axiomatic system.

We now present a novel conclusion from Gödels Second Incompleteness Theorem for addressing SIRP in PBM. A prerequisite for an axiomatic system that would prove its own consistency is the existence of formulas of its axioms that are true independently from any interpretation. Otherwise, there would still be a risk of finding an interpretation under which these formulas become false statements, undermining its consistency. Because Gödel showed that such a system could not exist, it follows that also formulas do not exist that could express the axioms of such a system as true statements in an absolute way, on a purely syntactical level. In other words, a direct consequence from Gödel's Second Incompleteness Theorem is that no axiomatic system exists, in which we could successfully address both, LIRP and SIRP. On the one hand, axioms are a successful way mathematicians address LIRP alone because, by the convention of asserting axioms as true statements, mathematicians accept them as the final answer to the *"Why"* question in a given mathematical theory. However, axioms can never ultimately answer the *"What"* question in that theory since their truth value is dependent on the particular interpretation and the *domain of discourse*, in which mathematicians are willing to interpret the axioms as true statements.

This contradicts the mainstream view on Hilbert's form of AM he invented in the *"Foundations of Geometry"*, which since then found their way to modern mathematics. We quoted this mainstream view from [42] in 5.10.2. This view wrongly suggests that we can successfully get rid of intrinsic definitions of basic mathematical notions and replace them either by relative definitions that contain axioms (examples of which we saw in

[216] see [20] pp. 39, 145

Definition 12, **Definition** 13, or **Definition** 14), or by omitting an explicit definition at all and provide axioms separately (examples of which we observe in Hilbert's [77], or Ebbinghaus's [7]). **Figure** 3 shows a chain of mathematical theories that we need to express the meaning of axioms in every theory. We might hope to end the chain at some *foundational* PBM theory, if we omit intrinsic definitions. We have shown above that the existence of such an ultimate theory would contradict Gödel's Second Theorem. If such a foundational theory existed, it would contain axioms that are tautologies, formulas that are true on a purely syntactical level, no matter how we interpret them, which we demonstrated is impossible.

5.11 The Vicious Mathematical Circle and Addressing SIRP in PBM

In the last subsection, we derived a result that contradicts a contemporary mainstream view: We have shown that modern AM is not able to encode semantics into mathematical objects using axioms successfully. Why is this result significant for the design of FPL? Because we are still looking for some way to encode semantics in FPL so that both humans and computers could share the same interpretation of the same FPL code. In Euclid's ancient form of AM, basic mathematical objects were defined by their essence, using intrinsic definitions. In modern AM, basic mathematical objects are defined by their logical relationships, expressed in axioms. The first, ancient form of AM seems to be not suitable for the design of FPL because it uses intrinsic definitions dependent on a subjective interpretation. In 5.10, we showed that also the modern form of AM does not what it promises: For sufficiently complex axiomatic systems, it is impossible to encode them in a way that they become tautologies. We will always need some subjective interpretation.

This section will show that if we still avoid subjective interpretations and especially intrinsic definitions, we create a fundamental problem that we will call the *"Vicious Mathematical Circle"*. We will also show that the only way to escape this circle is to allow intrinsic definitions at some fundamental level, as Euclid allowed them when defining the basic notions of a *"point"* or a *"line"*. In other words, the ancient type of AM is the only type of *axiomatic method* that allows us to address both at once successfully: LIRP and SIRP. Because intrinsic definitions are dependent on the interpretation of individual subjects, we will conclude that we have to allow the syntax of FPL to be ambiguous enough so that computers can create their own semantical representations of PBM theories and basic mathematical notions. Given the proceeding results, different subjects (both humans and computers) can *never share* individual interpretations. Still, it will be possible to encode semantics in FPL in such a way that both humans and computers agree about the consistency of a theory, despite different interpretations.

5.11.1 Significance of Definitions to Form Models in PBM

We learned in **Observation** 9 that the syntax of PLm ($m \geq 0$) does not allow to formulate definitions and that they fix new domains of discourse and new interpretations

of the other seven types of building blocks in PBM, i.e., axioms, theorems, propositions, lemmas, corollaries, proofs, and conjectures. We also learned that formalists strictly separate the syntax of PLm from its interpretation. Therefore, we can fully express PBM in some formal language if we allow both PLm *and* definitions to be part of the syntax of this formal language. Otherwise, the symbols in which we would formulate the axioms of any theory in such a language would be meaningless. Only if we fix the interpretation and form a *model* of these axioms will we enable the readers of such a formalized theory to interpret it as a consistent one. This observation is novel and important for the design of FPL. Therefore, we formulate it separately:

Observation 13 (Definitions are Models)

> *Mathematical definitions (both intrinsic and relative) in a PBM theory are a syntactical means to express a* ***model*** *for a mathematical theory explicitly in its syntax. Applying this model, a mathematical theory becomes consistent. Its axioms, theorems, and proofs are satisfied by the model.*

We will now address the question if FPL should avoid intrinsic definitions due to the inherent subjectivity of the models they induce and instead always require encoding the semantics of mathematical objects in some absolute, unambiguous way.

5.11.2 SIRP and the Vicious Mathematical Circle (VMC)

Avoiding intrinsic definitions attempting to encode semantics of PBM in some absolute, unambiguous way, inevitably creates a *vicious circle*, we will call the **Vicious Mathematical Circle (VMC)**, that we present in**Figure** 5. The VMC starts with a candidate theory that we hope to be foundational for mathematics.

1. Once we have created a new candidate for a foundational theory of mathematics, we face the fundamental ontological *issue of existence* of basic notions it involves and how *"much"* real objects can we describe using our theory.

2. As we saw in 5.9, by convention, only true statements can be statements about the real world. Trying to address this limitation, we would be lucky to prove the completeness of our theory.

3. Unfortunately, Gödel's First Incompleteness Theorem tells us that our system cannot be complete if it is complex enough for PBM (even when trying to describe simple arithmetics).

4. Making the best of the situation, we add new axioms that postulate the existence of basic notions in our theory (i.e., using the *"∃"* quantor).

5. This causes epistemological problems. How *can we ever know* that our system will now be free of contradictions?

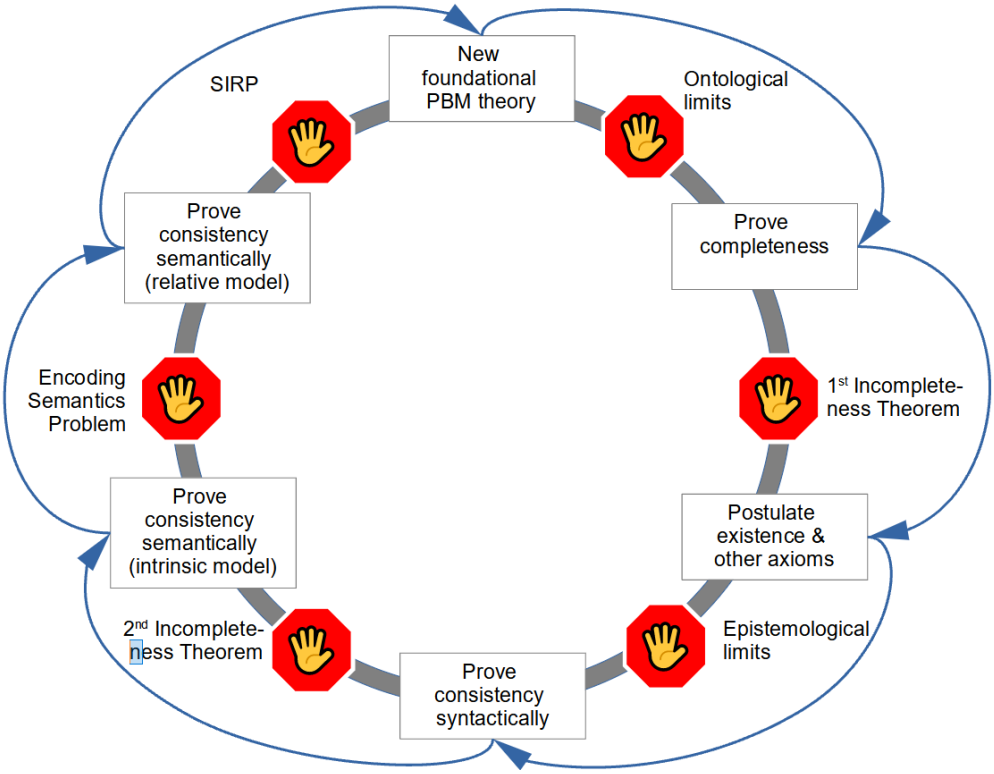


Figure 5: The VMC of Foundations in PBM

6. Trying to address this limitation, we would be lucky to prove the *consistency* of our system using its axioms, in an absolute proof (i.e., on a syntactical level that is independent of any interpretation).

7. Unfortunately, Gödel's Second Incompleteness Theorem tells us that we cannot find absolute proof inside our candidate fundamental system.

8. Making the best of this situation, we try to prove the consistency of our system semantically, i.e., by finding a model for our system.

9. One way to provide a model for our system is to explicitly define the basic notions using intrinsic definitions, in addition to relating them in our axioms, so that an individual reader can create its interpretation of these axioms and interpret them as true statements.

10. Intrinsic definitions bear the risk of being misunderstood. We cannot eliminate the possibility that the reader of our theory (or a computer parsing it) will create a subjective interpretation that identifies our axioms as false statements.

11. Due to this fundamental communication problem that we might call the *Encoding Semantics Problem* (ESP), we decide to replace intrinsic definitions with relative

definitions.

12. Unfortunately, we are trapped in the syntax of our candidate system. We face SIRP: To define the basic notions of our candidate foundational theory using relative definitions, we need a new foundational theory with a new set of symbols. Thus we have to replace our candidate theory with a new one, and the circle starts all over again.

The VMC constitutes a new possible fundamental limitation to the design of FPL. We formulate it for later reference:

Limitation 4 (Vicious Mathematical Circle (VMC))

> *Omitting intrinsic definitions to avoid the ESP causes a Vicious Mathematical Circle (VMC) in trying to explicitly construct basic mathematical notions unambiguously using relative definitions.*

5.11.3 ESP (Encoding Semantics Problem)

Following the *model of communication* due to Claude Shannon (1916 - 2001) and Warren Weaver (1894 - 1978), communication between a *sender* and a *receiver* requires *encoding* a message by the sender, sending the message through a *channel*, and *decoding* the received message by the receiver.[217] The Shannon-Weaver model addresses a possible *noise* in the channel, which can be overcome by using *error-detecting* and even *error-correcting* codes[218]. Thus, the sender can ensure that the receiver will reconstruct the core message by sending a redundant message that includes information enabling the receiver to detect and even recover the original core message. We have to stress that this approach only solves reconstructing the core message on the syntactic level. It does not solve the limitation that the receiver has still to *interpret* the message. Even if the core message is the same as the sender sent it, the receiver might decode a different intrinsic meaning of the sender's vocabulary. It is a possible limitation to the high-level design of FPL because if we try to encode a model, we will have to use some syntax. But the resulting code will still have to be interpreted.

Limitation 5 (ESP (Encoding Semantics Problem))

> *Replacing relative definitions to escape the VMC by intrinsic definitions causes the ESP (*Encoding Semantics Problem*), because the semantics of basic mathematical notions becomes dependent on subjective interpretations. When a subject A (e.g., human or computer) has a model of a mathematical theory $\mathcal{T}_1$ containing intrinsic definitions of mathematical objects, it is not ensured that another subject B will be able to interpret them in such a way that this new interpretation will also be a model for $\mathcal{T}_1$. In other words, B might interpret the* formal system *of the theory*

[217] compare, for instance [94]

[218] According to [95], so-called *Hamming codes* have been widely used as error-correcting codes in digital communication systems.

$\mathcal{T}_1$ as inconsistent. A will hopefully avoid ambiguities by defining the basic notions in $\mathcal{T}_1$ using relative definitions, referring to the syntax and semantics of a more foundational theory $\mathcal{T}_2$ such that both, A and B will have to find a model.

5.11.4 A Case Study: Addressing SIRP in Geometry using ZFC

We intentionally put a question mark *"?"* at the end of the box labeled *"Foundational Theory"* in the **Figure** 3. We demonstrated in 5.10 that no foundational PBM theory can exist that successfully addresses SIRP. The VMC shows which problems we face if we encode semantics to our foundational theory by omitting to define basic notions via intrinsic definitions, only relating them axioms. We will demonstrate this phenomenon once again based on the example of the Zermelo-Fraenkel set theory ZFC. It is a good example for two reasons: First, it is widely accepted as foundational[219] in modern mathematics. Second, it omits intrinsic definitions of sets. Third, it even omits relative definitions of sets since it relates them only in axioms and uses its syntax. ZFC is, therefore, a good example of how modern AM works at the level of a foundational theory.

First, we will demonstrate how we can address the encoding semantics for a given theory $\mathcal{T}_1$ by replacing its intrinsic definitions of basic notions with relative definitions that use the syntax of ZFC as a foundational theory $\mathcal{T}_2$. Finally, we will show how the syntax of ZFC is again inevitably ambiguous so that its meaning is not absolute. Consequently, the axioms of ZFC are in no way tautologies: their truth is dependent on individual, subjective interpretation. Consequently, the semantics of basic notions of the original theory is ambiguous and depend on subjective interpretation. Our case study will show that building a computer that would mechanically accept ZFC as a consistent theory (or any other theory that is based on ZFC), just because many human mathematicians consider it to be consistent, is questionable.

As an example of theory $\mathcal{T}_1$ we take geometry again, in analogy to [36][220]. An example of an axiom in $\mathcal{T}_1$ could be formulated like this:

> *"If x and y are distinct points on a line l, then there is a point z between x and y lying on the line."*

We can formalize it as

$$L(l) \wedge P(x) \wedge P(y) \wedge \neg E(x,y) \wedge I(x,l) \wedge I(y,l) \quad \Rightarrow \\ \exists z(P(z) \wedge B(x,z,y) \wedge I(z,l)). \tag{2}$$

Now, we have to explain these meaningless symbols using *intrinsic definitions*. In other words, we use undefined words like *"point"*, *"line"*, *"equals"*, *"lies on"*, *"lies between"*, etc. so that a (human) reader can create its own model, in which the above axiom becomes a true statement.

[219] compare for instance [36], [7], or [22]
[220] [36] p. 7

Model 1

$$
\begin{array}{rcl}
P(x) & : & x \textit{ is a point.} \\
L(x) & : & x \textit{ is a line.} \\
E(x,y) & : & x \textit{ equals } y. \\
I(x,y) & : & x \textit{ lies on } y. \\
B(x,y,z) & : & y \textit{ lies between } x \textit{ and } z. \\
\exists x(S(x)) & : & \textit{There exists an } x \textit{ such that } S(x). \\
S \Rightarrow T & : & S \textit{ implies } T.
\end{array}
$$

As mentioned above, such intrinsic definitions have highly subjective interpretations. It would be hard to program a computer that interprets them while accepting the axioms as true statements. We face the ESP. To avoid it, we may redefine all notions using the syntax of another theory for which we hope a computer has a model. This another theory might be ZFC - the modern form of set theory. The way we create a new model is not at all construed. We can find this way of presenting geometry in modern PBM books like [49], or [50].

Our new model looks now like this:

Model 2

$$
\begin{array}{rcl}
P(x) & : & x \textit{ is an element of a non-empty set } \mathbb{P}. \\
L(x) & : & x \textit{ is an element of a non-empty set } \mathbb{L} \\
 & & \textit{with } \mathbb{L} \cap \mathbb{P} = \emptyset. \\
E(x,y) & : & x \textit{ equals } y \textit{ in set-theoretical sense, i.e.} \\
 & & \alpha \in x \Longleftrightarrow \alpha \in y. \\
I(x,y) & : & \textit{The ordered pair } (x,y) \textit{ is element of the subset} \\
 & & I \subset \mathbb{P} \times \mathbb{L} \textit{ fulfilling the axioms of incidence.}^{*} \\
B(x,y,z) & : & \textit{The ordered tuple } (x,y,z) \textit{ is a element of} \\
 & & \textit{the subset } B \subset \mathbb{P} \times \mathbb{P} \times \mathbb{P} \\
 & & \textit{fulfilling the axioms or order.}^{\ddagger} \\
\exists x(S(x)) & : & \textit{There exists an } x \textit{ such that } S(x). \\
\exists! x(S(x)) & : & \textit{There exists exactly one } x \textit{ such that } S(x). \\
\forall x(S(x)) & : & S(x) \textit{ holds for all } x. \\
S \Rightarrow T & : & S \textit{ implies } T.
\end{array}
$$

Note how this relative **Model** 2 is based on *relative definitions* that *translate* notions like *"point lying on a line"* and *"point lying on a line between two other points on the line"* into the a set-theoretical syntax *"element of"*, *"ordered tuple"*, *"subset"*, *"set intersection"* $\cap$, *"empty set"* $\emptyset$, while adding some additional formulas describing which properties these elements and relations must fulfil. If A and B consider the set theory to be consistent, they are likely to agree that the **Model** 2 makes the formula 2 a true statement. The additional axioms are now

1. (∗) **Axioms of Incidence**:
   - $P(x) \wedge P(y) \wedge x \neq y \Rightarrow \exists_1 l(L(l) \wedge I(x,l) \wedge I(y,l))$
   - $\forall l(L(l)) \Rightarrow \exists x, y(I(x,l) \wedge I(y,l) \wedge x \neq y)$
   - $\exists x, y, z, l(P(x) \wedge P(y) \wedge P(z) \wedge L(l) \wedge I(x,l) \wedge I(y,l) \wedge \neg I(z,l))$.
2. (‡) **Axioms of Order**:
   - $I(x,l) \wedge I(z,l) \wedge x \neq z \Rightarrow \exists y(B(x,y,z) \wedge L(y,l))$.
   - $I(x,l) \wedge I(y,l) \wedge I(z,l) \wedge B(x,y,z) \Rightarrow B(z,y,z)$.
   - $I(u,l) \wedge I(w,l) \wedge u \neq w \Rightarrow \exists v, x(B(u,v,w) \wedge B(v,w,x))$.
   - $\exists u, v, w, x, l(I(u,l) \wedge I(v,l) \wedge I(w,l) \wedge I(x,l) \wedge B(u,v,w) \wedge B(v,w,x))$.

If A is a human, A might *"translate"* the relative **Model** 2 back to a more subjective, intrinsic model, for instance, by re-stating the axioms of incidence in natural language:

Ad (1) (∗) A's intrinsic model of the *axioms of incidence*:

- If x and y are distinct points, then there is exactly one line going through x and y.
- Every line l has at least two distinct points x, y lying on it.
- There are three points x, y, z such that two of them x, y lie on a line and the third z does not lie on the line.

Ad (2) (‡) A's intrinsic model of the *axioms of order*:

- If there are two distinct points x, y lying on a line l, then there is a third point z lying on that line that lies between x and y.
- If y lies between x and z on a line l then y also lies between z and x on that line.
- For any two distinct points u, w on a line l there are two other points v, x on that line such that v lies between u and w and w lies between v and x.
- On a line l we can always situate four points u, v, w, x on that line in such a way that v lies between u and w and w lies between v and x.

A computer B might verify the validity of the axioms of *"incidence"* and *"in-betweenness"* also by its own intrinsic model, for instance, modelling points and lines as the data structures *integer* and *list*. In practice, both A and B might not need any intrinsic models at all. By convention, A and B might agree on the convention that axioms are *asserted to be true*, no matter what they might mean. What then remains from the relative model is the faith of A and B in the consistency of the set theory only.

ZFC set theory is not the only other theory to be used in relative definitions for creating a mode of our example theory $\mathcal{T}_1$. The above axioms of incidence and the axioms of order

are part of the 20 axioms of Hilbert's work [77].[221] Hilbert proved the consistency of his 20 axioms not using the set theory but using the theory of real and complex numbers.[222] This requires the faith of A and B in the consistency of the theory of real and complex numbers.

5.11.5 Case Study (Cont.): Addressing SIRP in ZFC

First of all, we saw already in 5.8 that even (human) mathematicians have a rather reluctant attitude in their acceptance of the *axiom of choice* as a true statement. If this axiom had an absolute tautological meaning, there would be no dispute at all. However, the axiom of choice is not the only ZFC axiom whose truth is dependent on interpretation. In 5.4.5 we saw that for expressing ZFC, we need the syntax of PL2. In addition to PL2's syntax, set theory contains exactly two additional symbols: *"="* as a symbol for equality, and *"∈"* as a symbol for *"being an element of"*. ZFC never *"defines"* what both symbols are, but the following ZFC axiom relates these two symbols with each other:

- **Axiom of Extensionality**: Two sets are equal if they have the same elements, expressed formally in the syntax of ZFC:

$$\forall xy(\forall z(z \in x \Leftrightarrow z \in y) \Rightarrow x = y).$$

Let's have a closer look at the semantics of the basic concept of the symbol *"="*. In 5.4, we learned that expressing the meaning of "=" on a purely syntactical level is impossible in PL1. Only in PL2 we can express equality syntactically since we can write[223]

$$\forall x(\forall y((x = y \Leftrightarrow \forall \mathcal{P}(\mathcal{P}(x) \Leftrightarrow \mathcal{P}(y))))),$$

where $\mathcal{P}$ denotes any predicate. In the context of ZFC, the axiom of extensionality makes the symbol "=" replaceable by the symbol *"∈"*. The axiom of extensionality can be reversed without causing contradictions[224]. Thus, every expression like $x = y$ can be considered an abbreviation of the more complicated expression $\forall z(z \in x \Leftrightarrow z \in y)$. For this reason, all set-theoretical statements *"may be formulated entirely in terms of the one predicate $x \in A$ (x is an element of A)"*[225]. But all this doesn't help us one iota in explaining what "=" (at least in the context of ZFC) means unless we understand what $\in$ means.

So let us have a closer look at the semantics of the basic concept of *"being an element of"* $\in$. Note that none of the axioms of ZFC explains its meaning. Even if we put all the

[221] compare 5.10.5
[222] [77] pp. 19-24
[223] [19] p. 148
[224] Note that in the original Zermelo's formulation [62] of the axioms, the axiom of extensionality was stated using only an implication $\Rightarrow$ and not using an equivalence $\Leftrightarrow$. However, equivalence is also possible, since if two sets are equal, then they have the same elements (compare [21] p. 45)
[225] [36] p. 7

ZFC axioms together, in which the symbol $\in$ occurs, we only get logical consequences involving this symbol but fail to unambiguously explain what *"being an element of"* means. As mentioned above, we lack a *relative definition* which would allow us to interpret $\in$ unambiguously. To write down such a relative definition, we would need symbols of another theory outside the syntax of ZFC. We stay trapped within the syntax of ZFC because our supply of available symbols will remain insufficient to formulate a relative definition of $\in$!

We saw already in 5.4 that formalists keep semantics fundamentally independent from the syntax of a theory[226]. The syntax of ZFC is no exception: There are many different interpretations of *"$\in$"* that lead to models for ZFC. In [21][227], we find the following classification of ZFC models:

- Models, in which the symbol *"$\in$"* is interpreted as the relation of *"being an element of"* are called *standard models*, while other interpretations of this symbol lead to the so-called *non-standard models*.
- Moreover, there are *transitive* and *non-transitive* models of ZFC. In a transitive model of ZFC, if $z \in x$ and $x \in y$ then $z \in y$. It follows that sets can contain only other sets as their elements, while the most elementary set is the empty set $\emptyset$. In a non-transitive model of ZFC, sets might also contain elements that are not sets. As a consequence, there are potentially many different elementary sets.

We want to demonstrate how it is possible to interpret the symbol *"$\in$"* in different ways still evaluating the ZFC axioms as true statements. We take the ZFC *axiom of infinite set* as a first example:

- **Axiom of Infinity**: There is a set x containing the empty set[228] and with every set z also the set $z \cup \{z\}$:

$$\exists x(\emptyset \in x \wedge \forall z(z \in x \Rightarrow z \cup \{z\} \in x)).$$

Under a transitive standard model, only a single interpretation of the consecutive elements of x is possible:

$$\begin{aligned} a_0 &:= \emptyset \\ a_1 &:= \{\emptyset\} \\ a_2 &:= \{\emptyset, \{\emptyset\}\} \\ a_3 &:= \{\emptyset, \{\emptyset, \{\emptyset\}\}\} \\ &\vdots \end{aligned}$$

Note that in that standard model, for instance from $\emptyset \in a_1$ and $a_1 \in a_2$ it follows that $\emptyset \in a_3$. Under a non-transitive standard model, infinitely many interpretations of the

[226] [22] p. 339
[227] 162 ff.
[228] The *empty set* can be defined in the set-theoretic language as $\emptyset := \{z \mid z \neq z\}$.

infinite set would be valid. For instance, x might contain the elements:

$$\begin{array}{lcl} a_0 & := & \{\emptyset, \clubsuit\} \\ a_1 & := & \{\{\emptyset, \clubsuit\}\} \\ a_2 & := & \{\{\emptyset, \clubsuit\}, \{\{\emptyset, \clubsuit\}\}\} \\ a_3 & := & \{\{\emptyset, \clubsuit\}, \{\{\emptyset, \clubsuit\}, \{\{\emptyset, \clubsuit\}\}\}\} \\ \vdots & & \end{array},$$

where we have used the symbol $\clubsuit$ to denote anything that is not a set. Note that in that non-standard model from $\emptyset \in a_1$ and $a_1 \in a_2$ it does not follow that $\emptyset \in a_3$. (which would be the characteristic of a transitive model).

The next real-world example illustrates the difference between a transitive and non-transitive model in ZFC even more impressively. In many mathematical books, we often find definitions like this one (taken from [27], p. 8):

Definition 15 (Cyclic Group)

> *A group G is defined to be* cyclic *if there exists an element $a \in G$ such that every element of G is of the form a^n (written multiplicatively) or of the form an (written additively) for some integer n.*

What are the semantics of the symbols a and G? G is a *"group"*, but since $a \in G$, G is a also a *"set"* since it has elements. So G has two types: *"group"* and *"set"*. But which type does a have? Well, it will be a *"set"* only under the transitive model since, in this case, only sets can be elements of other sets.[229] Under a non-transitive model, a could be of any type, it still could be a *"set"*, or it could be something else. The point is, no matter which model a mathematician has in her mind, she will be able to interpret the **Definition** 15 properly and evaluate any proven theorem involving cyclic groups and their elements as being true.

Now, imagine we write down the **Definition** 15 in FPL. A computer parsing this definition will have to assign a type to the variable a, at least in use cases in which verifying a given group G is *cyclic* is needed. Ganesalingam ([9]) proposed to introduce the so-called *relational* type in such cases, which only applies *"to objects which do not have fundamental types"*[230]. Following his proposal, a could never be of the type *"set"* or any other fundamental type.[231] But this approach would exclude common use cases in which mathematicians construct groups out of elements that have already a fundamental type (like for instance the additive group of integers $(\mathbb{Z}, +)$ which is cyclic with $a = 1$ or $a = -1$ and in which a would already have a fundamental type *"integer"*).

[229] In this case, in principle, a could have an explicit set representation as some kind of bracketed empty set, e.g. a could be equal $\emptyset$, $\{\emptyset, \emptyset\}$, or $\{\{\{\emptyset, \emptyset\}, \emptyset\}, \emptyset\}$ etc., although no mathematician would care.
[230] [9], 134ff.
[231] This is because a *"set"* establishes a *"fundamental"* type in Ganesalingam's terminology.

So *"what"* kind of object does a mathematician interpret a in the above definition, and which type should a computer assign to it when parsing such a definition? We will have to elaborate a clear answer to this question to design FPL the right way.

5.11.6 Adaptivity and Self-Containment Fail in PBM

Relative definitions circumvent the problem of subjective interpretations but may create their problems. Suppose a basic mathematical notion is defined in theory $\mathcal{T}_1$ using a relative definition referring to other basic notions defined in the *domain of discourse* of some other theory $\mathcal{T}_2$. In that case, two different PBM publications dedicated to theory $\mathcal{T}_2$ might define these basic notions incompatibly. Let us demonstrate these inconsistencies taking for $\mathcal{T}_1$ the definition of probability (**Definition** 14). It makes use of two other basic mathematical notions: *function*, and *real numbers*. We choose these two notions without losing generality; other choices would lead us to the same analytical conclusions. We demonstrate how the definitions of these two notions can differ from each other in different publications:

- **Function**. In mathematical literature, we can find at least two semantical descriptions of a function:
  1. In [36][232], whose *domain of discourse* are classes and sets, we find

     Example 8 (Definition Function)

     > *A* ***function*** *from A to B is a triple of objects (f, A, B), where A and B are classes and f is a subclass of $A \times B$ with the following properties:*
     >
     > ***F1:*** $\forall x \in A, \exists y \in B$ *such that* $(x, y) \in f$.
     > ***F2:*** *If* $(x, y_1) \in f$ *and* $(x, y_2) \in f$, *then* $y_1 = y_2$.

  2. In [37][233], whose *domain of discourse* is combinatorics and basic algebra, we find

     Example 9 (Definition Function (Alternative))

     > *A triple (f, A, B) is called a* ***(total) map****, notated $f : A \to B$, if A and B are sets and $f \subseteq A \times B$ is a right-unique and left-total relation.*

     This source uses the term *"total map"* and states that this is equivalent to the term *"function"* which is *"how maps are called in physics"*[234]

  The two definitions are *semantically incompatible* since they refer to different basic notions (*"class"* vs. *"set"*) and differently axiomatically explained *relative models*:

[232] [36], p. 50
[233] [37], p. 85 (own transl. from German)
[234] [37] p. 85 (own transl. from German): *"Eine andere Sicht der Abbildungen finden wir in der Physik: Hier werden Abbilduingen meist als* Funktionen *bezeichnet [...]"*

Similarities	Differences
– Both examples refer to the same term *"triple"* (f, A, B). – F1 and F2 *mean* the same in 8 as being *"left-total"* and *"right-unique"* in 9, however, this is not apparent in the example. This would be only realized by more advanced mathematicians reading both definitions.	– A and B are *"classes"* in 8 and *"sets"* in 9. – In 8, f is a *"subclass"* of $A \times B$, while in 9, f is a *"subset"* of $f \subseteq A \times B$. – The above difference imply also the following semantical difference, which is not stated explicitly: (f, A, B) is a *"triple of classes"* in 8, while it is a *"triple of sets"* in 9. – Different linguistic structure. – Different terminology.

Table 6: Similarities and Differences of Semantics (Example Function)

In **Example** 8, F1 and F2 are inline-stated *axioms* of the concept *"function"* which itself a *"triple of classes"*, while in **Example** 9, the *"triple of sets"* is a special kind of another concept *"relation"*, that *"left-total"* and *"right-unique"*.

- **Real Numbers**. For the notion of *real numbers*, we observe the same phenomenon of being defined in semantically incompatible ways in different publications. For instance, in [38], real numbers are introduced as a *set* $\mathbb{R}$ whose elements fulfil the *field axioms*, together with the *ordered field axioms*, the *Archimedean axiom*, and the *axiom of completeness*. In [18], real numbers are introduced with almost the same axiomatic system, but the axioms of Archimedes and axioms of completeness are replaced by *Dedekind cuts*. In [48], real numbers are constructed as *equivalence classes* of *Cauchy sequences* of rational numbers that converge to the same limit. The axiomatic system found in this publication consists of the *Peano axioms* dealing with natural numbers. The notion of a *set* is assumed to be known by the readers, exactly as in [38].

We might argue that the theories of real numbers and functions are not fundamental enough to have consistent semantical representations and hope that all differences disappear on the level of a more fundamental theory. For this sake, let us continue our search for the ultimate semantics, focussing, again without loss of generality, on the mathematical concepts of *"set"* and *"class"*. Like it was the case for the terms *"function"* and *"real numbers"*, we will find again semantically incompatible definitions.

- **Set**.
  1. In [36][235], we find

[235] [36], p. 45

Example 10 (Definition Set)

*If $X \in Y$ for some class Y, the X is a **set**. If $X \notin Y$ for any class Y, then X is a proper class.*

2. In [25][236], we find

   Example 11 (Definition Set (Alternative))

   *A **set** is a well-defined collection of objects. The objects of a set are called **elements**.*

3. In [7][237], and in for instance [21][238] we do not find any explicit definition. The term *"set"*[239] is simply used in the Zermelo-Fraenkel axioms, that are stated as separate building blocks (compare 5.11.4). It is worth noting that, consistently to our **Observation** 14, in both sources, partially different axioms are used. For instance, in [7], the *axiom of existence* that is found in [21], is missing. On the other hand, [7] states this axiom for any set using the equality operator $\exists x(x = x)$, and the there is an explicit definition block for the empty set: $\emptyset := \{z \mid z \neq z\}$.

- **Class**.
  1. In [36][240], we find

     Example 12 (Definition Class)

     Our system of axiomatic set theory is based on just two undefined notions: The word class *and the* membership relation $\in$.

     Example 13 (Definition Element)

     *Let x be a class. If x is an element of some class A then x is called an **element**.*

     Example 14 (Definition Equality)

     Let A and B be classes. We define $A = B$ to mean that every class with A as an element also has B as an element, and vice-versa. In symbols

     $$A = B \text{ iif } (\forall X)(A \in X \Rightarrow B \in X \text{ and } B \in X \Rightarrow A \in X).$$

[236] [25], p. 75
[237] [7], 29ff.
[238] [21] pp. 44-52
[239] own. transl. from German *"Menge"*
[240] [36] p. 25

2. In [21][241], we find the following explanation: In the class theory, collections of objects are considered *"not as sets but as classes. If a class turns out to be so 'small' that it can be considered an element of another class without causing any contradictions, we call it a set. Thus, every set is a class, but not vice versa. Classes that are not sets are called 'proper classes'."*

We want to fix these findings in the following observation:

Observation 14 (Relative Definitions Might Cause Inconsistent Semantics)

When a PBM theory refers to external mathematical concepts using relative definitions, retrieving their semantics is difficult since they might be defined in different publications in semantically incompatible ways.

In 5.1, we learned about the process of *adaptivity* as proposed in [9] for natural languages or in [71] for controlled natural languages. These approaches presume that either mathematics as a whole or its publications are *"self-contained"*[242]. The above analysis shows that (if not due to SIRP or the ESP), adaptivity must fail due to the inconsistencies in the relative definitions we encounter in mathematical publications for the same concepts. If mathematics is indeed *"self-contained"*, then only if we accept incompatible semantical representations, similar to those of Benacerraf's incompatible representations of natural numbers using set theory (compare 5.8).

5.11.7 More Bad News

Finally, we want to make two further observations: First, the *lexicon of known mathematical objects grows* over time, both in the breadth of mathematical disciplines and in the depth of their theories, because mathematics is still developing. It is the *objective* growth of the lexicon of available concepts in mathematics. It should not be confused with the *subjective* growth of the lexicon of individual mathematicians reading mathematics or computers parsing mathematical texts we referred to as *adaptivity* in 11. Second, there is a significant *drift of the semantics* of known mathematical objects over time. [15] presents the historical changes in the semantics of different mathematical concepts. As an example, we show the different stages in the semantics of the mathematical concept of a *function*[243]:

- Ancient Babylon: A list of values of a trigonometric function is presented in a tabular form on the *Plimpton 322* cuneiform tablet.
- Ancient Greece: Curves formed by moving objects.

[241] [21] p. 43

[242] [9] p. 5: *"It is this property that allowed us to even formulate our goal of describing mathematics using a fully adaptive theory."*; [71] p. 14: *"The Naproche system [...] tries to establish all proof steps found in the text based on the information gathered from previous parts of the text, in a similar way as a mathematician reads a (logically self-contained) mathematical text if asked not to use his mathematical knowledge originating from other sources."*

[243] compare [15], 131ff.

- Middle Ages: First figures showing planetary movements, depending (as a function) of time, the invention of musical notes (function), and other time-dependent notations and figures
- 17th century: Newton: *"method of fluxions and fluents"*, Leibniz and Bernoulli (Jacob I) use the word *"function"* for the first time, Bernoulli introduces the concept of *"ordinate"*.
- 18th century: Johann Bernoulli, Euler: function as *"analytical term"*, Euler: function as *"freehand drawing curve"*, Lambert et al.: graphical *"figures of empirical data"*.
- 19th century: Fourier, Dirichlet, Dedekind: Functions as *"unambiguous maps"*, Peano, Peirce, Schröder: Generalization to a new concept of *relation* as a set of ordered pairs of elements.
- Turn of 19th/20th centuries/now: Hausdorff: Functions as *"left-total and right-unique relations"*
- Future:?

Again, the phenomenon we refer to here is the *objective* drift of semantics of mathematical concepts. It must be not confused with the *subjective* drift of the semantics of mathematical concepts as observed in [9], and referred to as *reanalysis*.[244]

The bad news is: there is no absolute, constant meaning of mathematical concepts we deal with in PBM. Therefore, we cannot hope that we can program computers in such a way that they can fully *"disambiguate"* the language of mathematics, no matter how rigorous it is (including FPL). Even worse: We cannot hope that even humans can fully *"disambiguate"* the language of mathematics. Even if we think we have consistent fundamental theories, like ZFC, or the class theory, their consistency can neither be proved using these theories (we learn it from Gödel's Second Incompleteness Theorem) nor be proved by finding a model for these theories. Why? Because such a model for a fundamental theory must be based on intrinsic definitions and not relative ones. Relative definitions (like Hilbert's reduction of lines and points to real and complex numbers). But such definitions are not available for foundational theories. They are trapped inside their syntax. Therefore, models of foundational theories must be based on subjective interpretations of what we are trying to express in the language of PBM.

So, what does it all mean for the consistency of our foundational theories? We have no deductive proof for their consistency. Only the observation that nobody hasn't found a new paradox inside our contemporary PBM foundations increases our confidence that

[244] *"Reanalysis is a phenomenon whereby an individual mathematician's analysis of a mathematical expression can change as more mathematics is learnt."* As an example, individual mathematicians might perceive the semantics of the notation x^n as different as $\underbrace{x \cdots x}_{n \text{ copies of } x}$ or as $\exp(x \log(n))$, depending on how much they know about mathematics.

these foundations are consistent. But this is not deductive proof but an inductive one, like the *"proof"* of physical theories based on experimental evidence. Concerning the consistency of foundational theories, mathematicians are not superior to natural scientists.

5.11.8 Escaping the VMC

The above analysis demonstrates that the modern form of AM fails to address SIRP in the case of a foundational theory. The reason for this is simple. Taking ZFC as an example, there are symbols in ZFC like the $"\in"$ relation between two sets that are neither defined explicitly in ZFC nor do the axioms of ZFC suffice to explain the semantics of $"\in"$ in some absolute way. The semantics is dependent on subjective interpretations. To fix the meaning of $"\in"$ unambiguously, we would need new symbols from yet another foundational theory. We have only three possibilities to escape the VMC:

1. The **intuitionistic** approach. Recall from 5.8 that intuitionists try to explicitly *construct* all mathematical objects and notions. This approach avoids intrinsic definitions. However, the use of relative definitions requires a (potentially) unlimited collection of basic mathematical objects and mathematical theories $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \ldots$ about them. The chain of theories we have shown in **Figure** 3 is then infinite, theoretically possible, but practically not feasible.

2. The **circular** approach. It is similar to the intuitionistic approach because we still try to explicitly *construct* all mathematical objects and notions. However, we allow only a finite number of mathematical theories $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$, such that we express the absolute meaning of basic notions in $\mathcal{T}_i$ in the syntax and symbols of $\mathcal{T}_{i+1}$. This approach necessarily creates a circular reference, i.e. there will be at least one cycle of indices i_1 and i_m such that the theory $\mathcal{T}_{i_k}$ models the theory $\mathcal{T}_{i_{k+1}}$ for $k = 1, \ldots, m-1$, and the theory $\mathcal{T}_{i_m}$ models theory $\mathcal{T}_{i_1}$. In practical terms, we only *translate* the code of one theory into the code of another theory on a syntactical level, but we never *explain* the semantics of any of these theories in an absolute way.

3. The **Euclidean** approach. We could allow intrinsic definitions in FPL for theories we consider foundational. Just like Euclid defined the notions *"point"* and *"line"* out of the air, forcing his readers to create their interpretation of what these objects *are*, we could allow an out-of-the-air intrinsic definition of a concept like $"\in"$ in the set theory ZFC. This approach seems to be the best one for the design of FPL. It ensures escaping the VMC like illustrated in **Figure** 6.

Because no additional relative definition is available in a foundational theory, intrinsic definitions of meaningless symbols (like the $"\in"$ relation between two sets) trigger the creation of a subjective model of these symbols. Only this subjective model enables the reader of the theory to interpret axioms involving these symbols (like the axioms of ZFC) as *true* statements and to escape the VMC. This includes non-humans *"readers"* like computers parsing and interpreting the theory. Note that also computers can have

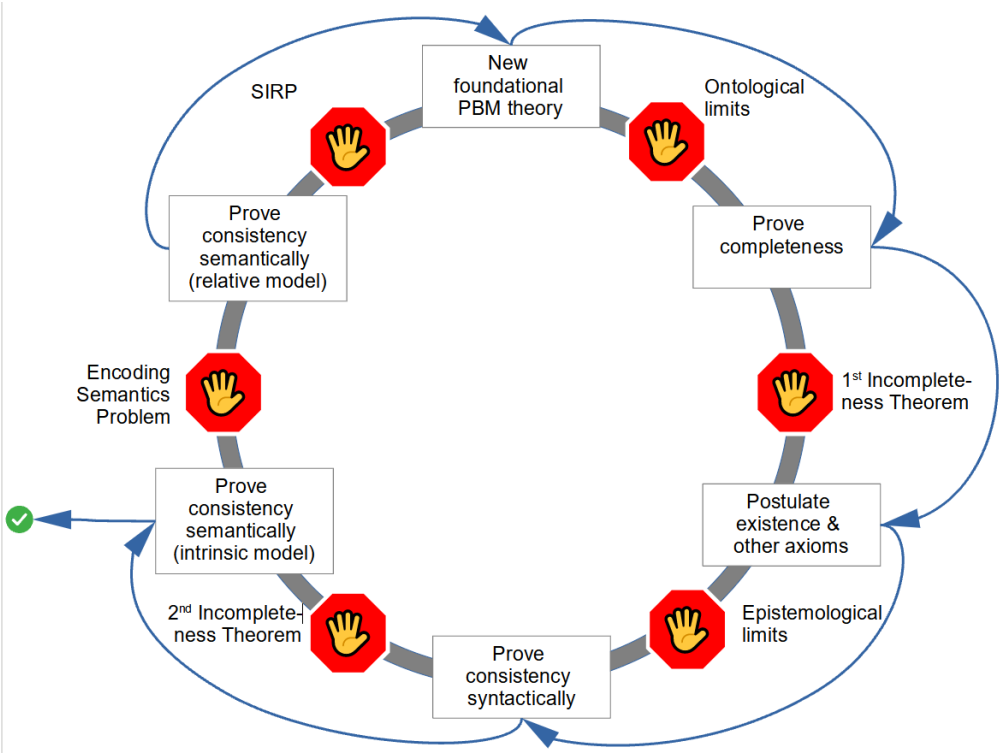


Figure 6: Escaping the VMC in PBM

intrinsic semantics. If a computer interprets a computer program, it has to create its internal representation of the meaningless symbols and strings it is parsing. We postulate it in a new assertion:

Assertion 2 (The Ultimate Source of Semantics in PBM Is Subjective)

> *At the level of a foundational theory, only intrinsic definitions of basic mathematical notions enable a user of FPL (both a human or a computer) to escape the VMC by creating her/its very own, subjective model of its axioms. Such a model makes the fundamental FPL theory (subjectively) a consistent theory.*

The assertion contradicts the (false) mainstream view of the capability of the modern view AM to encode absolute meaning to basic notions just by using axioms. However, our view is not novel. It was used already in Euclid's *"Elements"*, and there are quotations of famous philosophers, logicians, and physicists supporting it:

> *"I say, that the unity, necessitated by the object, cannot be anything but the formal unity of our consciousness in the synthesis of the manifold in our representations. Then and then only do we say that we know an object if we have produced synthetic unity in the manifold of intuition. Such unity is*

impossible, if the intuition could not be produced." (Immanuel Kant, 1781)[245]

"Pure mathematics consists entirely of assertions to the effect that, if such and such a proposition is true of anything, then such and such another proposition is true of that thing. It is essential not to discuss whether the first proposition is true and not to mention what the anything is, of which it is supposed to be true. [...] Thus, mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true." (Bertrand Russell, 1910)[246]

"As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality." (Albert Einstein, 1921)[247]

5.11.9 Consequences for the *"Theory of Everything"* in Physics

The Platonic view we discussed in 5.9 corresponds to the ERH propagating by some physicists. It states that *"there exists an external physical reality completely independent of us humans"*[248]. According to Tegmark, currently, there are two *"most successful theories"* describing parts of this external reality: the *quantum mechanics (QM)* describing the microcosmos and the *general relativity (GR)* describing the macro cosmos. A *"Theory of Everything"*, on which many theoretical physicists are working feverishly, would deliver a *"full description of reality"*[249].

As we have seen, physicists use AM to describe their theories, as mathematicians do. Therefore, our **Assertion** 2 also applies to physical theories based on AM. A *"Theory of Everything"* would necessarily be a foundational theory based on some axioms. However, in this case, the interpretation of its axioms can only be subjective, following **Assertion** 2. There are two alternative consequences from this:

1. First, if the ERH is true, then a *"Theory of Everything"* is unreachable. Otherwise, if it would contradict the **Assertion** 2, since no axiomatically-based *"Theory of Everything"* can *"fully describe"* the external world. It only fully describes some subjective interpretation of this physical world.

2. By contraposition, if (someday) somebody succeeds in finding a *"Theory of Everything"*, then the ERH must be false. A *"Theory of Everything"* can only fully describe the world if it does not exist independently from us humans and our interpretation.

[245] [17] p. 87
[246] [92]
[247] [91]
[248] [55] p. 254
[249] [55] p. 255

The above arguments lead us to another parallel between abstract mathematical and physical objects, complementing **Table** 5. QM states that the observation of a subject forces the **wave function** to collapse so that an observed particle takes a specific **quantum state**. Similarly, the (subjective) interpretation of intrinsic definitions forces the statements of a theory to become **true** or **false** statements. Both observation and interpretation are subjective and interact with reality (that might be external or not).

5.11.10 Consequences for the design of FPL

The evidence gathered above contradicts the promise of the modern AM of being able to encode an absolute meaning to basic concepts using axioms[250]. At least, modern AM will fail to do so when we attempt to formulate the meaning of basic concepts of a foundational theory through axioms. In this case, we will be limited to the symbols of the theory, and we won't be able to express the meaning of its basic symbols just by relating them in axioms. It would require axioms that are tautologies, true statements independent of subjective interpretation, but such axioms are not available in sufficiently complex axiomatic systems of PBM, only in PL0 and PL1.

The danger of that false promise is that it could bring us on the wrong track in the design of FPL. For instance, Printer states that *"neither intuition nor intelligence is needed to go through the steps of a proof. Ideally, it should be possible for a computer to verify or not a proof is correct."*[251] The promise is false since we gain nothing having a computer able to verify the steps of a proof from axioms mechanically. In addition, the computer needs to be able to interpret these axioms as true statements. As we have seen above, we are limited to the syntax of some foundational theory, and it is impossible to interpret its axioms in an absolute, unambiguous way. Moreover, it seems to be difficult to program a computer to understand what concepts like *"being equal"* or *"being an element of"* mean.

You may think that there is an easy workaround: We could just program the computer to simply *assert* the truth of an axiom it encounters during the parsing process. But this workaround is illusory. It is well-known that it is possible to conclude true statements from false statements.[252]. So if, for some reason, our axioms happen to be untrue, the computer is likely to certify the correctness of proofs of theorems that are false.

But ambiguities in interpreting foundational theories and their basic concepts are even worse because they are not limited to computers. Also, humans will inevitably stumble upon the impossibility to interpret the symbols of a foundational theory in an absolute (i.e., independent from the subject) way. Both computers and humans will be trapped in the syntax of the foundational theory. Note that prose sentences attached to the formal ZFC axioms like *"There exists a set that does not contain any elements"* or *"Two sets are equal if they have the same elements"* only seemingly help the reader to interpret the for-

[250] Especially, this promise is stated in [42] p. 21, [36] p. 6, or [20] p. 18
[251] [36] p. 6
[252] compare [25], p. 23

mulas. They leave the human reader (or computer parser) with the problem of *knowing* what *"being equal"* or *"being an element of"* means. For this reason, approaches to automatically derive the semantics of mathematical statements as described in frameworks like [9] for natural languages or like [71] for controlled natural languages are condemned to fail when it comes to interpreting the semantics foundational theories like ZFC. All that such computer programs would achieve when representing the semantics of the axioms of a foundational theory like ZFC would be to restate these axioms rather than interpret their semantics and evaluate these axioms as being true.

Above, we have identified *intrinsic* and *relative* definitions, and their advantages and disadvantages. The following **Table** 7 lists the pros and cons for using both types of definitions.

Definitions	Advantages	Disadvantages
Intrinsic	– Do not require additional symbols as those available in the theory. – The only way to escape the VMC.	– Create a *subjective* model. – Cause the ESP, need relative definitions to solve it. – It is not ensured that two subjects find a model of the theory.
Relative	– Allow to share a common, *objective* model of a given theory $\mathcal{T}_1$. – Solves the ESP by contructing the basic mathematical notions and expressing axioms of $\mathcal{T}_1$ explicitely using the symbols of a more foundational mathematical theory $\mathcal{T}_2$.	– $\mathcal{T}_1$ requires the symbols $\mathcal{T}_2$ to construct its basic mathematical notions and express its axioms explicitely. – Shifts the problem of finding a model from the symbols of $\mathcal{T}_1$ to those of $\mathcal{T}_2$. – Might cause inconsistent semantical representations accross different PBM publications containing a description of $\mathcal{T}_2$. – Cause the VMC, need intrinsic definitions to escape it.

Table 7: Pros and Cons of Intrinsic and Relative Definitions

Recall that we required FPL to support relative definitions in **Requirement** 23. In case we have not a fundamental theory expressed in FPL, we might prefer to use relative definitions for basic mathematical notions of one theory by constructing them in the syntax and symbolic of other, more foundational theories (like in the cases **Definition** 12, **Definition** 13, and **Definition** 14). This approach has the advantage of avoiding subjective interpretations. Still, it might potentially cause semantically incompatible

representations of the semantics of these objects as we saw in 5.11.6 (see **Observation** 14). Intrinsic definitions have the disadvantage of needing subjective interpretations but avoid incompatible representations of semantics and provide the only way to escape the VMC (see 2). For this reason, we cannot prohibit using intrinsic definitions. This is a new requirement:

Requirement 24 (Intrinsic Definitions in FPL)

> *The FPL standard* MUST *allow* ***intrinsic definitions*** *for mathematical notions to escape the VMC.*

The above analysis also supports our **Requirement** 21, in which we stated to avoid building any foundations into the syntax of FPL. When designing FPL, we have to avoid using axiomatic systems at any cost. Otherwise, these axioms would inevitably create semantical problems like the ESP (although we might address it using relative definitions) or the VMC (which is intractable without intrinsic definitions). Therefore, axiomatical foundations built into the syntax of FPL would require, at some point, intrinsic definitions, creating the danger that some end-users will interpret our axioms as *false*, making FPL useless for them.

5.12 Symbolic Notation

5.12.1 Objective

We identified *specific symbolic notation* as a feature of PBM in 5.1. In this subsection, we want to analyze the purpose of the symbolic notation in PBM and identify the way we have to take symbolic notation into account in the FPL specification.

5.12.2 Purpose of Symbolic Notation in PBM

There are deviating views of what the purpose of symbolic notation in mathematics is:

1. From Euclid's *"Elements"* 250 BC until the 16th century, mathematicians had perceived geometry as the foundation of all mathematics. Doing mathematics was the same as solving geometrical problems by construction, and no algebraical approaches were known. François Viète offered in his work *"Supplement of geometry"* published 1593 *"symbolic analysis of geometrical problems [...] as the first step in a three-step process that could render a geometrical solution"*[253]. However, his proposal created a big semantical problem. Suppose the variables like a and b in a solution denote the lengths of two segments. How could a third variable, say x in an equation like $x = a \cdot b$ (which is semantically the area of a rectangle), ever denote a segment again? How could areas denote the same as segments? René Descartes (1596 - 1650) solved this semantical problem in *La Géométrie* published 1637 by introducing a unit length and applying the *Intercept Theorem* for lines. Thus, for

[253] [72]

its inventors at the turn of the 16th century, symbolic notation was the first step in a proposed *method* to solve concrete geometrical problems algebraically.

2. Descartes' view on the purpose of symbolic notation in mathematics corresponds to the popular view because it is *educated* like that. At school, we learn to solve word problems or geometrical problems algebraically. The first step is always to translate these problems into an equation or inequation. In this popular view, equations and symbols are the actual *"language of mathematics"*.

3. According to Ganesalingam, *"[...] the primary function of symbolic mathematics is to abbreviate material that would be too cumbersome to state with text alone."*[254], while *"variables [...] serve as a mathematical alternative to anaphor."*[255]. In this linguistic view, symbols in mathematics abbreviate its semantics.

4. As we saw in 5.4, from the view of logicians and formalists, symbolic notation (as a part of the syntax of a formal language) is independent of the semantics. The purpose of symbolic notation is to generate and manipulate the formulas of a formal system. A formula at this stage is *"nothing more than a sequence of meaningless symbols that we combine in a determined way."*[256]. As we saw in 5.6, the notion of the *derivabilty* of a formula is fundamentally different from its meaning and truth[257].

All these different purposes seemingly contradict each other. Nevertheless, they all are valid in different contexts and should be taken into account. For instance, algebraic calculations and algebraic manipulation of equations are often parts of mathematical proofs, which corresponds to the first purpose, in Descartes' sense. Mathematical proofs also refer to variables just because they are good abbreviations of long definitions, which corresponds to Ganesalingam's linguistic view. Finally, computers can better handle symbolic notation, not only when manipulating equations but also to derive logical proofs, which corresponds to the formalistic view. We summarize this in the following observation:

Observation 15 (Purposes of Symbolic Notation)

> *The symbolic notation has at least three purposes in PBM: It serves the manipulation of equations, it abbreviates mathematical definitions and statements that would be otherwise too longish in natural language, and it helps (computers) to interpret/verify or formulate mathematical proofs.*

Historically, symbolic notation has ever been changing in mathematics and strongly depends on authors.

---

[254] [9] p. 17
[255] [9] p. 31
[256] [22] p. 72 (own transl. from German)
[257] *"The aim of transforming the expressions according to the given rules as part of the syntax of a logical calculus [...] is to reflect (at least partially) our 'logical thinking'. [...] The goal is to generate all possible conclusions as formulas from a set of symbols, given syntactical rules to derive these formulas. This would enable machines to derive conclusions, which is a worthwhile goal."* [57] p. 28-29, (own transl. from German)

1. Symbolic notation was not known in mathematics until the end of the 16th century[258], as we saw above.

2. Symbolic notation is a feature that has been since then, extremely mutable. To take the notation of logical statement as an example: Frege used in his *"Gesetze der Arithmetik"* (*"Basics of Arithmetics"*) published in 1894[259] a significantly different notation from that used by Russell and Whitehead in *"Principia Mathematica"* published in 1910, and this notation is significantly different from the contemporary notation of logical statements[260].

3. Symbolic notation depends upon the author of a mathematical publication, even if, across different publications, it "means" the same. For instance, one author might choose for *multiplication* the notation *"$a \cdot b$"*, another author the notation *"$a \times b$"*, and yet another author the notation *"ab"*.

4. The contemporary symbolic notation in PBM is very rich. We can classify symbols into the following sub-types:
   a) Symbols for variables, including indexed variables, for instance $x, y, z, \ldots$, $a_i, \ldots, b_i, \ldots, \alpha_n^{(m)}, \ldots$
   b) Symbols for unique mathematical objects, for instance $\emptyset$, $\mathbb{N}$, $\mathbb{R}$, $0, 1, 2, \ldots$, $\aleph_0, \aleph_1, \ldots$ etc.
   c) Symbols for functional terms and predicates, including
      - unary functional terms predicates: Succesor(n), $|x|$, $-x$, f', z^*, etc.
      - unary predicates: Unique(n), Positive(n), etc.
      - binary functional terms: $2 + 3$, $\frac{1}{4}$, etc.
      - binary predicates: $2 < 3$, $x = y$ $x \in y$, etc.
      - n-ary functional terms: $\sum a_i$, $\prod b_i$, etc.
      - n-ary predicates: IsBasis($\mathrm{a}_1, \ldots, \mathrm{a}_\mathrm{n}$) or matrix predicates, e.g.,

$$\begin{pmatrix} 1 & 2 & 3 \\ a & b & c \end{pmatrix}$$

   d) Bracketing symbols (), [], {}
   e) Defining symbol :=
   f) Quantors $\exists$, $\exists_1$, $\forall$
   g) Logical formulas $\neg a$, $a \Rightarrow b$, $a \Leftrightarrow b$, $a \vee b$, $a \wedge b$, etc.

[258] see [52] *"The invention of the mathematical formula"* 11ff.
[259] being a "first comprehensive approach to define logical foundations of mathematics"[22] p. 31 (own transl.)
[260] Figures from extracts of the original publications can be found [22] pp. 31, 41

Concluding on these findings, we can make the following requirement:

Requirement 25 (FPL Syntax Independent From Specific Notation)

As far as possible, the FPL specification SHALL NOT *prescribe symbolic notation. The syntax of FPL* MUST *be independent of specific notation.*

5.12.3 Ranges and Index Variables in PBM

In PBM, we can often find enumerations of variables using index variables like $a_1, a_2, \ldots, a_n$, or

$$\begin{bmatrix} a_{11} & a_{12} & \ldots \\ \vdots & \ddots & \\ a_{K1} & & a_{KK} \end{bmatrix}.$$

Enumerations are very useful since they abbreviate the notation even further, focussing on the main structure of compound mathematical objects like vectors, matrices, or sequencies. In the next **Example** 15, the index variable k is in the range between 1 and n:

Example 15

"The formula

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

holds for all natural numbers n.[261] *"*

The domain of discourse of the symbols $1, k, n$ is *"natural numbers"*, and it enables us to interpret the formula on the right side of the equation as well as to exactly know that k takes a *finite* number of distinct values between 1 and n and *which* exactly are these values.

It seems that index variables and their range bounds are always positive integers in PBM, but this is not the case. In the so-called *families* a_i, $i \in I$, the variable I is allowed to be an arbitrary set or even an arbitrary class, it can be infinite or fail to be ordered.[262].

There are also other types of ranges in PBM, in which stand alone variables range between two bounds instead of index variables. The bounds of these types of ranges and the index variabes are typically real numbers.

Example 16

"Let $f : I \to \mathbb{R}$ be a continuous function and let F be the antiderivative of f. Then

$$\int_a^b f(x)dx = F(b) - F(a)$$

[261] from [38], p. 2
[262] [36], 66ff.

holds for all $a, b \in I$.[263] "

In **Example** 16, the real numbers a, b are bounds for the stand-alone variable variable x inside the integral. Moreover, a and b are used as arguments of the antiderivative F on the right side of the equation.

The most cases of ranges in PBM require *strictly ordered* mathematical objects, i.e., for which only one of the cases $x < y$, $x > y$, or $x = y$ can occur at once.

This fact is implicitly taken for granted in most PBM publications, although a computer checking the correctness of such publications would have to check not only if it is possible to create a consistent semantical representation of the bounds of the range (like *"natural number"* or *"real number"*) but also if this representation is compatible with some strict order defined on its *domain of discourse*. A malicious publication could contain ranges of non-ordered objects like complex numbers, vectors, or matrices, and such errors would be hard to discover for computers.

Example 17

"Let $(a_n)_{n\in\mathbb{N}}$ *be a sequence of real numbers. The series* $\sum_{n=0}^{\infty} a_n$ *converges if and only if [...]*[264] "

Example 18

"Let $f : [a, \infty[\to \mathbb{R}$ *be a Riemann-integrable function over every interval* $[a, R]$, $a < R < \infty$. *We set*

$$\int_a^{\infty} f(x)dx := \lim_{R\to\infty} \int_a^R f(x)dx,$$

if this limit exists. By analogy, we define the integral $\int_{-\infty}^{a} f(x)dx$ *for a function* $f :]-\infty, a] \to \mathbb{R}$.[265] "

In the **Example** 17, the variable n ranges from 0 which is a natural number to ∞ which is not a natural number. In the **Example** 18, the variable x ranges in the integral from a, which is a real number, to ∞, which is not a real number. The same holds for the variable R in $R \to \infty$. Mixing semantically incompatible types for the ranges makes it even more difficult for computers to discover inconsistencies.

5.12.4 How to express ranges in FPL

In only some simple cases, ranges are finite and taking distinct values, like in the **Example** 15. In programming languages like our reference language C#, such constructs can be implemented using *loops* like `for` or `while`. Unfortunately, PBM expressions cannot be handled like this, for the following reasons:

[263] [38] p. 144
[264] from [38] p. 37
[265] from [38] p. 152

1. PBM ranges may involve *countably*, or even *uncountably* many values for the variable inside the range.
2. The range may be unbound.
3. The domain of discourse of the range or index variable might involve unordered mathematical objects.

For these reasons loops are inappropriate to express ranges in PBM. Therefore, we formulate another requirement:

Requirement 26 (Ranges and Index Variables in FPL)

> *The FPL syntax* SHOULD *support ranges that represent simple cases with* finite *bounds and distinct values but also more complicated ones, in which we have* infinite *bounds, continuous values or even unordered values.*

Moreover, enumerations are a pure syntactical feature in FPL as a meta language. We shall not confuse them with mathematical objects like tuples, vectors, matrices, or sequences for which we will still need separate definitions in FPL.

5.13 Definitions, Declarations, and Assignments in FPL

5.13.1 Objective

In this subsection, we are going to investigate mathematical definitions and their subtypes. We want to identify all features of mathematical definitions that are relevant for the high-level design of FPL.

5.13.2 Definitions Must Be Identifiable

Mathematical definitions can occur in PBM texts as separate building blocks that are typeset separately and start with the word **Definition**. (see for instance, **Example** 12 or **Example** 13). Another common possibility occurring in PBM publications is to define mathematical objects in the running text, like for instance *" [...] A ring is called* ***factorial*** *(or a* ***unique factorization ring****) if it is entire and if every element* $\neq 0$ *has a unique factorization into irreducible elements. [...] "*[266]. The *"[...]"* indicate that there is no separate definition block in this quotation. But readers might much harder recognize such texts as definitions and the publications typeset the terms to-be-defined in other font, for instance, bold font like in our quotation. It would be also significantly harder to program a computer to parse and recognize such texts as definitions. Therefore, we restate or **Requirement** 3 and **Requirement** 4 that FPL must be a structured language rather than one written in prose and that its syntax should use separate all eight building blocks of PBM, including definitions.

[266] [27] p. 111

5.13.3 Definitions of Mathematical Objects

Consider the following definition block, kind of which we find, for instance, in [42]:

Example 19 (Definition: Binary Relation)

Let V, W be sets. Any subset R of the Cartesian product $V \times W$ *is called a* ***binary relation*** *(or just* ***relation****). The set V is called the* ***domain*** *of R and W is called the* ***codomain*** *of R. We write vRw whenever $(u, w) \in R$. R is called:*

- ***right-unique****, if from vRw_1 and vRw_2 it follows that $w_1 = w_2$ for the respective $v \in V$, $w_1, w_2 \in W$,*
- ***left-unique*** *(or* ***injective****), if from v_1Rw and v_2Rw it follows that $v_1 = v_2$ for the respective $v_1, v_2 \in V$, $w \in W$,*
- ***unique****, if R is right-unique and left-unique.*
- ***left-total****, if for all $v \in V$ there is a $w \in W$ with vRw,*
- ***right-total*** *(or* ***surjective****), if for all $v \in V$ there is a $w \in W$ with vRw,*
- ...

This example demonstrates the following further features of mathematical definitions:

1. In a definition, it is possible to introduce more than one new term and symbolic notation (e.g. *"binary relation"*, *"domain"*, *"codomain"*, *"left-unique"*, etc.).
2. Among all the terms introduced in a definition, there is only one main term (*"binary relation"*). Other terms are introduced as the **properties** of the main term.
3. The main term introduces a new **type** - the type of a *"binary relation"*. The type can be referenced through its definition in the surrounding text. We have identified this already in the **Requirement** 15.
4. Properties have also *types*. The type of the properties can be either *predicative property* (e.g. a relation R is either *"left-unique"* or it is not), or the type might require another mathematical definition (e.g. the *"domain"* and the *"codomain"* are of type *"set"*).
5. For every term, alternative names can be defined (e.g. *"binary relation"* vs. *"relation"*, or *"right-total"* vs. *"surjective"*).
6. Some of the properties of the main term are **optional** and some are **mandatory**. For instance, every binary relation R introduced by the text *"Let $R \subseteq U \times W$ be a binary relation"* will have a domain and a codomain. However, it will not be left-unique necessarily. For R to be left-unique, we would require this explicitly, like in the *"Let $R \subseteq U \times W$ be a left-unique binary relation."*

7. A mathematical object **inherits** all of the mandatory properties of the parent object from which we derive it. Optional properties might become mandatory, but only if we explicitly require them. For instance, we could have a definition like

   Example 20 (Definition: Function)

   > *Let V, W be sets. A binary relation $f \subseteq V \times W$ is called a **function**, if f is left-total and right-unique. We write $f(v) = w$ whenever ufw for some $v \in V$ and $w \in W$. $f(x)$ is called the **value at** u of the function f. The set $f[V] := \{f(v) \mid v \in V\}$ is called the **image** of the function f. The set $f^{-1}[W] := \{u \in V \mid \exists w \in W(f(u) = w)\}$ is called the **inverse image** of f. For every $w \in W$, the set $f^{-1}(w) := \{u \in U \mid f(u) = w\}$ is called the **fiber** of w under the function f.*

   This kind of **inheritance** of types can be found, for instance, in [37]. The new mathematical type *"function"* inherits all mandatory properties of a relation (e.g. *"domain"* and *"codomain"*). The properties *"left-total"* and *"right-unique"* that were only optional for a binary relation, become now mandatory for a function since we require them.

8. A definition might depend on zero, one or more other objects of some particular type. For instance, when defining a binary relation as we observed in the above example, we need two objects V and W being of the type *"set"*. In other words, whatever mandatory properties sets might have (like being equal if they have the same elements), mathematicians automatically assume that the objects V and W have these properties. Moreover, since R is a set itself (it is a subset of a Cartesian product), it inherits all of the properties of a set.

Some of our observations of how mathematical definitions work have much in common with the way object-oriented (OO) programming works. We take the language C# as an example of an OO language, but we could have taken any other OO language. We list these similarities and differences in the **Table** 8.

Inheritance in OO programming creates SIRP. What is the class from which all other classes can inherit? In C#, this problem is addressed so that there is an in-built class `object` serving this purpose. In FPL, we are going to copy this idea. All defined mathematical objects in FPL will be derived from an out-of-the-air type `obj`. We formulate this as a new requirement:

Requirement 27 (OO-like Mathematical Definitions)

> *FPL* MUST *allow OO-like inheritance for mathematical definitions. There is an in-built parent class `obj` from which all classes inherit. In particular: (i) Definitions introduce a new type (like a new class in OO programming), (ii) A new type can be inherited from an already defined type, (iii) A definition allows none or a specific number of typed arguments, (iv) A definition allows the distinction between mandatory and optional typed properties and their inheritance. (v) A definition allows overriding of inherited objects.*

Mathematical Definitions	C# classes
Similarities	
Introduce new types that can be declared.	
E.g., using the *"let"* keyword like in *"Let R be a binary relation"*.	E.g., `BinaryRelation r;`
Allow properties having types	
Allow inheritance of types and properties	
Require zero, one, or more other typed mathematical objects	Allow constructors with zero, one, or more typed arguments
Differences (Excerpt)	
Allow alternative terms	Do not allow alternative names
Allow properties to be optional or mandatory	not applicable
not applicable	Allows access modifiers (like public, private, etc.)
not applicable	Allow methods and events

Table 8: Similarities and Differences of Mathematics and OO Programming

We have to reflect *inheritance* in the context of the `is` operator we have required in the **Requirement** 16. Recall that there were two contexts in which we can use this operator: In an axiomatic context, we can postulate or assert that some variable is of a specific type. In the checking context, we use the `is` operator to check if an instance is of a specific type. We have to modify this last requirement a little bit. If a mathematical type B inherits its properties from another type A, a variable x of a type B has both the properties of B and A. Therefore, in the checking context, `is(x,B)` and `is(x,A)` should both be evaluated to be true. This leads us to another requirement:

Requirement 28 (The `is` Operator and Inheritance)

> *In the checking context of the `is(x,<type>)` operator, it* MUST *return a truth value (either* true *or* false*), based on whether `<type>` is in the* inheritance tree *of the instance of the variable `x`.*

Another feature of definitions is the possibility of stating axioms inside their structure blocks, like in the **Example** 13 or the **Example** 14. Referring to what we stated above, axioms would be special kinds of mandatory properties being predicates. A more complex example was given in the **Example** 12, although the mandatory properties there were not called *"axioms"*, but simply *"properties"*. We summarize this finding in the following design decision:

Requirement 29 (Axioms Inside and Outside Definitions)

> *FPL* SHOULD *allow stating axioms inside the block of a mathematical definition, in which case they would be mandatory predicates this mathematical object must fulfill. The alternative is to define these properties as separate axioms or outside the definition block.*

5.13.4 Definitions of Functional Terms

Definitions of *functional terms* (compare 5.4) have a totally different syntactical structure as definitions of mathematical objects that has to be considered in the FPL specification. This observation is novel. Consider a definition like this one (taken from [48]):

Example 21 (Addition of Natural Numbers)

*The **addition** of natural numbers m, n is defined inductively as follows:*

$$n + 0 := n, \qquad n + m^* := (n + m)^*.$$

First of all, as we learned from the semantics of PL2 (compare **Definition** 9) and the **Observation** 13, a mathematical definition of a functional term has to encode an explicit *model* of the new symbols introduced. A model of a functional term is a *function* $f : \mathbb{D}^n \to \mathbb{D}$ where $\mathbb{D}$ is the current *domain of discourse*. In the above **Example** 21, the domain of discourse are *"natural numbers"* and their addition is a function taking two arguements of type *"natural number"* (m, n) and producing a new mathematical object $m + n$ that is also of the type *"natural number"*. In contrast to definitions of new types we dealt with in 5.13.3, FPL needs to distiguish another type of definitions do not follow this pattern. Obviously, addition has neither a type we could check via the `is` operator, nor is it inherited from `obj` or any other class. It has its own syntactical pattern and we formulate this as a new requirement:

Requirement 30 (Definitions of Functional Terms in fpl)

FPL MUST *provide a separate syntax for definitions of **functional terms**, requiring **typed parameters** and mapping them to a typed mathematical object (its **typed image**). The types of the parameters and of the image are defined using definitions inherited from `obj`, while the definitions of functional terms need a new keyword for this pattern like `func`.*

5.13.5 Definitions of Predicates

Similarly like definitions of functional terms, we will have to distinguish definitions of *predicates* (compare 5.4) in the syntax of FPL from other types of definitions, since they have another structure. Also this observation is novel. Consider the following definition (taken from [48]):

Example 22 (Order Relation of Natural Numbers)

*Let the natural numbers m, n be given. We say that m is **smaller or equal** n, formally $n \leq m$, if m is some predecessor of n or $m = n$.*

As we can see, this kind of mathematical definitions has internally the structure of a predicate. While the variables m, n are typed variables in some *domain of discourse* (here: *"natural numbers"*), the notion *"m is some predecessor of n or $m = n$"* can

be expressed as a PL1 predicate. The definition, together with the *interpretation* in the specific domain of discourse *"natural numbers"* enables the readers to assign a clear *truth* value to any expression like $m \leq n$ they encounter in a PBM text. Once again, a notion like $m \leq n$ has neither a type (it cannot be inherited from `obj`), nor it maps a tuple of natural numbers to a new one (it cannot be a functional term `func`). We need a separate keyword to distiguish such definitions in FPL and formulate this as a new requirement:

Requirement 31 (Definitions of Predicates in fpl)

> *FPL* MUST *provide a separate syntax for definitions of* ***predicates****, requiring* ***typed parameters*** *and containing a PLm expression in their body. We need a new keyword for this pattern like* `pred`*.*

5.13.6 Declarations in PBM

Other features of definitions become only apparent if we investigate how variables are *declared* in PBM texts. Temporary assignments occur, among others, in the following cases:

1. Premises of theorems. Often it is done using the notion *"Let [...variable(s)...] be [...mathematical object(s)...]"*. Note that this kind of premises is not limited to theorems but can also be found in definition blocks, like in the **Example** 19. We can also find them in proofs of theorems.

2. Variables *bound* by quantors, for instance, *"For all [...variable(s)...] it holds that [...condition...]"*,

3. Temporary assumptions. In particular, in *proofs by contradiction*, we encounter assumptions of the type *"Suppose, for [...variable(s)...] it holds that [...condition...]"*. We revoke this temporary assumption when we show a contradiction in the proof.

All these examples show that when variables are assigned a type in PBM texts, a specific scope is needed. In natural and controlled natural languages, identifying such scopes is limited because of the different writing styles of the authors. Scoping of variables is a common technique in programming languages and is much easier if the language structures according to a specific syntax. Therefore, we make the following design decision:

Requirement 32 (Blocks to Scope Variables)

> *FPL* SHALL *use blocks to easily identify the scope of variables.*

Moreover, the syntax should allow an easy distinction between definitions of types and declarations of a variable of these types. Declarations in FPL can be accomplished by listing the variables and then the type they shall have inside their scope. In order to be able to express at least PL2 in FPL, we need the possibility to declare variables with types `func` and `pred`. We formulate this as a separate requirement:

Requirement 33 (Declaration of Variables (at least PL2))

FPL syntax SHALL *enable us to declare variables in their scopes, covering all types that can be defined inherited from* `obj`*, as well as the type of functional objects* `func`*, as well as the type of predicates* `pred`*.*

5.13.7 Assignments in PBM

In the **Example** 20, we have seen examples of how certain variables can be assigned values using the signed " := " The sign is only a convention. In some texts, we find also other symbols, like " = "[267], or " :⇔ "[268] as alternatives in assignments like these. There are differences between assigning a value to a variable and declaring its type. In most programming languages, the interpreter has to *"decide"* which internal type a variable gets when assigning it a value. Similarly, in PBM, mathematicians reading a text have an idea of which type a variable is after its assignment. In the above example, mathematicians would typically determine the type of the *image* of a function $f[V]$ as a *"set"*, because the assigned value uses the so-called *set-builder notation*[269].

Determining which type the variable might be after the assignment is crucial in PBM because the type is decisive for interpreting logical formulas involving the variable as true or false. On the other side, it might be technically very hard for a computer (or even for a human) to determine the type after the assignment in natural or even controlled natural languages. These difficulties have several reasons, for instance:

- When writing $x := 1$, the symbol 1 might denote the corresponding natural number, the integer, the rational number, the real number, the complex number, the unity matrix, the neutral element of multiplication in a ring, etc.
- Not all mathematical objects are appropriate candidates to be assigned to variables. For instance, assigning the empty set to a variable, like in $x := \emptyset$ is allowed, but mathematicians would found assignments like $x := Set$ not well-defined. Why? Because the assigned object has to be unique. Note that in natural language, the formulations *"Let x be the empty set."* and *"Let $x := \emptyset$."* would be equivalent, while only the notion *"Let x be a set."* makes sense, the sentence *"Let $x := Set$"* makes no sense for a mathematician. A computer would stick at this point.
- Unfortunately, just determining the type and the uniqueness of the right side of the assignment is not enough for programming a computer to handle assignments properly. In some circumstances, mathematicians would also accept assignments that are neither unique nor of known type. For instance, a mathematician could make an assignment like $z_n := z_{n-1} + f(z_{n-1})$ and study its behavior for specific initial values of rational, real, and complex numbers z_0 and specific rational, real-,

[267] for instance in [32] p. 86 for the assignment of Fibonacci numbers $F_{n+1} = F_n + F_n - 1$
[268] for instance in [22] p. 133 in the formula $\models (\neg\phi) :\Leftrightarrow \not\models \phi$
[269] [25] p. 76

and complex-valued functions f. The reason why mathematicians would accept such an assignment is that they *"know"* that a function $f(z_{n-1})$ is always *injective*. Thus, mathematicians could, *in principle*, disambiguate a term like $z_{n-1} + f(z_{n-1})$ after fixing the specific *domain of discourse*.

Despite these technical diffuculties to program a computer to handle assignments properly, we require that assignments has to be allowed in FPL:

Requirement 34 (Assignments of Variables in FPL)

FPL syntax SHALL *support assignments of values and expressions to variables (e.g., via the symbol :=). An FPL interpreter* SHOULD *be accept the assignment only if it can verify the uniqueness of the assigned value and the compatibility of the types between the assigned value and the assignee variable.*

5.13.8 Need for Delegates

When discussing some feasibility issues in 5.5.6, we addressed already the problem of introducing the (potentially) infinitely many symbols $0, 1, 2, 3, \ldots$ into the syntax of FPL and identifying them with the type `Nat` of natural numbers. Now, we face yet another practical problem: How to interpret them. The interpretation of intrinsic definitions (**Requirement** 24) using *deterministic* interpreters (compare **Requirement** 10) seems to be not feasible to implement. Therefore, we need a flexible interpretation that can be user-defined. We must enable (human) mathematicians who decide to write intrinsic definitions in FPL to help the available FPL interpreter to interpret them in the inteded way. For instance, there might be a use case in which the interpreter has to determine the truth of a (seemingly simple) expression like $2 + 3 = 5$ but disposes only of the PBM *Peano axioms* for natural numbers (compare **Definition** 10) combined with the definition of the addition of natural numbers (**Example** 21).

A computer can create its own representations of data types like integers or natural numbers but these interpretations are at the moment not accessible for FPL interpreters. We have to change it by allowing FPL interpreters to *delegate* interpretation tasks to the surrunding computer environment. We formulate this as a new requirement:

Requirement 35 (Support for Delegation in FPL)

FPL interpreters are deterministic. Therefore, they MUST *support* ***delegation*** *to support user-defined interpretation of intrinsic definitions.*

5.13.9 Compound Parameters and Implicit Properties in PBM

There are cases in PBM in which we wish to create a compound mathematical object that requires other mathematical objects or functional terms of a special type that is initialized when we initialize the compound mathematical object. For instance, we might wish to define an *algebraic structure* with an own binary operation that is not *arbitrary*. The

binary operation of the structure should operate exactly on this algebraic structure and on nothing else. For instance, if we define an *additive group* $(G, +)$, then we expect the functional term $"+"$ be a binary operation valid only in the context of G but not outside this context. Moreover, if we have define $(G, +)$ as commutative, we expect $"+"$ to fulfill the *commutativity law* $x + y = y + x$ for all $x, y \in G$, whenever we use it. Furthermore, if we have two instances of such additive groups, say $(G_1, +)$ and $(G_2, +)$, we expect the symbol $"+"$ to denote two different binary operations, for instance, one could be commutative, and the other could be not commutative, although both use the same symbol $"+"$.

How can we achieve this kind of behavior in a formal language like FPL? Obviously, we need to make sure that we can somehow define the symbols G and $"+"$ together, although G is a mathematical object while $"+"$ is a functional term. Moreover, we have to remember their properties (e.g., $"+"$ being commutative or not). We formulate this as a new requirement:

Requirement 36 (Compound Parameters and Implicit Properties)

FPL MUST *support definitions with compound parameters having implicit properties.*

5.14 Identification of Semantically Different Objects in FPL

5.14.1 Objective

Sometimes, mathematicians tend to *identify* mathematical objects that have formally different types. We observed examples of identification in the mathematics of Ancient Greece[270]. Ganesalingam[271] points to modern examples of this phenomenon, including the identification of natural numbers with positive integers, real numbers with complex numbers, etc. In this subsection, we are going to investigate if identification should, and if yes, how it could be addressed in FPL.

5.14.2 Analysis

Despite how mathematicians otherwise try to keep their language possibly rigorous, they seem to ignore the rigor of language when it comes to identification. Mathematicians identify semantically different objects, like natural numbers and non-negative integers, for different pragmatic reasons: They either aim to *simplify the notation*, to introduce a *new convention of dealing* with mathematical objects or to *highlight the compatibility* of certain properties of objects of different types that seemingly have nothing in common. Concerning the high-level design of FPL, we should focus on the key observation:

[270] In [60], Book 1, Definition 3, Euclid identifies the mathematical object *"extremities of a line"* (more modern, we would say the endpoints of line) with the object *"point"*. Another example can be found in Definition 6, in which he identifies *"extremities of a surface"* with the object *"line"*. Note that Euclid never explained the justification for this identification, neither did he formulate a separate axiom for this.

[271] [9] 181ff

Observation 16 (Informal Identification is Common)

Identification of mathematical objects of formally different types is common in mathematics and exercised for pragmatic reasons, even at the cost of semantically incompatible interpretations.

In order to keep FPL user-friendly, we therefore should allow identification in FPL. However, FPL must allow identification only in a way that is acceptable by both humans and computers. The question is, therefore, how FPL should treat identification on the syntactical level. We have to introduce some formal syntactical means allowing identification and do it in such a way that a computer parsing such FPL code will (at least under some circumstances) accept it. Once again, we can have a closer look at existing programming languages and investigate if they have a phenomenon similar to identification in mathematics and how they treat it. Indeed, in existing programming languages there is a similar phenomenon, called *type casting* or *type conversion*. Most programming languages (like python, javascript, PHP, C#, etc.) allow implicit and explicit type conversions. The following example shows implicit data type conversion in C#:

Example 23 (Implicit Data Type Conversion in C#)

```
int i=2;
double j = 3.2;
Console.Writline(i + j + " is of type " + (i + j).GetType());
// this code will output the string
// 5.2 is of type System.Double
```

Note that the variables `i` and `j` have different types (`int` and `double`) but the C# interpreter allows to add both variables in the expression `i+j`, resulting in an object of the type `double`. In fact, this example contains even yet another implicit data type conversion: The whole output string `5.2 is of type System.Double` is an object of type `string`, although it concatenates objects of the type `int` (like the variable `i`), `double` (like the variable `j`), `Type` (like the expression `(i + j).GetType()`), and `string` (like the text `" is of type "`).

The following example (as a continuation of the previous examples) shows how data types can be explicitely converted in C#:

Example 24 (Implicit Data Type Conversion in C#)

```
// These lines will be accepted by the C# interpreter
// and will output 2 and 3
Console.WriteLine((double)i);
Console.WriteLine((int)j);
// this line will not be accepted by the C# interpreter and cause the error
'Cannot convert type 'int' to 'bool'
// Console.WriteLine((bool)i);
```

In the particular syntax of C#, an expression like `<some type> variable` instructs the interpreter to try to explicitly cast the type of a variable (whatever this type at runtime might be) to the type `sometype`. The above examples show that explicit data type conversion is often but not always *"accepted"* by an interpreter of a programming language. By the way, this limitation also applies to implicit data type conversion, which is not always possible either. The exact rules by which an interpreter allows or rejects (an implicit or explicit) data type conversion deviate from case to case and from programming language to programming language.

5.14.3 Conclusions

We can learn from our analysis that *identification* is a phenomenon that is not limited to mathematics only. It is very similar to data type casting in programming languages. Computer scientists have developed ways to deal with it on a syntactic level when designing interpreters of computer languages. We formulate, therefore, a new requirement for FPL:

Requirement 37 (Identification in FPL)

> *FPL* SHOULD *allow identification by providing means of casting different data types to each other. If a casting like this is possible or not is subject to the specific implementation of FPL interpreters.*

5.15 What else can we learn from modern programming languages?

5.15.1 Objective

Many sources dealing with the language of PBM like [9], [32] or with formal systems like [57], [22], or with set theory like [36], [7], suggest that we can express PBM in some higher predicate logic PLm . This is true, however only a partial description of how rich the syntax of modern PBM is. We have identified already many syntactical features of PBM going beyond the syntax of PLm. One exception are mathematical definitions. They cannot be expressed as any formal system written in PLm, because the syntax of a formal system is always pre-defined and can only interpret in some (externally) defined *domain of discourse* (compare **Observation** 5.5.5 and **Requirement** 14). However, definitions *increase* the vocabulary of a given formal system by introducing new models to it (**Observation** 13). Other extensions of syntax of PBM going beyond the syntax of pure predicate logic are declarations and assignments of variables (compare 5.13) as well as identifications (compare 5.14).

In this subsection, we will learn even more syntactical features of PBM that go far beyond the syntax of PLm and that we have to take into account when designing FPL. We will demonstrate that PBM supports many features, like defining *algorithms*, *loops*, or *generics*.

5.15.2 Procedural Constructs

Procedural programming languages allow writing an *algorithm*, i.e., *"a set of rules that must be followed when solving a particular problem."*[272] Surprisingly, there are examples of algorithm-like constructs that we can encounter in all types of building blocks in PBM:

Example 25 (Algorithmic Theorem: Euclidean Algorithm)

Let $a, b \in \mathbb{N}$. The sequence $(r_i)_{i\in\mathbb{N}}$ of natural numbers defined by[273]

$$\begin{array}{rcllll}
a & := & q_0 \cdot b & + \; r_1 & \text{with} & 0 < r_1 < b \\
b & := & q_1 \cdot r_1 & + \; r_2 & \text{with} & 0 < r_2 < r_1 \\
r_1 & := & q_2 \cdot r_2 & + \; r_3 & \text{with} & 0 < r_3 < r_2 \\
 & \vdots & & & & \\
r_{n-3} & := & q_{n-2} \cdot r_{n-2} & + \; r_{n-1} & \text{with} & 0 < r_{n-1} < r_{n-2} \\
r_{n-2} & := & q_{n-1} \cdot r_{n-1} & + \; r_n & \text{with} & 0 < r_n < r_{n-1} \\
r_{n-1} & := & q_n \cdot r_n & & &
\end{array}$$

is monotonically decreasing

$$r_0 = b > r_1 > r_2 > \ldots > r_{n-1} > r_n$$

with a minimal element r_n that equals the greatest common divisor of a and b.

Example 26 (Algorithmic Proof: The Intermediate Value Theorem)

The intermediate value theorem states:[274] *If $f : [a, b] \subset \mathbb{R}$ is a continuous function with $f(a) < 0$ and $f(b) > 0$ (respectively $f(a) > 0$ and $f(b) < 0$), then there ist a $p \in [a, b]$ with $f(p) = 0$. Proofs of this theorem usually use the so called* nested intervals method. *This is an algorithmic method constructing inductively a sequence of intervals $[a_n, b_n] \subset [a, b]$ for all $n \in \mathbb{N}$ with the properties:*

1. *$[a_n, b_n] \subset [a_{n-1}, b_{n-1}]$ for $n \geq 1$.*
2. *$b_n - a_n = 2^{-n}(b - a)$*
3. *$f(a_n) \leq 0$, $f(b_n) \geq 0$. (respectively $f(a_n) \geq 0$, $f(b_n) \leq 0$.)*

Example 27 (Algorithmic Formula/Definition: Continued Fraction)

A continued fraction[275] *is a ratio built from the positive integers $q_0, q_1, q_2, \ldots \in \mathbb{Z}$ of the following form:*

$$[q_0; q_1, q_2, \ldots] := q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots}}}.$$

[272] [87]

[273] the theorem can be found in [43], Proposition 2 of Book VII, or in almost every book about elementary number theory.

Example 28 (Algorithmic Axiom: Infinite Set)

One of the Zermelo-Fraenkel (ZF) axioms states:[276] *There exists a set x such that*

$$\begin{array}{ll} (i) & \emptyset \in x \\ (ii) & u \in x \Rightarrow (u \cup \{u\}) \in x \end{array}$$

All of the above examples involve some *repeated application* of the same formula. This is procedural in the sense that in programming languages repetitions are accomplished using so-called **loops**, examples of such are *while* and *for* loops. A computer repeats a loop as long as a condition is met. Inside a loop, a calculation can be done. Note that **Example** 25 involves a *finite* loop, the examples **Example** 26, **Example** 27, and **Example** 28 involve infinite loops.

In combination with assignments, loops are very useful because they allow defining (infinely) many objects at once like in **Example** 25 or functions that are recursive in nature (mathematicians call them also *"inductive"*), like[277]

Example 29 (Definition: Factorial)

The sequence $x_n = n!$ defining factorials can be defined inductively as

$$\begin{cases} x_1 & := & 1 \\ x_{n+1} & := & (n+1)x_n \quad \textit{for all } n \in \mathbb{N}. \end{cases}$$

Many programming languages provide the possibility to define functions called *recursive* to adapt this kind of definitions. For instance, in C#, the corresponding code would be

Example 30 (Factorial in C#)

```
public int factorial(int x)
{
   if (number == 1)
      return 1;
   else
      return x * factorial(x - 1);
}
```

Apart from some syntactical specifics of a particular programming language used in this example, there is one important characteristic that would be also necessary for a possible FPL syntax: The predicate *"factorial(x)"* is introduced in the definition `public int factorial(int x)` and is re-used again in the line `return x * factorial(x - 1)`. The **Example** 30 also shows another main difference between the capabilities of FPL and usual programming languages. The C# and any other programming language

[277] A real example taken from [32] p. 86

are important to define recursive functions that halt at a specific value. Note that the C# program decrements a given value of x until it reaches the value 1. In the mathematical version in the **Example** 29, we start with this value $x_1 = 1$ and then define higher and higher values of the sequence. Thus, the mathematical version does not address the halting problem, and it does it not by intention: It defines infinitely many values of the sequence. The programming version only defines how to calculate a specific sequence number and ensures that the function will require only a finite number of steps to do so.

The above analysis shows us, that we need some additional requirements:

Requirement 38 (Recursion in FPL)

FPL SHALL *allow recursive linguistic constructs like it is the case in PBM expressing inductive definitions.*

Requirement 39 (Flow Control in FPL)

FPL SHALL *allow flow control, including cases or loops to be repreated depending an a condition to be matched. Moreover, FPL* SHOULD *allow* infinite *loops.*

Last but not least, the definitions of the factorial in **Example** 29 and **Example** 30 demonstrate that *impredicative* self-reference in definitions (compare semantical issues raised in 5.10.8) should be allowed in FPL which is a new requirement:

Requirement 40 (Allow Self-reference in FPL Definitions)

FPL SHALL *allow self-reference in definitions.*

5.15.3 Generics

Another invention we can look up in modern programming languages is *generics*. In C#, generics *"introduce the concept of type parameters [...] which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code."*[278]. Therefore, the main idea of generics is to define some functionality requiring unknown types until we create an explicit instance of a class. A novel observation is that PBM requires a similar linguistic feature to generics. For instance, when defining a *binary operation* we learn that it is a *"mapping* $S \times S \mapsto S$ *on some set* S*"*[279]. Although such definitions usually fall back into the language of ZFC (like in this example), we learned in 5.8 that this might be only a passing trend in PBM. It is not important if S is a *"set"* or some other type of mathematical object. What is only important is that the symbol S fixes some *type*, some *domain of discourse*. If we combine two variables to a binary operation, we *require* that they have the same type in a fixed domain of discourse. Otherwise, the term *"binary*

[278] [111]
[279] We can find such a definition in many algebra books, for instance [27] or [37]

operation" would lose its sense.

We stated already the **Requirement** 27 that we will inherit all mathematical objects in FPL from an inbuilt parent type `obj`. Generics are different from inheritance. They only fix the type in definitions. Therefore, we formulate a new requirement:

Requirement 41 (Generics in FPL)

FPL MUST *support generic types.*

5.16 Self-Containment of Theories in FPL

5.16.1 Objective

We encountered the notion of *self-containment* already in 5.1 when it was formulated as characteristic criterion of PBM texts. Only now can we precisely define what self-containment of PBM means. In this subsection, we will make up for the missing definition of self-containment.

5.16.2 Analysis

In 5.10.3, we defined the *transtitive hull* $H(\mathcal{T})$ of all building blocks x and y of a mathematical theory $\mathcal{T}$ such that y refers back to x, which we denoted by $x \to y$. We asked the question, of which kind of blocks the **source**

$$S(\mathcal{T}) := \{x \in \mathcal{T} \mid x \neq z \text{ for all } y, z \in T \text{ with } y \to z\}$$

consists of. We now renew this question for a theory $\mathcal{T}$ that is expressed in FPL. To answer the question more easily, we refine the notion of *"reference"* $x \to y$ in the following way:

- According to **Requirement** 4, FPL expresses definitions, theorems, propositions, lemmas, corollaries, axioms, conjectures, and proofs as separate building blocks.
- According to **Principle** 5, a theory like $\mathcal{T}$ has its namespace. Therefore, we can require that every such building block $x \in \mathcal{T}$ has some unique identifier inside the namespace of $\mathcal{T}$. This can be the name of the defined object or a theorem-like block, axiom, or conjecture. Multiple proofs for the same theorem-like block can be identifiable by numbering them consecutively and attaching these numbers to the unique name of the theorem-like block they are proving.
- The existence of *relative definitions* (compare 5.10) and foundational theories indicates that, typically, there will be multiple theories and namespaces $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Therefore, we have to allow references $x \to y$ between the building blocks having identifiers in the namespaces of theories that might be different, i.e.,

  $$x \to y, \qquad x \in \mathcal{T}_i \wedge y \in \mathcal{T}_j$$

  for some $1 \leq i \leq j \leq n$.

- Finally, we specify the notion of a *"reference"* as follows: $x \to y$ shall hold if and only if at least one of the following properties hold:
  - y contains at least one variable declaration of the type defined in x.
  - y contains a predicate whose expression refers to the identifier of x.
  - y has a property whose type refers to the identifier of x.
  - If y is a theorem-like building block with the proofs $p_1, \ldots, p_k$ end if $x \to p_i$ for some $1 \leq i \leq k$, then $x \to y$.
  - y is a definition of a mathematical object derived from the type defined in x.
  - y is a definition of a functional term such that the type of at least one of its parameters or the type of its image is defined in x.

With these extensions, we can re-define the source S for all theories accross all namespaces:

$$S(\mathcal{T}_1, \ldots, \mathcal{T}_n) := \{x \in \mathcal{T}_i \mid x \neq z \text{ for all } y, z \in T_j \text{ with } y \to z, 1 \leq j \leq n\}.$$

5.16.3 Criteria for Self-Containment in FPL

The above definition of the source spanning all n theories is the key to define self-containment in FPL exactly. We list all *necessary* criteria for self-containment in FPL. If they are *sufficient*, we cannot say. To our best knowledge, the sources we used in this document provide no strict definition of self-containment based on which we could answer this question. However, some of these sources use the term *"self-containment"* extensively in the context of the language of mathematics. Therefore, we dare to define self-containment by providing our definition. The definition is novel. We assert that the following criteria suffice to *define* self-containment sufficiently exactly:

Assertion 3 (Self-Containement in FPL)

Let $T := \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ be a collection of FPL theories. We call T ***self-contained****, if it fulfills the following properties:*

1. *All building blocks in T have* unique *identifiers.*
2. *The reference relation $x \to y$ is* closed *in T, i.e., for all x, y with $(x \to y)$ there exist some indices $1 \leq i \leq j \leq n$ such that $x \in T_i$ and $y \in T_j$)*[280].
3. *The source $S(T)$ is not empty*[281].
4. *S contains only axioms and intrinsic definitions.*

[280] In other words, there are references pointing outside T (all identifiers are be defined in some theory $T_i \in T$).

[281] If it was, then all building blocks have a predecessor. In other words, T contain (vicious) semantical or logical circles we want to exclude).

5. *No axioms in S contradict each other*[282].

6. *All theorems in T have at least one proof.*

7. *All proofs are correct*[283].

8. *No proof in T contains a reference to a conjecture or another proof.*

The above criteria should help build FPL interpreters, since self-containment of all parsed FPL theories will be one of the consistency criteria such an interpreter has to check and report to the user after interpretation. This is a new requirement:

Requirement 42 (Checking Self-Containment)

Every FPL interpreter SHOULD *check the self-containment of the theories it had parsed and report any violations of it to the end-users.*

[282] This criterion is important for the consistency of the collection of FPL theories T as a whole. For instance, this collection cannot contain both Euclidean geometry, and non-Euclidean geometry since both theories contain the *parallel axiom* and its negation as true statements. Putting these axioms together would create an inconsistent theory as a whole. In other words, the axioms in S must be *complete* in the sense of the quality criteria we defined for axiomatic systems in 5.6.4.

[283] Meaning that the corresponding theorem-like building block is derivable from the available axioms and inference rules.

6 High-Level Design of the FPL Standard

6.1 All Observations

The following table summarizes all observations we have made in the analytic part.

Observation 1	Experts variate in characterizing what PBM actually is and what it is not.
Observation 2	Available criteria of PBM texts are not sufficient discriminant criteria for distinguishing mathematical texts from other texts.
Observation 3	PBM being 'self-contained' means both: complementing a lexicon of semantics and completing arguments.
Observation 4	Experts disagree which are the building blocks of PBM, in particular if axioms are; significant differences in terminology (e.g. umbrella, conjectures vs. theorems, need for proofs of corollaries, or whether definitions are mathematical statements).
Observation 5	Nearly 50% of typical mathematical publications are not formulated in PBM.
Observation 6	We can assign a truth value to axioms, theorems, lemmas, corollaries, and propositions in PBM.
Observation 7	We can (potentially) assign a truth value to conjectures in PBM.
Observation 8	We can assign a truth value to proofs in PBM.
Observation 9	We cannot assign a truth value to defininitions in PBM.
Observation 10	There are metaresults about AM as a method that might constrain the way of how FPL can be designed.
Observation 11	Ontological questions are out of scope in PBM.
Observation 12	PBM theories might refer to external mathematical concepts from other PBM theories.
Observation 13	Definitions encode models into PBM.
Observation 14	Relative definitions might cause inconsistent semantics PBM.
Observation 15	Symbolic notation abbreviates mathematical definitions and serves the symbolic manipulation of equations and logical formulas.
Observation 16	Informal identification of semantically incompatible mathematical objects is common in PBM.

Table 9: FPL Cheat Sheet: All Observations

6.2 All Requirements

The following table summarizes all requirements we derived in the proceeding sections. For the details of the specification, the original full formulation of the requirements is decisive.

Requirement 1	Express PBM only, nothing else.	MUST

Requirement 2	Support a clear separation of the concepts of truth and accuracy.	MUST
Requirement 3	Express PBM in a structured code to distinguish it from text passages written in prose.	MUST
Requirement 4	Support eight building blocks of PBM: definitions, theorems, propositions, lemmas, corollaries, axioms, proofs, and conjectures.	MUST
Requirement 5	Allow embedding FPL code into surrounding text that might contain non-PBM contents.	MUST
Requirement 6	Support localization	SHOULD
Requirement 7	Allow distinguishing four theorem-like building blocks: theorems, propositions, lemmas, and corollaries.	SHOULD
Requirement 8	Allow zero to many proofs of theorem-like building blocks and distinguish them syntactically from unproved conjectures.	MUST
Requirement 9	Disambiguate FPL code on a syntactical level using appropriate parentheses rules.	MUST
Requirement 10	Allow only deterministic interpreters to disambiguate FPL code on a semantical level.	MUST
Requirement 11	Incorporate the syntax of predicate logic (at least PL2).	MUST
Requirement 12	Allow referring to (unproven) conjectures in mathematical proofs and interpret such proofs accordingly.	MUST
Requirement 13	Support nesting and linking different logical steps and proofs into more complex ones.	MUST
Requirement 14	Use definitions as a meta-language to introduce new syntax and new domains of discourses allowing to interpret the formulas.	MUST
Requirement 15	Definitions introduce new types that can be used to declare FPL variables.	MUST
Requirement 16	Support for asserting and checking a type of a variable (`is` operator)	MUST
Requirement 17	Support flexible notation while defining new predicates, functional terms, or mathematical objects.	MUST
Requirement 18	Allow a free configuration of axioms and inference rules.	MUST
Requirement 19	Look&feel of notation possibly similar modern notation. Ideally, exploit the possibilities of LaTeX.	SHOULD
Requirement 20	Accept different levels of detail in mathematical proofs. If requested, provide additional details for validated proofs.	SHOULD
Requirement 21	Independence from foundations: No built-in axiomatic system of PBM in the syntax of FPL.	MUST
Requirement 22	Support both, intuitionistic and non-intuitionistic arguments and mathematical objects.	SHOULD

Requirement 36	Support definitions with compound parameters having implicit properties.	MUST
Requirement 23	Support for relative definitions	MUST
Requirement 24	Support for intrinsic definitions	MUST
Requirement 25	Syntax independent from notation	MUST
Requirement 26	Allow ranges of variables, including *countable* or *uncountable*, *ordered* or *unordered*, *finite*, or *infinite*.	SHOULD
Requirement 27	Support inheritance and overriding of properties of parent classes when defining new *types*.	MUST
Requirement 28	Determining the type (`is` operator) reflects the inheritance tree.	MUST
Requirement 29	Stating axioms inside a definition as mandatory properties or as separate building blocks.	SHOULD
Requirement 30	Support of definitions of functional terms	MUST
Requirement 31	Support of definitions of predicates	MUST
Requirement 32	Clear scope of variables, in which they are declared.	MUST
Requirement 33	Support to declare variables in the defined types, including support for at least PL2	MUST
Requirement 34	Support assignment of values or expressions to variables	MUST
Requirement 35	Support to delegate the interpretation of intrinsic definitions	MUST
Requirement 37	FPL to allow identification by providing means of casting different data types to each other.	SHOULD
Requirement 38	Allow recursive linguistic constructs	MUST
Requirement 39	Allow loops	MUST
Requirement 40	Allow self-reference in definitions.	MUST
Requirement 41	Support generic types.	MUST
Requirement 42	Check the self-containment (FPL interpreter)	MUST

Table 10: FPL Cheat Sheet: All Design Decisions

6.3 All Assertions

The following table summarizes all assertions we have made in the analytic part.

Assertion 1	We assert that we can use a computer-aided process of adaptivity to check the self-containment of FPL texts.
Assertion 2	The ultimate source of semantics of a foundational theory formulated in FPL are intrinsic definitions along with their subjective interpretations of different users.
Assertion 3	We assert exact criteria for self-containment in FPL.

Table 11: FPL Cheat Sheet: All Assertions

6.4 All Limitations

In our analysis part, we have identified the following possible limitations to our high-level design of FPL:

Limitation 1	Deciding if a given proposition in PL0 is satisfiable is NP-complete.
Limitation 2	A PBM theory that is formalized in some consistent axiomatic system based on FPL, is never complete.
Limitation 3	A PBM theory able to prove **Limitation** 2 cannot prove its own consistency.
Limitation 4	Omitting intrinsic definitions to avoid the ESP (Encoding Semantics Problem) causes a VMC in trying to explicitely construct basic mathematical notions unambiguously using relative definitions.
Limitation 5	Replacing relative definitions to escape the VMC by intrinsic definitions causes the ESP, because the semantics of basic mathematical notions becomes dependent on subjective interpretations.

Table 12: FPL Cheat Sheet: All Limitations

7 Proof of Concept (PoC)

7.1 About this PoC

The main result of this specification is a syntax of FPL (Version 1.0.0) in the Extended Backus Naur Form (EBNF)[284]. You can find the current version of the FPL syntax at Github. We chose a dialect of EBNF that enabled us to use TatSu to generete a `python`-based parser of this grammar automatically. Using a simple syntactical transformation of the EBNF we also generated syntax diagrams of FPL using the tool RR. We publish the syntax diagrams of FPL also at www.bookofproofs.org where this PoC is being continued by implementing new pieces of PBM in FPL.

The presented EBNF and the generated `python` parser are a **proof of concept** (*"PoC"*) fulfilling the requirements we identified in 6.2. When we published this document, an interpreter of FPL working together with the parser was not yet available. Nevertheless, every concrete implementation of a grammar, a parser, and an interpreter creates its specific requirements making the parser and the interpreter work together as intended. If we encounter such additional requirements when presenting this PoC in the following text, they are specific to our PoC implementation and we will mark them as *"*RECOMMENDED*"* or use the verb *"*MAY*"* to clearly distinguish them from the core FPL requirements we presented in 6.2. Nevertheless, keep in mind that some of these additional requirements might still be mandatory to make our particular PoC implementation working as intended.

[284] [100]

7.2 Organizing FPL Code

7.2.1 Naming conventions of FPL

The following naming conventions for FPL are RECOMMENDED and can be fulfilled only if using FPL parsers that are case-sensitive. We chose them in our implementation of FPL and will use them in the examples of our PoC:

1. All terminal symbols in FPL are written in small letters.
2. All variable names have to start with a small letter but then can contain a combination of small letters, capitals, or underscores.
3. Identifiers of anything else in FPL (i.e., which is not a variable and not a keyword) always start with a capital, followed by a combination of small letters, capitals, or underscores.

Moreover, almost all terminal symbols of FPL are *reserved words*. Some keywords have a long and a short version but are semantically equivalent. For instance, we can write `theorem` and the short form `thm`.

7.2.2 Mixing Prose and FPL Code, Namespaces, Partial Code

As we learned in **Observation** 5, about a half of a typical PBM publication consists of some non-PBM content. It is rarely the case that a publication consists of pure mathematical definitions, theorems, and proofs. Typically, authors present us PBM bit by bit. Moreover, it is simply not feasible to understand long mathematical theories or proofs in one piece. We have to explain these contents to the readers, provide explanations, examples, or motivate what we are doing.

We have to find a way to address **Requirement** 5, i.e., to enable us to split our FPL code and mix it with other contents. At the same time, we have to fulfill **Requirement** 1 and **Requirement** 3. In these requirements, we stated that FPL has to be a *structured language* rather than any natural language and that it only has to express PBM and nothing else.

We accomplish all these requirements by giving FPL a distinguished grammar and vocabulary that are different from any natural language. At the same time, we organize the FPL code into *theories* with unique *namespaces*[285]. Namespaces have a unique ID starting with a capital later (e.g. `Fpl`), and can be divided into smaller namespaces for specific purposes whose IDs we separate by dots `"."`. For instance, `Fpl.Commons` is the namespace `Commons`, being part of a bigger namespace `Fpl`. The following are examples of valid namespaces:

[285] See root rule Namespace

`Fpl.Commons`,
`Fpl.Commons.Structures`,
`Fpl.Algebra.Structures`,
`Fpl.Geometry.Euclidean`.

Please do not write your code in the namespace `Fpl`. It is *globally reserved* for this proof of concept and its continuation on www.bookofproofs.org. All other namespaces are not reserved.

Namespaces in FPL fulfill important purposes:

1. **Re-use of code**. A theory (being defined in its own namespace) can import other theories by referencing their namespaces using the `uses` keyword[286] like this:

```
Fpl.Geometry.Euclidean
{
   uses Fpl.Commons, Fpl.Set.ZermeloFraenkel
   theory
   {
      // do stuff
   }
}
```

2. **Avoidance of identifier conficts** accross different theories. For instance, if we have two theories in which we want to declare a `FundamentalTheorem`, FPL enables us to do this, as long as we put each theory into two different namespaces, for instance `Fpl.Arithmetics` and `Fpl.Algebra`. If we want to refer exactly one of the theorems, fully specified names like

   `Fpl.Arithmetics.FundamentalTheorem` vs.
   `Fpl.Algebra.FundamentalTheorem`

   enable us to do that. We can also abbreviate long namespaces we import by writing

   `uses Fpl.Arithmetics alias Ar, Fpl.Algebra alias Al`

   Then, we can distinguish between the two fundamental theorems by using one of the aliases like `Ar.FundamentalTheorem` vs. `Al.FundamentalTheorem`.

3. **Writing partial FPL code**. We need a preprocessor helping us to address **Requirement** 5, i.e. to mix FPL code with other content without loosing the possibility to identify the bit of FPL code with the namespace to which it belogs. When

[286] see rule UsesClause

we embed FPL into other content, for instance prose text, we MAY use the following syntax:

`@Some.Namespace { // some partial FPL code }`
: Extends the code of the whole namespace `Some.Namespace`.

`@Some.Namespace!th{ // some partial FPL code }`
: Extends the code inside the theory of the namespace `Some.Namespace`.

`@Some.Namespace!inf{ // some partial FPL code }`
: Extends the code inside the inference rules of the namespace `Some.Namespace`.

`@Some.Namespace!loc{ // some partial FPL code }`
: The code in the braces extends the localization the namespace `Some.Namespace`.

`@Some.Namespace!Identifier{ // some partial FPL code }`
: The code in the braces extends the block of the identifier inside of the theory block of the namespace block `Some.Namespace`. Since every identifier in a theory is unique (even unique in the whole namespace), it is this way possible to split long mathematical proofs or definitions into many bits of FPL code we can mix with other contents.

An FPL preprocessor MAY then extract such bits of FPL code from the surrounding text and concatenate them with significant whitespaces to create regular FPL code so that it can be finally parsed and interpreted. **Note: In this PoC. we will use the above convention to split the FPL code we want to describe bit by bit!**

Of course, writing pure FPL code is also possible. By convention, FPL code MAY be stored in files with the extension `.fpl`. The support of utf8 is RECOMMENDED. The idea is to write FPL code using appropriate integrated development environments (IDEs) with capabilities to support syntax highlighting, code-completion and debugging.

We can use wildcards to import many namespaces at once. Thus `uses Fpl.Commons.*` will import any theories that have identifiers starting with `Fpl.Commons`. This is semantically different from `uses Fpl.Commons` which will import only this namespace but no subordinated namespaces.

The FPL keywords we saw already are `uses`, which imports the namespace(s) of other theories, `alias` that we can use to abbreviate long namespaces, and `theory` (shortform `th`) that initiates a theory block. In our PoC, namespaces contain exactly one theory block[287].

[287] compare rule TheoryBlock

7.2.3 Declarations and Scope of Variables

The simplest form of a variable declaration[288] in FPL has the syntax

```
<var>,...,<var> :  <type>
```

It consists of a variable (or a list of comma-separated variables), followed by a colon and type. There are only five pre-defined types in FPL:

`obj` (short for `object`): This is the parent class inherited by all user-defined types[289]. Of course, user-defined types can be used, as desired instead of `obj`.

`tpl` (short for `template`): This is a generic type, addressing **Requirement** 41. FPL supports using arbitrary many generic types by the syntax `tpl<id>` where `<id>` is an identifier starting with a capital or digit[290]. Thus, we can abstract from concrete types and use generic types instead, for example `tplFieldElem`, or `tplField` that is particularly useful when defining algebraic structures. We will present examples of this later.

`func` (short for `function`): This type denotes variables that represent functional terms. It is not to be confused with a general notion of a *"function"* that is a mathematical object we can derive in FPL from `obj`.

`pred` (short for `predicate`): This type denotes variables that represent predicates.

`ind` (short for `index`): Denotes a digit with dollar sign before it like *"$0"*, *"$1"*, etc.. It is not to be confused with natural numbers that have to be defined by the end-users in FPL. We will show later how we can define natural numbers and how we can identify digits with them. We will learn about index variables and how we can use them in FPL later.

The types `func` and `pred` make it possible to express at least PL2 (**Requirement** 11 and **Requirement** 33) in FPL. Below, we will describe definitions of functional terms and predicates in more detail.

Variables in FPL have a clear *scope* in which they are valid, addressing **Requirement** 32. The scope is always the *block* indicated by curly braces `{...}` in which we declare them. If there are more blocks nested into each other, FPL interpreters MAY interpret the variables declared in the outer block as also valid in the scope of the inner block. This also holds for variables that we declare in the *signatures* of axioms, theorem-like statements, definitions, conjectures, or inference rules in FPL. We will discuss signatures and this kind of variable declaration later in 7.3.2.

[288] The syntax of a variable declaration is defined in VariableDeclaration. We will discuss the more complex anonymous declarations later.
[289] see definitions of classes below
[290] see LongTemplateHeader

7.2.4 Whitespace, Comments

FPL syntax controls the use of whitespaces (spaces, tabulators, or new lines). There are insignificant whitespaces[291] and significant whitespaces[292] in FPL. This feature significantly improves the user-friendliness and readability of FPL since it allows us to almost completely[293] get rid of command delimiters like `";"`. Because of this feature, variable declarations like

```
p:  pred
a,b:  tpl
```

and

```
p:  pred a,b:  tpl
```

are equivalent since FPL requires significant whitespaces in these cases.

In almost any place in FPL code, in which we can insert insignificant whitespaces, we can also write comments instead[294]. One-line comments start with `//`, multi-line comments are enclosed by `/* .... */`.

7.3 Mathematical Definitions in FPL

7.3.1 Propositional and Predicate Logic in FPL

We stated in the **Requirement** 11 that we have to incorporate the syntax of predicate logic (at least PL2). In addition to the predefined types `func` and `pred`, the following keywords[295] and the related grammar rules support this requirement. The syntax of the below Boolean functions and quantors requires parameters separated by commas and parantheses, addressing **Requirement** 9:

`true`: Denotes the logical truth value $\top$.

`false`: Denotes the logical truth value $\bot$.

`not (<expr>)`: negation with a unary paramenter `<expr>`, being any predicate.

`and (<expr>,...,<expr>)`: conjunction, requires at least one up to n predicates, separated by commas.

`or (<expr>,...,<expr>)`: disjunction, requires at least one up to n predicates, separated by commas.

[291] rule IW
[292] rule SW
[293] there is only one exception of this we will discuss later in the syntax of `localization`.
[294] rule CW
[295] compare Predicate

`impl (<expr>,<expr>)`: implication, requires exactly two predicates, separated by commas.

`iif (<expr>,<expr>)`: equivalence (*"if and only if"*), requires exactly two predicates, separated by commas.

`xor (<expr>,<expr>)`: exclusive disjunction (*"either ... or ..."*), requires exactly two predicates, separated by commas.

`all <var>,...,<var>(<expr>)`: The all quantor $\forall$, requires one or more variables separated commas and a predicate in parantheses (see ParenthesisedPredicate). FPL interpreters are RECOMMENDED to check, if all variables are bound.

`ex <var>,⋯,<var>(<expr>)`: The existence quantor $\exists$. The syntax is analogous to `all`. FPL supports by `ex$n` expressing the existence of exactly $n \geq 0$ instances of the bound variables. Thus, `ex$0 <var> (<expr>)` and `not (ex <var> (<expr>))` are logically equivalent in FPL.

7.3.2 Definitions of New Predicates in FPL

Addressing **Requirement** 31, we can *define* new predicates, introducing new syntax to FPL[296]. We want to demonstrate this by the example of the equality predicate. We know from 5.4.4 that we need at least PL2 syntax to define equality in a formal language. Since equality is very common in PBM, we decided to include it in the `Fpl.Commons` namespace, which we will import in this PoC in many other examples:

```
Fpl.Commons
{
   theory
   {
      // definition of equality in FPL
      pred Equals(x,y:  tpl)
      {
         p:  pred
         a,b:  tpl

         all a, b
         (
            iif
            (
               Equals(a,b),
               all p
               (
                  iif ( p(a), p(b) )
```

[296] see DefinitionPredicate

```
                )
            )
        )
    }
}
}
```

A closer look at the rule PredicateDefinitionBlock reveals that the to-be-defined predicate is optional. It is an important feature of the FPL language because we have to be able to support intrinsic definitions (**Requirement** 24). We will present an example of an intrinsic definition of the predicate $"\in"$ later when we present how we could state the ZFC axioms of the modern set theory in FPL.

7.3.3 Overloading and Signatures

Note that the identifier `Equals` comes along with two declared and typed parameters `Equals(x,y: tpl)`. In FPL, we call such expressions *signatures*[297]. All new identifiers in FPL (i.e., those of definitions, axioms, conjectures, theorem-like building blocks, and inference rules) have signatures. Signatures fulfill multiple purposes in FPL:

1. First, signatures introduce a new identifier in a theory and its namespace. In our example, the new identifier is `Equals`. An FPL parser cannot check the uniqueness of such identifiers since uniqueness is not a concept we can specify grammatically. It is a semantical concept. Therefore, an FPL interpreter MAY accomplish this task.

2. Second, signatures enable us to *overload* identifiers in a theory and its namespace. We can define the same identifier more than once in the same namespace if it comes with different signatures.

3. Third, signatures introduce the way we are allowed to refer to the newly introduced identifiers from the surrounding FPL code. In our example, we would write `Equals(r,c)` to refer to the predicate, where `r,c` must be variables we declared in the calling code.

4. Last, signatures might declare zero, one, or more valid variables inside the block following the signature. As with identifiers, only an FPL interpreter can and MAY check that every variable is declared uniquely inside its scope and used consistently to its type. FPL parsers cannot accomplish these tasks because they deal only with the syntax of the language, not with its semantics.

[297] compare Signature

7.3.4 Generic Types

No rule in FPL would prohibit us from defining `Equals` using `obj` instead of `tpl` like this:

```
pred Equals(x,y:  obj)
{
   p:  pred
   a,b:  obj
   // do stuff
},
```

Nevertheless, we used the generic type `tpl` as parameters `Equals(x,y:  tpl)` and in the variable declaration `p:  pred a,b:  tpl`. Using generic types instead of `obj` (or user-defined types inheriting from it) are significantly different in FPL: We force the FPL interpreter to use exactly the same *domain of discourse* for the variables `a,b` inside the body of `Equals` as the types of arguments used to call it, as expected as the signature parameters `x,y`. A signature with the type `obj` would accept variables that have different types that are derived from `obj`. Accepting different types is a behavior we should avoid when implementing the Equals operator.

We want to explain this important difference by another example. Imagine, we had two user-defined types `RealNumber` and `ComplexNumber`, both inherited from the parent type `obj`. We could declare the variables like this

```
r:  RealNumber
c:  ComplexNumber
```

and then call `Equals(c,r)`. In this case, the variables `c` and `r` would differ by type. A declaration of `Equals` using the type `obj` would allow us to call `Equals(c,r)`, since `obj` would be a common *base class* of `RealNumber` and `ComplexNumber`, because it is the root of all user-defined types in FPL. On the other hand, a declaration of `Equals` using the generic type `tpl` forces an FPL interpreter to reject such a reference. We should handle equality of such variables differently, if they differ by type. In other words, FPL interpreters MAY treat generic types differently than user-defined types derived from `obj` according to this example.

We MAY implement FPL interpreters to return the value `false` when we call the signature `Equals(x,y:  tpl)` by `Equals(c,r)` even if `r,c` are still equal in the sense of *identification* about which we will talk in 5.14. Alternatively, we MAY implement FPL interpreters to reject such references (this latter implementation seems to be more preferable).

7.3.5 Definitions of New Functional Terms and Their Properties

We address **Requirement** 30 by allowing definitions of functional terms[298]. They have a different syntax from definitions of predicates. The first difference is that we introduce their signatures by the keyword `func` instead of `pred`. The second difference is that we use them to *map* their typed parameters to some mathematical object of another type. As an example, we provide the definition of a *binary operation*:

```
Fpl.Algebra.Structures
{
   uses Fpl.Commons // ...  and others

   theory
   {
      func BinOp(x,y:  tplSetElem) -> tplSetElem
      {
         optional pred IsCommutative()
         {
            a, b:  tplSetElem
            all a, b
            (
               Equals
               (
                  @self(a,b),
                  @self(b,a)
               )
            )
         }

         optional tplSetElem LeftNeutralElement()
         {
            assert IsLeftNeutralElement(self)
         }

         optional pred IsLeftNeutralElement(e:  tplSetElem)
         {
            Equals(@self(e,x), x)
         }
         // other stuff
      }
   }
}
```

[298] see DefinitionFunctionalTerm

There are many additional features of FPL we want to cover in this example.

1. Note how the generic type `tplSetElem` forces a binary operation `BinOp(x,y: tplSetElem) -> tplSetElem` to be a *closed operation*: Both, the parameters `x,y` of the binary operation and the mathematical object it maps them to have to be of the same type. As discussed above in 7.3.4, we couldn't accomplish this effect using `obj`. The generic type `tplSetElem` *fixes* the type for all places it occurs in the scope of the subsequent FPL code.

2. Our example defines some mathematical properties of binary operations. We define them *inside* the definition block of the binary operation because they only make sense in the context of a binary operation. Last but not least, we also have to fix for them the same generic type `tplSetElem` we used in the signature of `BinOp`.

3. Properties have their own type. In our example, `LeftNeutralElement` is a property of the type `tplSetElem`, i.e., it is a mathematical object by its own, while `IsLeftNeutralElement` and `IsCommutative` have the type `pred` - they are *predicative properties*, i.e., they can be either interpreted `true` or `false`.

4. All these properties are *optional*: For instance, not all binary operations might be commutative or have a left-neutral element. The keywords `opt` (short for `optional`) and `mand` (short for `mandatory`) are modifiers of properties of functional terms or classes (see below) that we can use in FPL to indicate if FPL interpreters MAY or MAY NOT assert them automatically.

5. Mandatory predicative properties (i.e., those of the type `pred`) address **Requirement** 29, because they are comparable to axioms stated inline a mathematical definition of an object. Every FPL interpreter MAY evaluate the truth value of a mandatory predicative property and attach this fixed truth value to this property of the object whenever we instantiate it.

6. The body of a predicative property is - not surprisingly, a predicate. In a predicate, we can `assert` the truth of other properties. We accomplish this by using the *assert* keyword as we see it in the above example in the body of the property `LeftNeutralElement`. There are use cases (like shown in the above example) when it makes sense to assert a property (becoming mandatory) inside an optional property. Then, it suffices to assert this optional property to assert all of the assertions it contains implicitly.

7. The above example contains two usages of the keyword `self`. It is part of the FPL language to address **Requirement** 40. We can only use the keyword `self` inside a definition. It holds for all kinds of definitions: predicates, functional terms, properties, and FPL classes. `self` always has the scope of the block in which we use it. If we put `@` before the keyword, it means we want to address the object in the outer scope of a property (if any). The FPL interpreter MAY count the number of

proceeding `@` to identify the correct scope, in which we have to apply `self` as a self-reference of an object, predicate, or functional term. In our example, `@self(e,x)` is used in the block of `IsLeftNeutralElement` and addresses the block of `BinOp`, while `self` used in the block of `LeftNeutralElement` addresses this neutral element.

As was the case with definitions of predicates, a closer look at the rule FunctionalTermDefinitionBlock and its subordinated rules reveals that the contents of a block defining a functional term might be empty to support intrinsic definitions. A real use case of such an intrinsic definition of a functional term is the successor of a natural number. We need such a notion so we can refer to it in the Peano axioms:

```
func Succ(n:  Nat) -> Nat {}
```

There is no explicit definition of this successor but a relative, stated in axioms. For instance, we can state the axiom of *complete induction* we saw in 5.4.5 in FPL like this (note like we refer to the intrinsic definition of `Succ` and use PL2 syntax inside this axiom, allowing the variable `p` to denote a predicate:

```
axiom CompleteInduction()
{
   n:  Nat
   p:  pred
   all p
   (
      impl
      (
         and ( p(0), all n ( impl ( p(n), p(Succ(n)) ) ) ),
         all n ( p(n) )
      )
   )
}
```

Because FPL controls whitespaces and supports short forms for many keywords, we could have written this axiom much shorter, for instance like this:

```
ax CompleteInduction()
{
   n:Nat p:pred
   all p(impl(and(p(0),all n(impl(p(n),p(Succ(n))))),all n(p(n))))
}
```

A computer will easily parse both versions. However, we might prefer using indentations for some parentheses or blocks to increase the readability of FPL code humans readers.

The type `Nat` is user-defined, and we need yet another possibility to define it. It is neither a predicate nor a functional term. It is what we call in FPL a class. Moreover, the symbol `0` in `p(0)` is not pre-defined in FPL[299]. We discussed already possible feasibility issues in 5.5.6 regarding the problem of introducing infinitely many new symbols for natural numbers in a formal language like FPL. We have to find a way to accomplish this task in FPL so that expressions like `p(0)` or `p(198)` become interpretable in FPL by computers. In the following examples, showing how we can define a type like `Nat` as an FPL class, we will describe how our PoC addresses these requirements.

7.3.6 Identifying New Symbols with User-Defined Types in FPL

We learned above about the in-built `index` data type and that we must not confuse it with a type `Nat` that is a user-defined data type we chose for natural numbers. We will demonstrate how end-users can to extend the grammar of FPL to identify digits $0, 1, 2, \ldots$ with a self-defined type like the type `Nat` of natural numbers introduced above.

There is an (experimental) way to introduce new grammar rules to the core grammar or FPL and include them in the parsing process. It is RECOMMENDED to use a preprocessor of the FPL parser to accomplish this task. The core FPL syntax supports the following preprocessor directives `:ext` and `:end`[300]. We present the following example:

```
Fpl.Arithmetics.Nat.Peano
{
    :ext
       extDigit = /\d+/;
    :end

    theory
    {
       // do stuff
    }
}
```

The pre-processor directive `:ext <some regex syntax> :end` enables us to include additional grammar rules in a dialect of EBNF inspired by TatSu. Because we create the FPL parser using TatSu, the idea is to extends the core FPL grammar by the user-defined grammar rules and to create an extended TatSu parser that uses those rules to parse user-defined FPL code. To avoid naming conflicts with grammar rules of the core FPL grammar that all start with a capital letter, we chose the convention that all user-defined grammar rules have the prefix `ext`, followed by some identifier starting with a capital, the rule, a colon and the keyword `end`. In our case, the grammar extension consists of a single rule named `extDigit` defined to be a regular expression accepting

[299] However, decimals are predefined in the syntax of the exists quantor like `ex[0]`
[300] compare rule ExtensionBlock

non-negative integers.

In principle, every theory can have its syntax extension. In multi-user scenarios, conflicting names of syntax extensions and conflicting syntax rules are possible, for instance, when importing third-party theories. For this reason, the FPL pre-processor MAY create a separate FPL parser per theory file. OPTIONALLY, we should create as many FPL parsers as we need to avoid name conflicts. Consequently, we RECOMMEND to create FPL interpreters being able to interpret FPL code that was parsed by separate FPL parsers, depending on the required syntax extensions.

The idea of extending the core syntax of FPL by user-defined grammar rules addresses only the problem of extending the core vocabulary and grammar of FPL. The extended FPL parser is now able to parse infinitely many symbols $0, 1, 2, \ldots$ using the regular expression `/\d+/` and tokenize them with the token `extDigit`. However, we did not yet address the problem of interpreting all these symbols as natural numbers, i.e., identify them with the user-defined type `Nat`. For this purpose, we need definitions of new classes in FPL that we will discuss in the next subsection.

7.3.7 Definitions of New Classes (Types), Constructors, Properties

An FPL class should not be confused with a *class* as a mathematical object like it is defined for instance in the class theory (compare **Example** 12 from [36]) because FPL does not prescribe or incorporate any mathematical foundations. We use the term *"class"* in the context of object-oriented programming. Definitions of FPL classes[301] have the most composite syntax as compared with the definitions of predicates and definitions of functional terms we discussed above. First of all, we need the keywords `cl` (alternatively `class`) to introduce them, followed by the inheritance from another type. The pre-defined type `object` (short form `obj`) is the base class of all user-defined classes in FPL. The signatures of classes are located in their *constructors*. We can *overload* constructors of a class with different signatures (i.e., we can have more than one constructor per class). Moreover, classes allow us to intialize and store private variables and return them as public properties. Properties may be mandatory or optional in the class. We can *override* optional properties of inherited classes making them mandatory. All these features make FPL a strictly object-oriented language, addressing **Requirement** 27.

We want to demonstrate all these features by presenting a more comprehensive example in which we will use some of the features of FPL we have already introduced. In the following example, we present how we could define natural numbers in FPL:

```
Fpl.Arithmetics.Nat.Peano
{
    :ext
```

[301] see DefinitionClass

```
   extDigit = /\d+/;
:end

uses Fpl.Commons

theory
{
   // intrinsic definitions of 0 and a successor function
   class Zero:  obj {}
   func Succ(n:  Nat) -> Nat {}

   // Peano Axioms
   axiom ZeroIsNat()
   {
      is(Zero,Nat)
   }

   axiom SuccessorExistsAndIsUnique()
   {
      n, successor:  Nat
      all n
      (
         ex[1] successor
         (
            and (NotEquals(successor,n), Equals(successor,Succ(n)))
         )
      )
   }

   axiom ZeroIsNotSuccessor()
   {
      n:  Nat
      all n
      (
         NotEquals(Zero(), Succ(n))
      )
   }

   axiom SuccessorIsInjective()
   {
      n,m:  Nat
      all n,m
      (
         impl ( Equals(Succ(n),Succ(m)), Equals(n,m) )
```

```
      )
   }

   axiom CompleteInduction()
   {
   // see previous examples
   }

   // class defining the new type Nat
   class Nat:  obj
   {
      Nat(x:  @extDigit)
      {
         case
         (
            Equals(x,0):
               self := Zero()
            Equals(x,1):
               self := Succ(Zero())
            Equals(x,2):
               self := Succ(Succ(Zero()))
            else:
               // else case addressed using a python delegate
               self := Succ(py.decrement(x))
         )
      }
   }

   // Addition of natural numbers
   func Add(n,m:  Nat)->Nat
   {
      k:  Nat
      case
      (
         Equals(m,0):
            return n
         Equals(Succ(m), k):
            return Succ(Add(n,k))
      )
   }
}
}
```

This comprehensive example demonstrates several features we haven't introduced yet:

1. Since FPL requires unique identifiers, we are independent of the relative order, in which the definitions and statements in FPL occur. We can refer to `Nat` before the actual occurence of its class definition in the FPL code. The arbitrary order of occurences in FPL is different from a fixed order needed in PBM publications written in natural languages[302]

2. We use *intrinsic definitions* of the class `Zero` and the functional term `func Succ(n: Nat) -> Nat`, demonstrating how we addressed the **Requirement** 24.

3. We list the five *Peano* axioms relating the notions `Zero`, `Nat`, and `Succ` (compare **Definition** 10) with each other logically. Note that we can do this completely without referring to set-theoretical terminology. In fact, the real-case example **Definition** 10 taken from [48] defines the *"set of natural numbers"*, while we define the *"type"* `Nat`. This is important because a natural number is semantically different from the set of all natural numbers. In a later example, we will show how to define the set of all natural numbers in FPL, based on the type `Nat`.

4. We address the problem we raised in the previous subsection of identifying infinitely many user-defined grammar extension symbols $0, 1, 2, \ldots$ with the user-defined class `Nat` of natural numbers. In [48], the author solves this problem using a remark stated in natural language (compare *"Note 1.2"* in equation 1) that is not part of the actual definition of the set of natural numbers. In the note, the author appeals to the symbols *"we are well-acquainted with"*. In FPL, we cannot do this that informal way. We have to identify infinitely many new symbols introduced by the syntax extension `extDigit` with the type `Nat` formally.

5. The examples shows us how we address the **Requirement** 23 in FPL: We define the class `Nat` *relatively*. Unlike the intrinsically defined class `Zero`, the class `Nat` has an explicit constructor. As we learned in 5.11, definitions are relative if they make use of some additional symbols that we defined outside the theory. Our theory is in the namespace `Fpl.Arithmetics.Nat.Peano`. The extension `extDigit` enhances the symbols of this namespace by a potentially infinite stock of additional symbols. The constructor with the signature `Nat(x: @extDigit)` uses one of these external symbols to construct a natural number explicitly. For this purpose, we use a `case` statement. The first three cases translate the external symbols *"0"*, *"1"*, *"2"* into the syntax of the theory `Fpl.Arithmetics.Nat.Peano`, namely the symbols `Zero` and `Succ`. We use the `else` case to cope with the potentially infinite domain of discourse for the variable `x`. We delegate handling all other cases again, using another language and its syntax (relative definition!) - the programming language python. For humans, it may suffice to refer to the symbols *"we are well-acquainted with"* like in [48]. For humans *and* computers, we can do this the way we propose in FPL, as presented in this example.

[302] [9] p. 5: *"We can only refer to a term, concept or notation **after** the appropriate definition has been encountered."*

6. Python delegation is part of the FPL core syntax[303]. It is an (experimental) way to implement a computer-friendly *model* for basic mathematical notions defined intrinsically in FPL, addressing the **Requirement** 35. We would need to implement the python delegate `decrement(x)` in such a way that it returns a semantical representation of `x` consecutive applications of the functional symbol

   `Succ(...(Succ(Zero()))...).`

   It would be demanding to program such a delegate but still feasible using the available FPL parser and FPL interpreter at hand. On the other hand, it is necessary to implement a delegate like this if we want to check expressions like `Equal(Succ(197),198)` for being true or false in FPL.

7. The constructor uses the keyword `self` to create an instance of itself representing it when the constructor is involved. We can call the constructor by assigning a variable of type `Nat` the constructor, for instance,

   ```
   Nat k
   k := Nat(198)
   ```

   At the same time, this example shows us that FPL supports assignments of variables (**Requirement** 34). The FPL interpreters MAY check if the type of the assigned expression or value is consistent with the type of the variable we declared. We cannot accomplish such a consistency check on the syntactical level alone.

8. In comparison to the functional term `Succ(n: Nat) -> Nat` which we defined intrinsically, we provided the example of `Add(n,m: Nat)->Nat` that is defined relatively in in analogy to **Example** 21. In contrast to a constructor using the keyword `self` to construct a value of an instance of its class, a functional term that is defined relatively MAY use the keyword `return` (short form `ret`) to return a value of the FPL type to which the functional term maps its parameters. The EBNF of the FPL syntax does not prohibit us from misplacing the keyword `self` in a functional term and the keyword `return` in a constructor. It also does not prohibit us from returning variables of types incompatible with the signature of the functional term or to assign values to `self` that are incompatible with the type of the class. We RECOMMEND that FPL interpreters enforce these conventions and check the type compatibility of such statements.

9. The axiom `ZeroIsNat` asserts that the intrinsically defined class `Zero` has also the class `Nat`, although it is not derived from the class. This is accomplished using the FPL keyword `is`, addressing **Requirement** 16. This feature allows *polymorphism* in the object-oriented FPL language because we can attach another type to a variable that is defined independently of its class own class. Nevertheless, such

[303] see PythonDelegate

polymorphism could cause semantical problems, for instance, if the constructors of both classes are not compatible. However, no problems can occur if we define at least one of the two classes intrinsically (like it is the case in our example). Therefore, FPL interpreters SHOULD reject polymorphism via the `is` operator, if both classes are not defined intrinsically, but using a relative definition, except they discover no semantical problems that would prohibit using polymorphism.

10. Since FPL interpreters have to reflect the inheritance tree when interpreting the `is` operator (**Requirement** 28), we could have accomplished the same effect by simply deriving the class Zero from Nat:

```
class Zero:  Nat {}
```

and omitting the axiom `ZeroIsNat`. We decided to implement the *Peano axioms* as we did to reflect their original, historical formulation better.

11. We stated the axioms *outside* the block of the class `Nat`. We could have accomplished the same semantics by stating them *inside* this block as mandatory properties:

```
 // class defining the new type Nat
class Nat:  obj
{
   Nat(x:  @extDigit)
   {
      // like above
   }

   // Peano Axioms
   mandatory pred ZeroIsNat()
   {
      // like above
   }

   mandatory pred SuccessorExistsAndIsUnique()
   {
      // like above
   }

   mandatory pred ZeroIsNotSuccessor()
   {
      // like above
   }
```

```
        mandatory pred SuccessorIsInjective()
        {
            // like above
        }

        mandatory pred CompleteInduction()
        {
            // like above
        }
    }
```

All these examples show that we can implement in FPL the same semantics in different ways, just like we can implement the same algorithm in different ways using a programming language. Moreover, the examples show how we have addressed **Requirement** 14 and **Requirement** 15 in the proof of concept of FPL.

7.4 Expressing Theorems and Proofs in FPL

After we have seen many examples of how we *define* mathematical expressions and objects in FPL, we are now well-prepared to present how to express the heart of PBM - the mathematical theorems and their proofs. Recall our example theory $\mathcal{T}$ (**Example** 4 to **Example** 7) in which we studied how the *Hilbert calculus* and the *provability relation* work together. The theory $\mathcal{T}$ covered a simple mathematical lemma about the existence of a mathematical object that is greater than two other mathematical objects based on some pre-defined inference rules and axioms. We reformulate these examples fully in FPL, calling the namespace `T`, starting with the inference rules (**Example** 6).

7.4.1 Inference Rules in FPL

Inside a namespace, we may add self-defined inference rules or use inference rules defined in the namespaces we import. We add new inference rules in a block named `inference` (short form `inf`)[304]. Inference blocks are optional in theories. We need them only when we want to enhance the FPL interpreters to check our proof's correctness formally. An inference block contains a list of *inference rules* that are valid in the theory we define it (and any theories in which you import its namespace).

Each inference rule consists of a *premise* and a *conclusion* that are introduced via the corresponding keywords `pre` (short for `premise`) and `con` (short for `conclusion`). Our FPL formulation of **Example** 6 is the following:

```
T
{
    inf
```

[304] compare RulesOfInferenceBlock

```
{
   ModusPonens()
   {
      p,q:  pred
      premise:
         and (p, impl(p,q) )
      conclusion:
         q
   }

   ProceedingResults(p:  +pred)
   {
      proceedingResult:  pred
      premise:
         loop proceedingResult in p ( assert proceedingResult )
      conclusion:
         and (p)
   }

   ExistsByExample()
   {
      p:  pred(c:  obj)
      premise:
         p(c)
      conclusion:
         ex x(p(x))
   }
}
```

Note that each inference rule has an identifier (here `ModusPonens()`, `ProceedingResults`, or `ExistsByExample()`) and declares variables to be used in the rule of type `pred`. The rule `ProceedingResults` exploits the possibility of variadic variables and loops we will discuss in more detail in 7.5.2. The signature `ProceedingResults(p:  +pred)` requires at least one argument that is a predicate. The premise of this inference rule loops over all these predicates and asserts that each of them is true. We will introduce loops and how exactly we can use them in FPL later in more detail. In the conclusion, we find a statement that also the conjunction of all these predicates is true. Note the use of the variable in `and (p)`. This is correct since `p` is a variadic variable and the syntax of `and` expects a list of predicates (compare Conjunction).

7.4.2 Axioms of Theories

We will now postulate facts about objects of the theory $\mathcal{T}$. We will do this using axioms. The four keywords `ax` (short for `axiom`), and `post` (short for `postulate`) are equivalent

and introduce mathematical axioms in FPL. Axioms are one of the eight building blocks of PBM that we implement in FPL (compare **Requirement** 4). We address **Requirement** 18 to be able to configure our axioms and inference rules the way we desire. There are no pre-defined or in-built axioms or inference rules in FPL! Thus, we fulfill also the **Requirement** 21.

Like inference rules, axioms have unique identifiers inside their namespace. In addition, any of these four keywords MAY proceed their identifiers to distinguish them from other blocks parts of the `theory` block. The axioms of **Example** 5 look like this:

```
@T!th
{
   axiom GreaterAB()
   {
      a:  A
      b:  B
      Greater(a,b)
   }

   axiom GreaterBC()
   {
      b:  B
      c:  C
      Greater(b,c)
   }

   axiom GreaterTransitive()
   {
      x,y,z:  obj
      impl
      (
         and ( Greater(x,y), Greater(y,z) ),
         Greater(x,z)
      )
   }
}
```

We choose the following approach to address **Requirement** 2 in FPL: Axioms and inference rules are meaningless arrays of symbols explaining how to prove mathematical statements accurately. Axioms in FPL are not true per se. We still need an interpretation of these symbols to be able to decide about their truth values. To provide the interpretation of symbols in the axioms, we have to explain what the types `A,B,C` and `Greater` mean. We do this via definitions. These definitions narrow the *domain of discourse* of our example theory:

```
@T!th
{
   // definitions (domain of discourse)
   class A:obj {}

   class B:obj {}

   class C:obj {}

   pred Greater(x,y:  obj) {}
}
```

Of course, intrinsic definitions are not very fruitful in helping us to interpret the classes `A,B,C`. In the absence of some fundamental mathematical theory, in which we could construct them explicitly, we define them *intrinsically* in our example theory $\mathcal{T}$. Nevertheless, we now do know more than before when we had only the axioms and the inference rules in our FPL code. We (or a computer) now can interpret variables of types `A,B,C` as mathematical objects derived from the base class `obj`. We also know that the predicate `Greater` will accept two variables of any two of these three types `A,B,C` since we derived all of them from `obj`. In contrast to these definitions, the axioms relate them to each other logically using `Greater`. However, they do not contribute to our interpretation.

Is this interpretation a *model* of our theory $\mathcal{T}$? In other words, can we interpret the axioms as true statements? Well, it depends on our *individual* interpretation. We face **Limitation** 5, because we do not have *definition relative* of the objects `A,B,C` and the predicate `Greater`. If we had such relative definitions, it would help us to confirm that we found a *model* of $\mathcal{T}$. Nevertheless, in this latter case, we would face another limitation, the **Limitation** 4. We can spin and rotate it as we like. We somehow have to address SIRP and the VMC we have discussed in detail in 5.11.

7.4.3 Theorem-like Statements and Conjectures

FPL allows distinguishing between theorems, propositions, lemmas, and corollaries, addressing **Requirement** 7, although it treats them syntactically equivalent[305]. Thus, end-users can decide when to use which type of statement. For simplicity reasons of the FPL's syntax, we decided to treat conjectures syntactically as we treat theorem-like building blocks. The only difference is the keyword introducing the identifier. The relevant keywords are:

`thm` (short for `theorem`): Proceeds a theorem identifier.

`prop` (short for `proposition`): Proceeds a proposition identifier.

[305] see TheoremLikeStatementOrConjecture and related rules)

`lem` (short for `lemma`): Proceeds a lemma identifier.

`cor` (short for `corollary`): Proceeds a corollary identifier.

`conj` (short for `conjecture`): Proceeds a conjecture identifier.

A consequence of this simplification is that an FPL parser accepts code that contains conjectures with proofs and theorem-like statements without proofs. Therefore, only an FPL interpreter is able and MAY distinguish between unproved theorem-like statements and unproved conjectures *semantically* and reject code containing proved conjectures, while it will accept code containing unproved theorem-like statements that we allow in FPL, according to **Requirement** 8.

As inference rules, theorem-like statements and conjectures consist of a premise and a conclusion. We reformulate the lemma of the **Example** 4 in FPL like this:

```
@T!th
{
   lemma Example4()
   {
      x,y,z:  obj
      pre:
      con:
         ex x ( and (Greater(x,y), Greater(x,z)) )
   }
}
```

In our simple construed case, we can leave the premise empty. For this lemma, the variable declaration itself is a premise since it limits the domain of discourse in the lemma. In natural language, we could state the lemma like this:

> *"Let y, z be mathematical objects. Then there exists a mathematical object x that is greater than y and z."*

Theorem-like statements with empty premises, i.e., in which we only need to introduce some domain of discourse, are very common in mathematics, for instance, *"For all $z \in \mathcal{C}$ we have $|z| \geq 0$"*[306]. Also, theorems stating biconditionals $\Leftrightarrow$ require only an introductory definition of the domain of discourse of the involved variables as a premise.

We cannot put the variable declaration inside the premise in FPL like it seems to occur in natural languages. In FPL, the scope of the variables is the whole lemma, including the conclusion, not only the premise. Therefore, in our PoC we have separated the syntax of variable declarations from the syntax of premises and conclusions that contain predicates. Consequently, variable declarations have to proceed the premise and conclusion in FPL, even if we have to leave the premise empty, like in our example.

[306] [38] p. 81

7.4.4 Proofs

We present the syntax of FPL in the diagram Proof. Is starts with the keyword `proof` (short form `prf`). A proof is named after the theorem-like statement that it proves, followed by a digit index. If we have more than one proof for the same theorem-like statement, we must number them consecutively to ensure unique identifiers. When mixing long proofs with prose text, we MAY split them into bits of code like this:

```
@Some.Namespace!Identifier$n{ // some partial FPL code of the proof }
```

where `Identifier` is the identifier of the theorem-like statement to be proved and `$n` is the consecutive number of the proof we are writing. In our **Example** 7, the proof is very short so that we can add it to our little theory $\mathcal{T}$ this way:

```
@T!th
{
   proof Example4$1
   {
      a:A
      b:B
      c:C
      x,y,z:  obj

      1.  /GreaterAB |- Greater(a,b)
      2.  /GreaterBC |- Greater(b,c)
      3.  /ProceedingResults(/1,/2) |- and (Greater(a,b), Greater(b,c))
      4.  /3, /GreaterTransitive
          |- impl ( and (Greater(a,b), Greater(b,c)), Greater(a,c) )
      5.  /4, /ModusPonens |- Greater(a,c)
      6.  /ProceedingResults(/5,/1) |- and (Greater(a,c), Greater(a,b))
      7.  /6, /ExistsByExample |- qed
   }
}
```

The block of the proof starts with a variable declaration, followed by a sequence of named logical arguments. The names of the arguments can be digits or digits followed by strings starting with a small letter. Such a convention provides the flexibility to structure the proof (addressing **Requirement** 13) and to reference the steps among each other. Each named argument (compare diagram ProofArgument) contains an optional argument Justification, followed by a the symbol of the *provability relation* `|-` (*"is derivable from"*), followed by the argument. Arguments are usually predicates.

The arguments referenced in the justification must lie in the same proof block or be the identifiers of the available axioms, respectively, the identifiers of the available infer-

ence rules. Our PoC does not RECOMMEND referencing the steps from proofs of other theorems that we can occasionally encounter in some PBM publications.

If the signature of the reference requires some arguments, we have to provide them. An example of such a reference is how we refer `ProceedingResults` in the justifications of the steps `3` and `6`.

Each argument is usually a predicate like it is the case in our example. However, arguments could be more complicated. For instance, we can assume them (using the keywords `assume` or `ass`). It is especially useful if the theorem to be proved has a long premise. In this case, we do not have to restate the whole predicate of the theorem's premise. We can abbreviate it by stating `assume pre` as the first logical step in our proof.

We can revoke predicates which we assumed previously, if we derive `false` from them (i.e., a contradiction). This MAY be checked by FPL interpreters. Revoking arguments requires the keyword `revoke` (short form `rev`) their original id, for instance,

```
1.  assume ...  // (some predicate)
1a.  |- ...  // derive some argument
1b.  |- ...  // derive some argument
1c.  |- false // we derived a contradiction
2.  revoke /1 // we revoke 1, since it has led us to a contradiction
3.  assume ...  (assume something else)
4.  ...  //continue the proof
```

FPL syntax neither prescribes nor forbids proofs by contradiction or using inference rules that simplify a double-negation of a predicate. Therefore, we address **Requirement** 22, by allowing both, intuitionistic and non-intuitionistic arguments.

The last derived argument MAY be the keyword `qed` (Lat. *"quod erat demonstrandum"* meaning *"what was to be shown"*) that is typical in PBM proofs. FPL interpreters MAY verify if is possible to replace it by the conclusion of the statement we claim to have proved. In our case, it would be exactly the predicate `ex x ( and (Greater(x,y), Greater(x,z)) )`.

The above example is very comprehensive. Because we justified every step, it enables us to verify every single step of the provability relation. FPL does not forbid skipping logical steps or omitting justifications in each step so that we can start each argument simply by `|-`. There are also cases in which we need only one or two inference rules or axioms to derive the statement's conclusion from its premise. In these cases, we can use the keyword `trivial` as an argument. The shortest proof accepted by the FPL grammar that you can write is:

```
1.  |- trivial
```

This way, we address **Requirement** 20 because we allow a wide range of how detailed our mathematical proofs are. Thus, teachers or mathematicians who use FPL are flexible enough to write the details they think are sufficient for their readers or audience.

7.5 Advanced Techniques

7.5.1 Identification of Semantically Incompatible Types

As we learned in 5.14, informal identification of semantically different objects, like Banecerraf's *"infinitely many ways of identifying natural numbers with pure sets"*[307], is common in mathematics and we have to find a way to express it formally in FPL, addressing **Requirement** 37. For instance, if we desired to identify natural numbers with the non-negative integers and had another class `Int` defining integers using an explicit constructor, we neither could derive the type `Int` from the type `Nat` nor could we assert their identity using the `is` operator, without causing contradictions or semantically incompatible representations.

Let us first start with observing how such an identification is done in a real PBM publication. The common way to contruct integers from natural numbers is an algebraical one. Taking [48] as an example once again, the authors use the monoid of natural numbers $(\mathbb{N}, +)$ and construct a commutative group $(G, \circ)$ containing pairs of natural numbers as its elements. Then, they define an *equivalence relation* on G as follows[308]:

$$(a, b) \sim (c, d) \Leftrightarrow a + d = b + c \qquad (a, b, c, d \in \mathbb{N}).$$

The authors then show that each of these *equivalence classes* can be respresented by a representative $[a, b]$ defined in the following way:

$$[a, b] := \begin{cases} [a - b, 0] & \text{if } a \geq b, \\ [0, b - a] & \text{if } b > a \end{cases}$$

Finally, the authors state: *"For a natural number $n \neq 0$ we set $-n := [0, n]$. Considering the previous identification and using the previous definition, we recognize that G has the form*

$$G = \{0, 1, 2, 3, \ldots\} \cup \{-1, -2, -3, \ldots\}.$$

We call $(G, \circ)$ the (additive) group of integers and denote it by $(\mathbb{Z}, +)$."

The notion *"we recognize"* appeals again to a human way of interpreting PBM expressed like this. In FPL, we have to find a formal way enabling humans *and* computers to perform such an identification. The *"right"* way to do this kind of identification in FPL is overloading the constructors in the class definitions of `Nat` or `Int`, along with using

[307] [78], 4.1
[308] [48] 72ff.

deterministic FPL interpreters (**Requirement** 10). The following example demonstrates this kind of overloading:

```
Fpl.Arithmetics.Nat.Peano{
   uses Fpl.Commons, Fpl.Arithmetics.Int
   th
   {

      class Nat:  obj
      {
         // code like above

         // second constructor overloading the first one
         Nat(x:  Int)
         {
            case
            (
               IsGreaterOrEqual(x.RightMember(), x.LeftMember()):
                  self:=x.RightMember()
               else:
                  self:=undefined
            )
         }
      }

      pred IsGreaterOrEqual(n,m:  Nat)
      {
         k:  Nat
         ex k ( Equals(n,Add(m,k)) )
      }
   }
}
```

First, we have to import another theory `Fpl.Arithmetics.Int` that would contain a class `Int`, for which we do not provide a full definition in this PoC. Assume that we have implemented the class `Int` in such a way that the type `Int` contains two mandatory properties `x.RightMember()` and `x.LeftMember()` being of the type `Nat` and representing any instance of the class `Int`. The new constructor of `Nat` compares them and returns the `RightMember` of the variable `x`, if it is greater or equal the `LeftMember`, otherwise, the constructor is `undefined` (short form `undef`), which are in-built keywords of FPL.

Note that this example would require the namespace `Fpl.Arithmetics.Int` to import `Fpl.Arithmetics.Nat.Peano` and vice versa. Such a circular dependence of theories SHOULD be allowed in FPL interpreters to fulfill the **Requirement** 37. Nevertheless, it

would be considered a *"code smell"* in other programming languages or even rejected by some compilers. The issue of such a circular reference is not caused by the design of FPL as a language. It is rather caused by our desire to identify some semantically different mathematical objects in PBM with each other. We already addressed the fact that such identifications might cause semantically incompatible representations in 5.14.

7.5.2 Lists, Arrays, Ranges and Loops in PBM

In **Requirement** 26, we identified the need for ranges in PBM that can be finite or infinite, ordered or unordered, countable or uncountable. Moreover, in **Requirement** 39, we addressed the need for expressing loops as a necessary mean to control the flow in PBM. The following features of the FPL syntax support these requirements:

1. There are two similar constructs in FPL: loops[309] and ranges[310], they are syntactically identical except the initial keywords `loop` and `range`. After one of these keywords, we need an entity that we can loop or range. In most cases it is a variable, an indexed variable, or `self`. Then a range follows with a block of statements that we put in parantheses.

2. We recommend to differentiate between the two syntactical constructs `loop` and `range` semantically. While ranges are allowed for all kinds of expressions, FPL interpreters MAY limit using loops only for expressions involving ranges that are countable and ordered. Otherwise, we could not ensure the flow control of such loops. In other words, FPL interpreters have to reject using loops in cases they cannot generate an enumerable internal representation. In those cases, the `range` keyword has to be used and accepted by the FPL interpreters instead.

3. As we saw above, FPL supports variadic variables that define *lists* of objects of a specified type [311]. We can declare variadic variables with an unspecified number of occurrences of some fixed type. Such lists are enumerable data structures built into FPL, and we should not confuse them with tuples or vectors we have to define in FPL as mathematical objects of their own via classes. As we will see later, we can access members of variadic variables using an in-built digit index that we shouldn't confuse with natural numbers either since we have to define natural numbers also by our own in FPL.

4. FPL supports *arrays* of coordinated variables. Arrays are also part of the FPL metasyntax and not mathematical objects. To access their members, we can use one or more dimensional coordinates. We intentionally use another term *"coordinates"* instead of *"indices"*, because unlike in-built digit indices of lists, coordinates are always user-defined and, as such, mathematical objects. They can have any type, including uncountable and unordered types. This allows us to express uncountable *families* of mathematical objects in FPL.

[309] compare LoopStatement
[310] compare RangeStatement
[311] compare GeneralType

We will present some examples using these syntax features of FPL. Common examples of mathematical operations that require loops are (finite or infinite) sums or products since their indices are enumerable and require ranges with values we can order.

The following FPL code snippets present three implementations of a sum of natural numbers that overload each other because of their different signatures, so that we could use them at once in our FPL code. The first signature `Sum(list: *Nat)->Nat` defines a variadic variable named `list`:

```
func Sum(list: *Nat)->Nat
{
   result: Nat
   result:=Zero()
   loop addend list
   (
      result:=Add(result, addend)
   )
   return result
}
```

The *"*"* in `list: *Nat` means that we can pass none, one, or more arguments interpretable as natural numbers to `Sum`, separated by commas, like for instance `Sum(1,n,m)`, `Sum(1,2,3,4,5)`, or `Sum()` (with no arguments). Another possible modifier is *"+"*. The signature like this `Sum(+list: Nat)->Nat` would mean that we have to pass at least one but are allowed to pass more arguments separated by commas, all of which have to be of type `Nat`.

The body of the above definition uses a variable called `result` that is initialized by the value `Zero()`, followed by a `loop` in which we range over enumerable addends within the variadic variable `list`. After the loop, we return the value of `result`. Please note that using the `Add` operation inside the loop is independent of whether `Add` itself is associative or commutative. Just like in procedural programming languages, the programmer (in our case the end-user of FPL) would have to decide or even prove if expressions like `Sum(1,2,3)` and `Sum(2,3,1)` would generate the same result.

To demonstrate how we range the key of a variadic variable and access the members of variadic variables using a digit index, we provide an alternative implementation of the above sum:

```
func Sum(list: *Nat) -> Nat
{
   i: index
   result:=Zero()
   loop i list$
```

```
    (
        result:=Add(result, list$i)
    )
    return result
}
```

The loop uses an `index`-typed variable `i` that loops over the keys of the variable `list`, denoted by `list$`. In the loop body, we add the `i`-th members denoted by `list$i`. The keys of an index variable MAY start with the digit `1`. For instance, if we want to access the third member of the variadic variable `list`, FPL interpreters MAY allow us to do this via the expression `list$3`. When this member is not available because the variadic variable has less then three members, FPL interpreters MAY return `undefined`. `list$0` is always `undefined` because the indices start with `$1` when the variable is not empty.

What is also interesting about the two examples above is the way we assign `Zero()` to the initial value of `result`. Recall that by assigning a value like this, we have to invoke the constructor of the corresponding class. Since we defined the class `Zero()` intrinsically above, it had no explicit constructor. In FPL we can still call a constructor like this, because FPL interpreters MAY provide a *default constructor* in classes defined intrinsically.

Indices like `list$3` are in-built digits into FPL that should not be confused with natural numbers that we have to define in FPL on our own, as we already demonstrated. Can we implement sums ranging over self-defined natural numbers in FPL? Yes, it is possible. However, we cannot use in-built indices, but we have to use what we call coordinates in FPL. The next example of a sum implementation requires two variables `from` and `to` of the type `Nat` that we want to use for a range of natural numbers.

```
func Sum(from, to:  Nat, arr:  Nat[from~to]) -> Nat
{
    i, result:  Nat
    result:=Zero()
    loop i [from~to]
    (
        result:=Add(result, arr[i])
    )
    return result
}
```

We declare the variable `arr` in the signature of the function to be an array of natural numbers. We expressed that type by `Nat[from~to]`. According to FPL' syntax[312], we have to put the tilde character ~ between the bounds of the range. The body of the

[312] compare TypeWithCoord

function contains a loop similar to the first sum, however, we create a range `[from~to]` in which we loop the index variable `i`. In the body of the loop, we repeat adding the coordinated array variable `arr[i]` to the result.

The FPL syntax supports excluding the bounds from a range. The notation `[!from~to]` would mean a range in which the bound `from` is excluded. Similarly, the notation `[from~to!]` would denote a range, in which we exclude the bound `to`. Finally, `[!from~to!]` denotes a range in which we exclude both bounds.

Consider yet another example of an implementation of a sum. This time, we will present how to express a power series

$$\sum_{i=k}^{\infty} a_i^n$$

of some user-defined type `Real` and for some initial index k. Suppose, that our FPL namespace contains a class defining such a type and also definitions of addition `Add` and exponentiation `Exp` of this type as well of the mathematical object `RealZero()`. An FPL representation of such a functional term could look like this:

```
func RealValuedPowerSeries(k:  Nat, arr:  Real[k~]) -> Real
{
   result:  Real
   i:  Nat
   result:=RealZero()
   loop i [k~]]
   (
      result:=Add(result, Exp(arr[i],i))
   )
   return result
}
```

In the example, the range in the signature has only a bound to the left, but no bound to the right, expressed by `Real[k~]`. FPL interpreter MAY reject calling such a function with a bound ranges like `RealValuePowerSeries(1, arr[1:25])`, because `arr[1:25]` is *"bounded from above"* but the signature expects only a range that is *"bounded from below"*, starting from some fixed coordinate `k`.

FPL syntax allows also expressing ranges (and loops) that are *"bounded from above"* which would be expressed like `Real[~k]`, or even ranges that are unbounded, which would be expressed `Real[~]`. Note that such an expression would not mean *"all"* real numbers but *"all"* real numbers from an array of real numbers having an unbound range.

Also the range `[k~]` of the variable `i` in the loop is bounded from below. In a real

programming language, a loop variable `i` running for all such values would inevitably create an infinite loop. Unlike programming languages, infinite loops are not something we want to avoid in FPL. In cases like the power series shown above, we sometimes and by intention must allow an infinite repetition of mathematical operations. Therefore, FPL interpreters cannot actually execute fpl code like it would be necessary for a code of any *"real"* programming language. Instead, an FPL interpreter MAY interpret it until it can confirm the self-containment of all the references and can create an internal semantical representation of the mathematical objects and expressions involved, including a representation of infinite loops.

The above examples involve loops, since all variables are semantically *enumerable* (i.e. they are either *finite* or *countable*) and can be ordered (e.g., we determine the order of all values in the variables `list`, `arr`, or a contruct like `[from~to]`. Enumerability is a semantical notion and cannot be discovered by FPL parsers. We need deterministic FPL interpreters to construct their own semantical representation of the variables involved so they can confirm that end-users of FPL can indeed use the `loop` keyword. If the variables are not semantically enumerable or if their domain of discourse cannot be ordered (like, for instance, in the case of complex numbers or vectors), FPL interpreters SHOULD reject the keyword `loop` and require the keyword `range` instead.

The next example demonstrates how we can create uncountable or unordered ranges for variables in FPL. Those ranges involve the keyword `range` and not the keyword `loop`. Except for this detail, the syntax remains the same. We assume that there is a user-defined type `Real` that implements the syntax and semantics of real numbers. Moreover, we assume that we want to define the *Riemann integral*

$$\int_a^b f(x)dx$$

of some real-valued function f being defined on all real numbers $\mathbb{R}$. First of all, we could define such a function using an intrinsic definition, since we want to allow *any* possible real-valued function:

```
func RealValuedFunction(x:Real) -> Real
{
}
```

Now, we need some FPL code that implements the semantics of what being "bounded" means in the case of real numbers and real-valued functions. We could accomplish this task by defining two new predicates in our example FPL code:

```
pred IsBounded(x:  Real)
{
   upperBound, lowerBound:  Real
```

```
   ex upperBound, lowerBound
   (
      and (LowerEqual(x,upperBound), LowerEqual(lowerBound,x))
   )
}

pred IsBounded(f:  RealValuedFunction)
{
   x :  Real
   all x
   (
      IsBounded(f(x))
   )
}
```

The first predicate `IsBounded` is an unary predicate expecting a variable of the type `Real` as a parameter. It makes use of some user-defined predicate `LowerEqual` (not provided in this example) that would allow us to compare two real numbers. The second predicate `IsBounded` overloads the first one because its argument `f` has the type `RealValuedFunction` instead of `Real`. When it calls the first implementation via `IsBounded(f(x))`, the type of the argument `f(x)` is again `Real`. After these preparations, we could define the Riemann integral of such functions as follows:

```
func RiemannIntegral(a,b:Real, f:  RealValuedFunction ) -> Real
{
   x, result:  Real
   assert IsBounded(f)
   result:=ZeroReal()
   range x [a∼b]
   (
      result:=Add(result, Mult(f(x), D(x)))
   )
   return result
}
```

The functional term of the Riemann integral expects three variables: `a,b` of the type `Real` and `f` of the type `RealValuedFunction`. Because it is not ensured that `f` will be a *bounded* real-valued function when we call it, we limit the scope of `f` by adding the expression `assert IsBounded(f)`. Instead of a loop, we use a range construct `range x [a∼b]` because we do not care about the exact order in which the values of `x` may be taken in the body of the range or whether or not there are only countably many of them. The remaining FPL code refers to some other user-defined mathematical object `ZeroReal()`, two binary functional terms `Add()` and `Mult()`, as well as a unary functional term `D()`. We would have to define all these notions separately to be able to refer to them. In

particular, we could use the approach proposed in [38] to define Riemann integrals using limits of *staircase functions* that can approximate the real-valued function `f` from below and above which themselves are defined as sums on a (potentially *infinite*) sum on a *partition* $a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$, as the *mesh*

$$\max(x_i - x_{i-1}), i = 1, \ldots, n$$

tends to zero. We omit all these details in this example.

The way to define a Riemann integral in FPL may appear to be cumbersome and laborious as compared to the simple formula written in usual PBM notation such as $\int_a^b f(x)dx$. However, in real PBM publications, we would have to define a lot of stuff before we can express this formula. In the cited book [38], the Riemann integral can be defined not before page 130. With this respect, FPL is not significantly more laborious as expressing PBM the conventional way.

7.5.3 Control Flow in FPL

Above, we have learned already most of the control flow statements available in FPL, the case statement `case ... else`, the assertion statement `assert`, assignments of variables using the `:=` operator, loops and ranges, the return statement `return`, and the python delegate `py`. As mentioned above, FPL code is never *"executed"* step by step like the code of a programming language. Nevertheless, all these statements, together with variable declarations and the object-oriented design, allow us to express algorithms in FPL syntax. It is particularly useful when we want to express how to calculate functions numerically that we encounter in many PBM publications dealing with *numeric mathematics*. In 5.15 we presented use cases from other mathematical disciplines, including combinatorics and number theory.

Expressing procedural constructs in FPL is possible via the grammar rule VariableSpecificationList that is part of all building blocks of FPL. This list contains variable specifications that can either contain declarations of variables or statements we mentioned above. The list is ordered according to the order in which the variable specifications have been parsed. Thus, an interpretation of such an FPL code MAY imply a sequential execution of the code, even if the FPL interpreter never executes loops in the code, for instance, because they might involve infinitely many steps in PBM. In this sense, algorithms written in FPL are comparable to algorithms expressed in some *pseudo-code*.

We present a simple example of expressing the algorithm for calculating an (infinite) *continued fraction* we discussed in **Example** 27

$$[q_0; q_1, q_2, \ldots] := q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots}}}.$$

where the q_i are some integers. In FPL, we can exploit loops, assignments and the ability to express recursion (**Requirement** 38) to define continued fractions:

```
func ContinuedFraction(k:Nat, arr:  Int[k~]) -> Real
{
   i:  Nat
   cf:  Real
   cf := Real(arr[k])
   i := Succ(k)
   return Add( cf, Div(Real(1) , ContinuedFraction(i, arr[i~])))
}
```

Since a continued fraction usually results in a real number, the functional term returns a real number although it takes an (infinite) array of integers `Int[k~]` and an index variable `k` as input parameters. After declaring the auxiliary variables `i` and `cf`, a first procedural step is to convert the integer on the k-th index of the array `arr[k]` into a real number using the assignment `cf := Real(arr[k])`. This assignment involves a constructor of real numbers that takes an integer as an input parameter and creates an equally-valued real number (see how to implement identification in FPL 7.5.1 above). In the next step, we increment the natural number `k` and assign it to the auxiliary variable `i`. Finally, we return the result of an expression involving some functional terms `Add` and `Div` we have to implement for real numbers. The `Div` contains an (infinite) recursive call of the same functional term with arguments that use the incremented index `i`.

7.5.4 Are there Set-builder and Roster notations in FPL?

In accordance with **Requirement** 21, the clear answer is *"no"*. Nevertheless, such notations, in fact, a whole ZFC set theory[313] including these notations, are quite easy to implement in FPL. This is novel since FPL is to date one of the first (if not the only existing) formal language in which we can try to accomplish something like this. We introduced the set-builder notation in 5.4.5

$$y := \{u \in x \mid \phi(u)\}$$

when talking about the syntax of PL2. Even if you do not accept ZFC as a modern foundation of mathematics, the set-builder notation is very useful and common in mathematics. It enables us to create a set y from scratch that consists of elements $u \in x$ fulfilling an arbitrary predicate $\phi(u)$, given that we have some other set x already at hand. Moreover, such a notation is very handy since it abbreviates a much more complicated predicate expressed in PL2 that follows from ZFC *axiom of separation*[314].

[313] see Appendix 9.2
[314] For more details, recall 5.4.5.

Another useful notation following immediately from the *axiom of extensionality* is the *Roster* notation[315] that allows us to create sets from scratch by simply listing their elements:

$$y := \{a_1, a_2, \ldots, a_n\}.$$

A full listing of how to implement ZFC in FPL would go far beyond the scope of this PoC, and we will only provide an excerpt. You can find a more comprehensive listing of the namespace `Fpl.Set.ZermeloFraenkel` at Github. We will introduce our example step by step, starting with an empty theory and showing which elements we need to get to the point that we can define set-builder and Roster notations we can then use in FPL.

Since set theory is fundamental, we start with *intrinsic* definitions not only for sets but also for the predicate `In` relating two sets to each other (in PBM this relation is usually denoted as $"\in"$):

```
Fpl.Set.ZermeloFraenkel
{
   uses Fpl.Commons

   th
   {
      class Set:  obj {}
      pred In(x,y:  Set) {}
   }
}
```

First, we import `Fpl.Commons`. This namespace implements, among others, some common inference rules, and the predicate `Equals` in PL2 (see 7.3.2). We will need this predicate in ZFC set theory. Moreover, we derive `Set` from `obj`. The class has no explicit constructor. The signature `pred In(x,y:  Set)` of the intrinsic definition of `In` requires that whenever we relate two mathematical objects via `In`, they must have the type `Set` (or be derived from that type). Therefore, in our implementation of ZFC, only sets can be elements of other sets. Thus, we decide to implement the *model* of `In`[316]. If we decided to implement a *non-transitive model*, we would have used the base type `obj` or a generic type `tpl` like this `pred In(x:  obj, y:  Set)`. Such a signature of `In` would allow elements of sets that are other mathematical objects than sets. This simple detail demonstrates impressively how FPL allows us to implement many possible interpretations of $"\in"$ explicitely for the end-users of FPL, both humans and computers.

As a first axiom, we want to express the ZFC *axiom of existence*. It states that there is such thing called an *"empty set"*. Because both expressing the empty set and checking

[315] [25], p. 76
[316] compare 5.11.5

if a given set is empty are common notions in ZFC, we have a rationale to save duplicate code by defining a predicate `IsEmpty` before we express the axiom in which we will only refer to this predicate. We will make it unary, expecting one parameter being a set:

```
@Fpl.Set.ZermeloFraenkel!th {
   pred IsEmpty(x:  Set)
   {
      y:  Set
      all y
      (
         not ( In(y, x) )
      )
   }

   axiom EmptySetExists()
   {
      x:  Set
      ex x
      (
         IsEmpty(x)
      )
   }
}
```

Declaring variables denoting empty sets is already possible, for instance by writing `x: Set assert IsEmpty(x)`. However, we want to abbreviate it a little more by providing a separate class containing an own contructor. Note how we can use the `self` keyword to assert the emptiness of the set to be created:

```
@Fpl.Set.ZermeloFraenkel!th {
   class EmptySet:  Set
   {
      EmptySet()
      {
         assert IsEmpty(self)
      }
   }
}
```

Next, we want to abbreviate the relation between a subset of a superset. We want to

state that a set is a *"subset"* of another *"superset"* if for all sets u we have the implication

$$u \in \texttt{subset} \Rightarrow u \in \texttt{superset}.$$

We can do this by introducing a new predicate `IsSubset`. Moreover, we provide another class to abbreviate defining subsets of other sets:

```
@Fpl.Set.ZermeloFraenkel!th
{

   pred IsSubset(subset,superset:  Set)
   {
      u:  Set
      all u ( impl (In(u, subset), In(u, superset)) )
   }

   class Subset:  Set
   {
      Subset(superSet:  Set)
      {
         assert IsSubset(self, superSet)
      }
   }
}
```

Note that the constructor of `Subset` is unary, expecting a parameter that is a set. With this abbreviation, it becomes very easy to declare a variable denoting a subset of another set. For instance, the sentence *"Let x be a set and let $y \subseteq x$."* becomes that simple in FPL: `x,y:  Set y:=Subset(x)`. Of course, we could also put a new line between the variable declaration `x,y:  Set` and the assignment `y:=Subset(x)` to increase the readability of the code. Nevertheless, this shows that FPL lacks nothing that we need to express PBM in a short, concise, and rigorous way.

It becomes now very simple to express the next ZFC axiom: the *axiom of extensionality*. We now need the `Equals` predicate we imported from the namespace `uses Fpl.Commons`:

```
@Fpl.Set.ZermeloFraenkel!th
{
   axiom Extensionality()
   {
      x,y:  Set
      all x,y
```

```
      (
         impl
         (
            and ( IsSubset(x,y), IsSubset(y,x) ), Equals(x,y)
         )
      )
   }
}
```

The axiom `Extensionality` states that two sets are equal if they have the same elements. It motivates the Roster notation that allows enumerating set elements to create a new set. We add this functionality to our ZFC theory by exploiting variadic variables:

```
@Fpl.Set.ZermeloFraenkel!th
{
   class SetRoster:  Set
   {
      SetRoster(listOfSets:  *Set)
      {
         elem:  Set
         loop elem listOfSets
         (
            assert In(elem, self)
         )
      }
   }
}
```

The signature of the constructor `SetRoster(listOfSets:  *Set)` requires zero, one, or more sets separated by commas. We loop this variadic list and assert that the arguments of the list are elements of the set we are creating. If the list has zero arguments, the FPL interpreter MAY skip the loop. The created set will be empty in the sense that it has no elements at all. It would be very simple to verify the truth of the predicate `Equals(SetRoster(), EmptySet())` because the constructor of `EmptySet` creates a set for which we assert that no other set is its element. The constructor of `SetRoster` called with an empty list `()` creates a set for which we never assert that some other set is its element. Thus, by axiom `Extensionality`, both sets are equal. We leave writing this simple mathematical proof in FPL as an exercise for the readers.

Using the Roster notation, it becomes, for instance, very easy to define a functional term that creates a *singleton* set $\{x\}$ out of a given set x:

```
@Fpl.Set.ZermeloFraenkel!th
{
```

```
   func Singleton(x:  Set) -> Set
   {
      return SetRoster(x)
   }
}
```

We are rapidly approaching defining the set-builder notation in FPL. First of all, we define the ZFC *axiom of separation*:

```
@Fpl.Set.ZermeloFraenkel!th
{
   axiom SchemaSeparation()
   {
      p:  pred
      x,y:  Set
      all p,x (ex y ( Equals(y,SetBuilder(x,p)) ) )
   }
}
```

This ZFC axiom is a whole schema of *infinitely* many axioms. We state that for all predicates `p` and all sets `x` there is a set `y` equal to a set we create by the constructor `SetBuilder(x,p)`. The signature of this constructor shall accept a set as a first parameter and a predicate binding its elements as a second parameter. In this respect, the second parameter is not an arbitrary predicate. In FPL, we can easily define a signature defining not only that it expects a predicate variable `p` but also how this variable should be structured before we can use it to call the constructor:

```
@Fpl.Set.ZermeloFraenkel!th
{
   class SetBuilder:  Set
   {
      SetBuilder(x:  Set, p:  pred(u:  Set, o:  *obj))
      {
         assert
            all u
            {
               iif ( In(u,self), and ( In(u,x), p(u,o) ) )
            }
      }
   }
}
```

The constructor contains an assertion that all variables `u` denoting a set (see implicit declaration in the signature) are elements of `self` if and only if they are elements of `x`

and fulfill the predicate `p(u,o)` where `o` is a variadic variable that can contain zero, one, or more variables of any type bound by the predicate `p`.

We are now able to use this constructor to create new sets by passing a given set `x` and any predicate `p` about its elements. For instance, if we had defined the predicate `IsGreater` for two natural numbers, we could define the set of all natural numbers that are greater than 100 in FPL like this:

```
class N100:Set{N100(){n:Nat self:=SetBuilder(SetNat(),IsGreater(n,100)) }}
```

which is the FPL version of

> *"Let N_{100} be the set defined by $N_{100} := \{n \in \mathbb{N} \mid n > 100\}$."*

7.5.5 Compound Parameters and Implicit Properties

In 5.13.9 we discussed the need to be able to define compound mathematical objects with parameters that have their implicit properties. We formulated this in the **Requirement 36**. In 7.5.4, we already used this FPL feature when defining the constructor of the class `SetBuilder` - its signature not only declared a predicate `p` but also its *arity* and the types of the predicate's parameters.

Situations like these are common PBM and FPL's syntax addresses the need to define types implicitly via the syntax rules AnonymousDeclaration and AnonymousSignature. For instance, these rules allow us to ensure that when we pass a binary operation to the constructor of an algebraic structure, we also can enforce the type of variables on which this binary operation will operate. Consider the following example:

```
Fpl.Algebra.Structures
{
   uses Fpl.Commons, Fpl.Set.ZermeloFraenkel
   th
   {
      func Composition(operands:  *tplSetElem) -> tplSetElem
      {
         // defined intrinsically
      }

      class AlgebraicStructure:  obj
      {
         myCarrierSet:  Set
         myOps:  +Composition(*tplSetElem)

         AlgebraicStructure(x:  tplSet, ops:  +Composition(*tplSetElem))
```

```
        {
            a:  obj
            myOps := ops
            myCarrierSet:= x
            assert and ( is(tplSet, Set), is(tplSetElem, Set) )
            assert all a ( impl (is(a,tplSetElem), In(a,myCarrierSet) ) )
        }

        mandatory Set CarrierSet()
        {
            self:=myCarrierSet
        }

        mandatory func NthOp(n:  index) -> Composition(*tplSetElem)
        {
            return myOps$n
        }

    }
  }
}
```

The above example defines what in PBM is called an *algebraic structrure*. In mathematics, algebraic structures are typically expressed as tuples like $(\mathbb{N}, +)$, $(\mathbb{Q}, +, \cdot)$, $(B, \wedge, \vee, \neg, T, F)$, $(S, \cdot, ^{-1})$, etc. The tuples start with some set, called the *carrier set* of the algebraic structure, followed by at least one *composition* that is an operation on the elements of the carrier set. Each composition is variadic. For instance, in the so-called *Boolean algebra*[317] $(\mathbb{B}, \wedge, \vee, \neg, T, F)$ the compositions $"\wedge"$ and $"\vee"$ are binary, the composition $"\neg"$ is unary, and the constants $"T"$ and $"F"$ can be defined as 0-ary compositions.

We start with an intrinsic definition of a functional term `Composition`, taking zero, one, or more arguments of some generic type we called `tplSetElem` and mapping them to a new object of the same generic type. Only this definition ensures that every composition will be *closed* regarding their type. Next, we derive the class `AlgebraicStructure` from the FPL base class `obj` and declare two variables `myCarrierSet` and `myOps` whose validity is limited to the scope of the class block. Whenever we instantiate a variable with the user-defined type `AlgebraicStructure`, we will call its constructor with a list of arguments. The first argument will be (any) carrier set, and the next argument will be a list of at least one composition. Each of these compositions might be of any arity. After calling the constructor, we want to access these compositions and the carrier set in the context of the algebraic structure we have just created. This is possible via *"remembering"* the arguments that we used when calling the constructor in two inter-

[317] see Appendix 9.1

nal variables `myCarrierSet` and `myOps` and implementing two properties `mandatory Set CarrierSet()` and `mandatory func NthOp(n: index) -> Composition(*tplSetElem)` that return these internal values again. Since the number of compositions is dynamic, we can dynamically access the n-th composition.

Recall that we cannot access internal variables of a class from outside since the scope of these variables is limited to its block. In this sense, FPL variables declared inside a block are always *private* in this block. However, properties in FPL are always *public*. We know already that we can use the modifier `mandatory` to ensure that every new class that we derive from the class `AlgebraicStructure` will inherit these properties. It will allow us to access the compositions and the carrier sets of any such derived class without having to re-implement these properties again.

The two additional assertions inside the constructor of `AlgebraicStructure` need our special attention. They have to do with how we prefer to interpret the set-theoretical $"\in"$ operation. In this example, we have imported, among others, the namespace `Fpl.Set. ZermeloFraenkel` containing the definitions of the external identifiers `Set` respectively `In`, denoting mathematical sets respectively the $"\in"$ relation between two sets. As we presented in 5.13.9, we decided to implement a transitive model of $"\in"$ which means that only sets can be elements of other sets. However, the constructor of `AlgebraicStructure` uses the generic types `tplSet` and `tplSetElem`. These both types could be something else than sets. For this reason, we need two assertions in the constructor of `AlgebraicStructure`:

1. First, we assert that the generic types `tplSet` and `tplSetElem` are sets. We do this by exploiting the polymorphism capabilities of the FPL operator `is`. Such an assertion is always possible unless we call the constructor with some types that are inconsistent with the ZFC axioms. When interpreting such an assertion, FPL interpreters MAY check if the domains of discourse of the types `Set` and the domains of discourses of the types used to call the constructor of `AlgebraicStructure` are semantically compatible. Only in this case, the assertion that both `tplSet` and `tplSetElem` are sets MAY be accepted, otherwise it MAY be rejected by the FPL interpreters.

2. Next, we assert that all mathematical objects of type `tplSetElem` for which we use the variable `a` are elements of the carrier set of our algebraic structure. Together with the definition of the functional term `Composition` producing an object of the type `tplSetElem`, this is a convenient way in FPL of making sure that each of the compositions used to create an instance of `AlgebraicStructure` are *closed* with respect to its carrier set.

In many mathematical sources, for instance, in [37], we can find definitions of algebraic structures and compositions expressed conventionally as a mix of symbolic PBM notation together with a natural language. The definitions together with explanations of

all symbols involved take there about one full page[318]. The above FPL example is not longer. Moreover, in some ways, it is superior to the conventional way:

1. When defining compositions, the source states:

   "For a set M and $n \in \mathbb{N}$ we call a map

   $$\tau : \begin{cases} M^n & \to M \\ (x_1, x_2, \ldots, x_n) & \to \tau(x_1, x_2, \ldots, x_n) \end{cases}$$

   *an **n-ary composition** on M. For 1-ary, 2-ary, or 3-ary we say also **unary**, **binary**, or **ternary**. We haven't excluded the case $n = 0$ in this definition. However, to interpret this definition for $n = 0$, we need some minimum of reflection. [...]"*[319]

   The text that comes after this quotation explains how this definition still covers a 0-ary composition, based on some interpretation of M^0 as a map of the empty set $\emptyset$ to M and why this map is unique. In FPL, we can circumvent such complications elegantly because we can use variadic variables and store them as implicit properties in mathematical objects.

2. In-built variadic variables and `index`-typed variables to access their members in FPL eliminate the need to refer to other mathematical concepts like tuples and natural numbers or to set-theoretical concepts like the *Cartesian product* M^n.

3. Moreover, this is how the above publication [37] adds semantics to the formulas addressing the possibility of the closure of the algebraic structure with respect to the compositions:

   *"If the map τ is partial then we say that the composition is **partial**. If τ is not partial, we can emphasize it by formulating that τ is **defined everywhere**. More specifically, we call such compositions **inner compositions** [...] If we define the composion τ only on a subset $A \subseteq M$ and there is an $(x_1, x_2, \ldots, x_n) \in A$ such that $\tau(x_1, x_2, \ldots, x_n) \notin A$, we call A to be **not closed** with respect to τ."*[320]

   In our FPL example, we can make these semantics explicit by our two assertions inside the constructor. We do not need to explain in natural language that the definition of `AlgebraicStructure` we wrote involves inner compositions only. The explicitness of the two assertions in the constructor increases the strictness of the language to define a mathematical object. In contrast, the cited example expressed in a natural language requires that readers cautiously *"keep in mind"* which interpretation is allowed - one with inner or one with outer compositions.

[318] p. 142, formulated in German
[319] [37] pp. 141-142, transl. from German
[320] [37] p. 142, transl. from German

4. We could very easily have made our definition more general to cover cases in which the algebraic structure is not closed with respect to its compositions (i.e., allows outer compositions). To define such a generalized algebraic structure in FPL, it would have sufficed to remove the two assertions inside the constructor we presented above. This way, we would make it *explicit* for humans and computers that we do not exclude outer compositions. Nevertheless, we could still use such a generalized definition to create instances of algebraic structures in which we require the compositions to be inner compositions only. To do this, we would have to assert the closure of the compositions and the carrier set before we use them as arguments to call the constructor of `AlgebraicStructure`. Since the constructor stores them in the internal variable `myOps`, it would also store all properties of the compositions, including the asserted ones.

5. Omitting the two assertions inside the constructor would make the definition not only more general but also allow us to get rid of any reference to the namespace `Fpl.Set.ZermeloFraenkel` in which we import set theory in our example.

7.5.6 Expressing Actual Infinity in FPL

We talked about *actual infinity* in 5.8 and 5.9. Regardless of the question of intuistionists whether or not actual infinity exists, we will demonstrate how to express it in FPL. As an example, we will define the object $\mathbb{N}$, i.e., the *set of all natural numbers* that is not to be confused with the `type Nat` we defined above. Defining such a set is simple in FPL if we import the namespace `Fpl.Set.ZermeloFraenkel` that contains all the abbreviations of set-theory we need:

```
@Fpl.Arithmetics.Nat.Peano
{
   uses Fpl.Commons, Fpl.Set.ZermeloFraenkel
}
```

We can now use the type `Set` to derive the new class from it:

```
@Fpl.Arithmetics.Nat.Peano!th {
   class SetNat:  Set
   {
      SetNat()
      {
         n:  Nat
         assert
            all n (and ( is(n, Set), In(n, self) ))
      }
   }
}
```

Recall that we decided to implement the *transitive model* of the `In` relation in 7.5.4, so `SetNat` can only have other sets as elements. On the other hand, recall that we derived `Nat` from `obj` and not from `Set`. Therefore, we cannot take for granted that if we declare variables by `n: Nat`, they will denote some sets. To enable us to make them elements of the set `SetNat`, we *assert* that they *are* sets and (as such) are elements of the newly created object `SetNat`. After calling such a constructor, the object `SetNat` is (because of the ZFC axiom of extensionality) *"the"* set of *all* natural numbers. Moreover, we can deal with it like with all other sets.

Of course, such assertions may cause semantical problems. For instance, we might write a new constructor that overloads the first one and asserts some contradiction to the first one like this:

```
@Fpl.Arithmetics.Nat.Peano!SetNat {
   SetNat(x: Set)
   {
      n: Nat
      assert
         all n
         (
            impl (In(n,x), not(In(n,self)))
         )
      }
   }
}
```

If we call the second constructor with a set `x` containing some natural numbers, then we assert that all these natural numbers are *not* elements of `SetNat`. By calling the first constructor, we assert that *all* natural numbers without any exception are elements of `SetNat`. Thus, both constructors involve a contradiction.

Please note that such contradictions are not caused by the specifics of FPL. Their roots are rather in PBM itself. We can easily create such a contradiction expressing the above FPL code conventionally like this:

"Let n be a natural number and let x be a set. If $n \in x$ then $n \notin \mathbb{N}$".

If you think that this is not possible to state this in PBM, because the notion *"Let n be a natural number"* is *"equivalent"* with *"$n \in \mathbb{N}$"*, recall that the *Peano axioms* do not define the set of all natural numbers $\mathbb{N}$! They only define what it means for a meaningless variable n *to be* a natural number. We can define the set $\mathbb{N}$ only *after* we already know what *"being a natural number"* means, not before! Unfortunately, set theory is so omnipresent in modern PBM that most mathematicians forget that expressions like $n \in \mathbb{N}$ never characterize the variable n *sufficiently*, only *necessarily*! Nothing in mathematics prevents us

from defining a set x that contains *some* natural numbers and, independently from that, to define a set $\mathbb{N}$ containing *all* of them. A contradiction occurs not during the definition process but only if we relate both symbols x and $\mathbb{N}$ logically in a relation like stated above.

So how can we avoid contradictions like this one in FPL? Obviously, an FPL parser cannot discover them. An FPL interpreter MAY check if assertions we make in all constructors of the same class (and even all classes) are consistent. It is a novel extension of the notion of consistency. In predicate logic, only predicates being part of building blocks that are subject to the provability relation "$\vdash$", namely mathematical axioms, theorems, and their proofs, can be consistent[321]. We should extend the notion of consistency to predicates that are part of mathematical definitions to avoid semantical problems like the problem described above, even though we cannot express mathematical definitions in predicate logic[322] from which the term *"consistency"* originates.

7.5.7 Functions and Relations in FPL

We learned already in 7.3.5 how we can define functional terms and predicates using the `func` and `pred` keywords. Nevertheless, we shall not confuse functional terms with the mathematical notions *"functions"* and *"relations"* that we can define in FPL as mathematical objects of their own.

Since functions and relations are fundamental mathematical concepts in PBM, we want to show in our PoC how we can define them in FPL. We will approach this task the modern and standard way of defining relations as subsets of some *Cartesian product* and functions as special cases of binary relations that are *left-total* and *right-unique*. Similarly, we will demonstrate how we can define *equivalence relations* in FPL as special cases of binary relations that are *reflexive*, *symmetric*, and *transitive*.

```
Fpl.Commons.Structures
{
   uses Fpl.Commons, Fpl.Set.ZermeloFraenkel, Fpl.Arithmetics.Nat.Peano
   th
   {
      class Tuple :  tpl[Nat~Nat]
      {
         myLength:  Nat

         Tuple(listOfTpl:  +tpl)
         {
            elem:  tpl
            i:  Nat
            i:= Zero()
```

[321] compare, for instance, [57] 28ff.
[322] compare 5.5.5

```
            loop elem listOfTpl
            (
               i:=Succ(i)
               self[i]:=elem
            )
            myLength:=i
         }

         Tuple(length:Nat, x:  tpl[1~length])
         {
            self:=x
            myLength:=length
         }

         mandatory Nat Length()
         {
            return myLength
         }
   }
}
```

We put our implementation into a new namespace `Fpl.Commons.Structures` and use three other namespaces because we will need natural numbers, sets, and the `Equals` predicate. Moreover, we start with implementing a class `Tuple` that we derive from the array of some objects with the same (generic) type whose coordinates range between two natural numbers (`tpl[Nat~Nat]`). The class implements two different constructors, one expecting a variadic list of objects with the corresponding generic type (`+tpl`), the second expecting an array `arr` and a natural number `length` being the right bound of this array. In both constructors we initialize also the local natural number variable `myLength` that we make accessible by the (public) mandatory property `Nat Length()`.

The next step is to define the Cartesian product. Usually, a Cartesian product is defined in mathematical publications like this[323]:

*"For any sets $A_1, \ldots, A_n$ the **Cartesian product** is defined by"*

$$A^n := A_1 \times A_2 \times \cdots \times A_n := \{(x_1, \ldots, x_n) \mid x_i \in A_i, 1 \leq i \leq n\}.$$

This kind of definition is semantically problematic in the sense that it uses the set-builder notation. Recall that the set-builder notation is axiomatically justified by Zermelo's *axiom of separation* that he invented to circumvent Russell's paradox[324]. To define a Cartesian product A^n using the set-builder notation, we already need an existing set u

[323] compare, for instance, [37], p. 34
[324] compare 5.8

of tuples $(x_1, \ldots, x_n)$ from which we could separate the set A^n as a subset fulfilling the additional property $x_i \in A_i, 1 \leq i \leq n$. However, the set A^n is already *"maximal"* in the sense that no other set exists from which we could separate A^n. Thus, this definition is *impredicative*, it contains a self-reference of A^n from which we want to separate it. This observation is novel. For this reason, we take in FPL another approach to define a Cartesian product:

```
@Fpl.Commons.Structures!th
{
   class CartesianProduct:  Set[Nat~Nat]
   {
      myLength:  Nat

      CartesianProduct(setList:  +Set)
      {
         setItem:  Set
         i:  Nat
         i:= Zero()
         loop setItem setList
         (
            i:=Succ(i)
            self[i]:=setItem
         )
         myLength:=i
      }

      CartesianProduct(n:  Nat, setArray:  Set[1~n])
      {
      self:=setArray
         myLength:=n
      }

      mandatory Nat Length()
      {
         return myLength
      }
   }
}
```

Like for the class `Tuple`, we provide two different constructors of `CartesianProduct`, one that allows a variadic list of sets and one that allows an array of sets with a fixed length `n`. The constructors only store all sets that we use when we call the constructors. They still make no assumptions whatsoever about the structure of these sets. Obviously, we have to narrow the possibilities, which allow us to define new Cartesian products as

follows:

1. Since we derive the class `CartesianProduct` from an array of sets `Set[Nat~Nat]`, we want this class to be a `Set` of its own. We obviously need an additional assertion `is(self,Set)`.
2. We want to state that if `self` has some set element, let's call it `tupleElem`, then this element will necessarily have the type `Tuple[1~myLength]`. In addition, for all natural numbers `i` in the range `[1~myLength]` we would need to assert `In(tupleElem[i],self[i])`.
3. Note that the objects `self[i]` already have the type `Set`, by construction.

We can accomplish this by complementing our class with a mandatory predicate like this (let us call it `AllTuplesIn`):

```
@Fpl.Commons.Structures!CartesianProduct
{
   mandatory pred AllTuplesIn()
   {
      tupleElem:  Set
      i:  Nat
      and
      (
         is(@self,Set),
         all tupleElem
         (
            impl
            (
               In(tupleElem,@self),
               and
               (
                  is(tupleElem,Tuple[1~myLength]),
                  assert In(tupleElem[i],@self[i])
               )
            )
         )
      )
   }
}
```

This code completes our definition of a Cartesian product that circumvents a problematic impredicative definition using the set-builder notation that is common in many PBM publications.

Now, it is easy to implement mathematical $n-ary$ relations as a new class. It is simply a subset of a corresponding Cartesian product. We provide a constructor of a relation

and a mandatory property being a natural number that will allow us to access the arity of the relation whenever we create a new instance of it:

```
@Fpl.Commons.Structures!th
{
   class Relation:  Set
   {
      myArity:  Nat

      Relation(arity:  Nat, setList:  +Set)
      {
         cartProd :  CartesianProduct
         cartProd := CartesianProduct(setList)
         myArity := cartProd.Length()
         self := Subset(cartProd)
      }

      mandatory Nat Arity()
      {
         return myArity
      }
   }
}
```

We are continuing our PoC example. Our final goal was to show how to define *equivalence relations* and functions in FPL. For this purpose, we need a special case of relation: a binary relation (i.e., with the arity 2). We will use it as a base class from which we will finally derive functions and equivalence relations. The following base class `BinaryRelation` with a fixed arity 2 will also allow us to introduce two mandatory properties with the type `Set` that we will call the *"domain"* and the *"codomain"* of the binary relation:

```
@Fpl.Commons.Structures!th
{
   class BinaryRelation:  Relation
   {
      myDomain, myCodomain:  Set

      BinaryRelation(x,y:  Set)
      {
         self:=Relation(2,x,y)
         myDomain:=x
         myCodomain:=y
      }
```

```
      mandatory Set Domain()
      {
         return myDomain
      }

      mandatory Set CoDomain()
      {
         return myCodomain
      }
   }
}
```

Now, calling `BinaryRelation(x,y)` only creates a new binary relation with the domain `x` and the codomain `y`, i.e. a subset of the cartesian product $R \subseteq x \times y$. We need a predicate about its elements, so we can check if a given tuple is its element like in $(u, v) \in R \subseteq x \times y$. We do this by an additional predicate defined like this:

```
@Fpl.Commons.Structures!th
{
   pred AreRelated(u,v:  Set, r:  BinaryRelation)
   {
      tuple:  Tuple[1~2]
      tuple := Tuple(u,v)
      assert
         self := Relation(2,x,y)
         and
         (
            In(tuple, r),
            In(u, r.Domain()),
            In(v, r.Codomain())
         )
   }
}
```

We can call this predicate with three parameters, two of which are sets `u,v` and the last of which is a binary relation `r`. The predicate creates a tuple `Tuple(u,v)` and returns the truth value of a conjunction of three predicates: The tuple must be an element of the binary relation `r` and its arguments `u`, respectively `v` must be elements of the domain of `r`, respectively the codomain of `r`.

This new predicate allows us to extend the class `BinaryRelation` by multiple optional properties, which will allow us to build the most important types of binary relations known in modern mathematics, including functions, *equivalence relations*, and order re-

lations. There are at least 12 such properties. For instance, the properties *left-total* and *right-unique* will allow us to define *functions*. The properties *reflexive*, *symmetric*, and *transitive* will allow us to define *equivalence relations*. Some other of the 12 properties allow mathematicians to define *order relations*[325]. We provide only two of these properties demonstrating how to accomplish this in FPL:

```
@Fpl.Commons.Structures!BinaryRelation
{
   optional pred LeftTotal()
   {
      v,w:  Set
      all v
      (
         ex w
         (
            and
            (
               In(v, Domain()),
               In(w, CoDomain()),
               AreRelated(v, w, @self)
            )
         )
      )

   optional pred RightUnique()
   {
      v,w1,w2:  Set
      all v,w1,w2
      (
         impl
         (
            and
            (
               In(v, Domain()),
               In(w1, CoDomain()),
               In(w2, CoDomain()),
               AreRelated(v, w1, @self),
               AreRelated(v, w2, @self)
            ),
            Equals(w1, w2)
         )
      )
```

[325]For more details, see, for instance, in [42] p. 31.

```
   }
}
```

We finally show how we then could define a function in FPL. The two assertions in the constructor make the two properties `LeftTotal` and `RightUnique` mandatory in the class `Function` while they were only optional in the class `BinaryRelation`. Nevertheless, the class `Function` inherits the mantatory properties `Domain` and `Codomain` from `BinaryRelation` and `Arity` from `Relation` so we do not need to reimplement them:

```
@Fpl.Commons.Structures!th
{
   class Function:  BinaryRelation
   {
      Function(x,y:  Set)
      {
         self:=BinaryRelation(x, y)
         assert self.LeftTotal()
         assert self.RightUnique()
      }
   }
}
```

Using the same approach, we could easily define an equivalence relation in FPL like this:

```
@Fpl.Commons.Structures!th
{
   class EquivalenceRelation:  BinaryRelation
   {
      EquivalenceRelation(x,y:  Set)
      {
         self:=BinaryRelation(x, y)
         assert self.Reflexive()
         assert self.Symmetric()
         assert self.Transitive()
      }
   }
}
```

7.6 Flexible Notation and Localization

As we saw above in multiple examples of our PoC, FPL's core syntax replaces common symbols like `+` by identifiers like `Add`. Instead of writing `x + y` we write `Add(x,y)`, writing

the operand in a kind of a parenthesized *prefix notation*. The original prefix notation (also called the *Polish notation*) was invented by the Polish logician Jan Łukasiewicz (1878 - 1956) to circumvent using parentheses, for instance, by writing `+ x y` instead of `(x + y)`. Nevertheless, we will call `Add(x,y)` "prefixed notation". There are many reasons we chose this notation in FPL:

1. We have to disambiguate the syntax of mathematical operations (**Requirement** 9) that we want to define by ourselves (**Requirement** 14), while no mathematical operations and axiomatic systems are built into FPL's core syntax (**Requirement** 21).

2. We required that FPL's syntax must be independent of mathematical notation because the latter is dependent on individual authors, trends, and transient conventions (**Requirement** 25). Our prefixed notation addresses this requirement. Moreover, because mathematical notation might change over time, we might want to replace an old notation with a more modern one without rewriting our FPL code (compare **Requirement** 17).

3. Also modern PBM notation uses prefixed notation whenever we write functions, e.g., $f(x)$ or $g(x, y)$. Also, many programming languages use this notation when calling methods or functions. Therefore, it is not unusual for many end-users.

4. Prefixed notation increases the readability of FPL code because it reduces the number of parentheses to be used to a minimum.

5. Prefixed notation allows indentation without affecting the syntax. For example, by indenting the arguments of a prefixed operation, we increase the readability of complex logical formulas that would otherwise become very hard to read and understand by humans.

Despite these reasons, many of us may still prefer infix notation for some operations (like writing `x + y` rather than `Add(x,y)` or want to exploit the possibilities of LaTeX when writing mathematical publications. On the other hand, we may want to exploit the computer power of FPL parsers and interpreters to write PBM contents that are semantically clear and consistent. We stated exactly these requirements in **Requirement** 6 and **Requirement** 19.

To address all these requirements at once, the PoC of FPL comes with the (experimental) concept of *localization*. In every namespace we can include an optional block introduced by the keyword `localization` (short form `loc`). We can reuse localizations from other namespaces by importing them. Nevertheless, we MAY override them in our own namespaces. The syntax of this block is defined in LocalizationBlock. For instance, the namespace `Fpl.Commons` contains (among others) the following localizations:

```
@Fpl.Commons
{
```

```
   localization
   {
      iif(x,y) :=
      {
         ~tex:  x "\Leftrightarrow" y
         ~eng:  x " if and only if " y
         ~ger:  x " dann und nur dann wenn " y
         ;

      Equals(x,y) :=
      {
         ~tex:  x "=" y
         ~eng:  x " equals " y
         ~ger:  x " ist gleich " y
         ~ita:  x " è uguale a " y
         ~pol:  x " równa się " y
         ;
      }

      Succ(x) :=
      {
         ~tex:  \operatorname{succ}(" x ")"
         ~eng:  "successor of " x
         ~ger:  "Nachfolger von " x
         ~pol:  "następnik " x
         ;
      }
   }
}
```

The above excerpt contains an example of how we can translate in-built predicates in FPL (e.g., the predicate `iif`) and self-defined predicates (in our example `Equals` and `Succ`). Each translation starts with a tilde ~ followed by an ISO-639-2 language code. Moreover, by an FPL convention, we use the code `tex` (which is not part of this ISO standard) to denote LaTeX translations for which we take the same approach as for natural languages. The translations contain a receipt how to translate FPL code, mixing the same variables we used in the assignee (before the *":="*) with some terminal expressions formulated in these other languages (after the *":="* and the language code).

The idea is that we could trigger the actual translation process on demand, for instance in specialized FPL IDEs. Whenever we refer to the translated predicates somewhere from FPL code, we might use other variables. The FPL interpreter MAY replace the variables used in localization by the caller variables. For instance, if we had a reference like `Equals(m,Succ(n))` somewhere in our FPL code and we wished to translate it, we could

choose a language into which we want to translate it. Our FPL interpreter would then produce one of the following desired translations:

tex: *"m=\operatorname{succ}(n)"* (would be rendered by $m = \operatorname{succ}(n)$)

eng: *"m equals successor of n"*

ger: *"m ist gleich Nachfolger von n"*

pol: *"m równa się następnik n"*

Of course, our experimental approach is too simple to allow generating translations that are always grammatically correct. The more complicated the grammar of a natural language is, the more likely it is to get grammatically incorrect translations. For instance, in our above example, only the translation to LaTeX would be correct. The grammatically correct translations of the three natural languages in our example should have been

eng: *"m equals the successor of n"*,

ger: *"m ist gleich dem Nachfolger von n"*, and

pol: *"m równa się następnikowi n"*,

Instead, even grammatically not perfect translations into the own natural language should help novice users of FPL to understand their own or third-party code better.

8 So what?

The original title of this section was *"Next Steps"*. Taking into account how many attempts there have been in history to create a kind of *"Characteristica universalis"*[326] since Leibniz's vision, we do not expect that FPL will do the trick.

Nevertheless, we hope this document will help the community interested in this subject better understand why establishing such a language resists all attempts to create it and what we could ever expect it to look like (if it exists).

We also hope that the examples given in 7 demonstrate that FPL gives us the freedom, flexibility, and expressiveness we need to write and read PBM. Some of the examples also show that FPL makes mathematical definitions much more explicit than we could ever write them using a conventional mix of some natural language and conventional symbols of PBM. We hope that these examples will convince you to use FPL to express PBM, for instance, in educational scenarios, to teach proof-based mathematics!

There are, indeed, next steps of this project, and we invite the Internet community to collaborate. The project tasks include but are not limited to the following:

[326] see 4.1

1. First of all, along with the existing FPL parser, we need an appropriate FPL interpreter that would fulfill the semantic requirements of this specification and the PoC.
2. We need to continue the PoC by translating real use cases of mathematics into FPL to ensure that its syntax and semantics cover all features of PBM we need.
3. We need localization sections for every PoC theory.
4. We have to implement translators from FPL to LaTeX and natural languages based on localization.
5. We need IDEs with code completion and debugging capabilities.
6. We have to enhance FPL interpreters to check the correctness of mathematical proofs written in FPL or even to auto-generate proofs in FPL.
7. We need a conception of a byte code so we can pre-compile FPL theories and include libraries without having to parse and interpret them again; also, we need the corresponding FPL compilers.
8. We should create a globally accessible REST API service to provide the source code and the byte code of broadly accepted FPL formulations of PBM theories to facilitating importing them by end-users via the Internet.

9 Appendix

9.1 Boolean Algebra

A set $(B, \wedge, \vee, \neg, T, F)$ is called **Boolean algebra**, if and only if it fulfills the following axioms[327]:

1. $\wedge$ (**conjunction**) and $\vee$ (**disjunction**) are *associative* in B, i.e. for all $x, y, z \in B$:

$$\begin{aligned}(x \wedge (y \wedge z)) &= ((x \wedge y) \wedge z),\\ (x \vee (y \vee z)) &= ((x \vee y) \wedge z).\end{aligned}$$

2. $\wedge$ and $\vee$ are *commutative* in B, i.e. for all $x, y \in B$:

$$\begin{aligned}(x \wedge y) &= (y \wedge x),\\ (x \vee y) &= (y \vee x).\end{aligned}$$

3. $\wedge$ and $\vee$ are *distributive* over each other in B, i.e. for all $x, y, z \in B$:

$$\begin{aligned}(x \wedge (y \vee z)) &= ((x \vee y) \wedge (x \vee z)),\\ (x \vee (y \wedge z)) &= ((x \wedge y) \vee (x \wedge z)).\end{aligned}$$

4. For $F \in B$ (**false**) and every $x \in B$ it holds $F \wedge x = F$ and $F \vee x = x$.

5. For $T \in B$ (**true**) and every $x \in B$ it holds $T \wedge x = x$ and $T \vee x = T$.

6. For every $x \in B$ there is an element $\neg x \in B$ (**negation**) with $(x \wedge (\neg x) = F)$ and $(x \vee (\neg x)) = T$.

9.2 Zermelo-Fraenkel Axioms

ZFC: The Zermelo-Fraenkel axioms (including the axiom of choice)[328]:

- **Axiom of Existence**: There is an empty set.

$$\exists x(\forall z(z \notin x)).$$

  We denote this set by $\emptyset$.

- **Axiom of Extensionality**: Sets with the same elements are equal:

$$\forall xy(\forall z(z \in x \Leftrightarrow z \in y) \Rightarrow x = y).$$

[327] [37] p. 180

[328] There are different arrangements of these axioms, compare [62], [7], [22], or [61]. In particular, the *axiom of existence* is sometimes stated for any set or sometimes for the empty set only.

- **Schema of Separation Axioms**: For every[329] predicate of the form $p(z, \overset{n}{x})$ the following axiom holds: For all $\overset{n}{x}$ and all sets x there is a set y containing exactly those elements z of x fulfilling $p(z, \overset{n}{x})$:

$$\forall \overset{n}{x}(\forall x(\exists y(\forall z(z \in y \Rightarrow z \in x \wedge p(z, \overset{n}{x}))))).$$

  The schema justifies the set-builder notation since it ensures the existence (and with the axiom of extensionality the uniqueness) of a subset of elements of a set x fulfilling some given property $p(z, \overset{n}{x})$:

$$\forall \overset{n}{x}(\forall x(\exists y(y = \{z \in x \mid p(z, \overset{n}{x})\}))).$$

- **Axiom of Pairing**: For any two sets x, y there exists a set z containing them as elements.

$$\forall x, y(\exists z(\forall w(w \in z \Leftrightarrow w = x \vee w = y))).$$

- **Axiom of Union**: For every set x there is a set containing all elements of the elements of x:

$$\forall x(\exists y(\forall z, w(z \in w \wedge w \in x \Rightarrow z \in y))).$$

- **Axiom of Power Set**: For every set x there is a set y containing all subsets[330] of x (as is elements):

$$\forall x(\exists y(\forall z(z \subset x \Rightarrow z \in y))).$$

- **Axiom of Infinity**: There is a set x containing the empty set[331] and with every set z also the set $z \cup \{z\}$:

$$\exists x(\emptyset \in x \wedge \forall z(z \in x \Rightarrow z \cup \{z\} \in x)).$$

- **Schema of Replacement Axioms**: For every predicate of the form $p(x, y, \overset{n}{x})$ the following axiom holds: For all $\overset{n}{x}$: If for every x there is exactly one y with $p(x, y, \overset{n}{x})$, then there is for every u a set w containing for every x the element y that fulfills $p(x, y, \overset{n}{x})$:

$$\begin{array}{l} \forall \overset{n}{x}(\\ \quad (\forall x(\exists! y p(x, y, \overset{n}{x}))) \\ \quad \Rightarrow \\ \quad (\forall u(\exists w(\forall x y(x \in u \wedge p(x, y, \overset{n}{x}) \Rightarrow y \in w)))) \\ ). \end{array}$$

[329] We use the symbol $\overset{n}{x}$ to abbreviate the aliteration $x_1, \ldots, x_n$. Note that the schema contains infinitely many axioms (i.e. for infinitely many predicates). Thus, ZF(C) are an axiomatic system containing infinitely many axioms!

[330] The *subset* symbol $\subset$ is an abbreviation (more precisely a binary predicate) introduced in the set theory from a more complicated expression: $z \subset x \Leftrightarrow \forall w(w \in z \Rightarrow w \in x)$.

[331] The *empty set* can be defined in the set-theoretic language as $\emptyset := \{z \mid z \neq z\}$.

With the set-builder notation, this axiom justifies the existence (and with the axiom of extensionality the uniqueness) of a set w being the image of a set u under some predicate $\mathcal{P}$ in which the variable x is bound and some arguments $\overset{n}{x}$):

$$\forall \overset{n}{x}(\forall u(\exists! w(w = \{x \mid (x \in u \wedge \mathcal{P}(x, \overset{n}{x}))\}))).$$

- **Axiom of Foundation**: Every nonempty[332] set x contains an element that is disjoint from it:

  $$\forall x(x \neq \emptyset \Rightarrow \exists z(z \in x \wedge x \cap z = \emptyset)).$$

  The axiom ensures that every non-empty set contains an element that is minimal with respect to the $\in$ operator. It also implies that every set x is different from its singleton set $\{x\}$.

- **Axiom of Choice**: For every set X of disjoint and non-empty sets there is a set Y containing exactly one element from each of these sets.

$$\begin{array}{l}
\forall x(\\
\quad \forall uw(\\
\qquad (u \in x \wedge w \in x \Rightarrow u \neq \emptyset \wedge (u = w \vee u \cap w = \emptyset)) \\
\qquad \Rightarrow \\
\qquad (\exists y(\forall v(v \in x \Rightarrow \exists z(y \cap v = \{z\})))) \\
\quad) \\
).
\end{array}$$

[332] A *nonempty* set is a set that contains at least one element.

9.3 List of Abbreviations

Abbreviation	Full Text
AM	axiomatic method
API	Application Programming Interface
CCAF	Category of Categories
EBNF	Extended Backus-Naur Form
ERH	External Reality Hypothesis
ESP	Encoding Semantics Problem
IDE	Integrated Development Environment
IRP	Infinite Regress Problem
LIRP	Logical Infinite Regress Problem
FPL	Formal Proving Language
NBG	Neumann-Bernays-Gödel set theory
NP	Class of problems solvable by a Non-deterministic Turing machine in Polynomial time
OO	Object-oriented
PBM	Proof-Based Mathematics
QED	(Lat.) Quod erat demonstrandum
PL0	Propositional Logic
PL1	First-Order Predicate Logic
PLm	mth-Order Predicate Logic
PoC	Proof of Concept
REST	REpresentational State Transfer
RH	Riemann Hypothesis
SAT	Decision problem of satisfiability of a proposition in PL0
SIRP	Semantical Infinite Regress Problem
VMC	Vicious Mathematical Circle
ZF	Zermelo-Fraenkel axiomatic system
ZFC	Zermelo-Fraenkel axiomatic system with the Axiom of Choice

9.4 List of Tables

9.5 List of Figures

9.6 Syntax Diagrams of the FPL Grammar

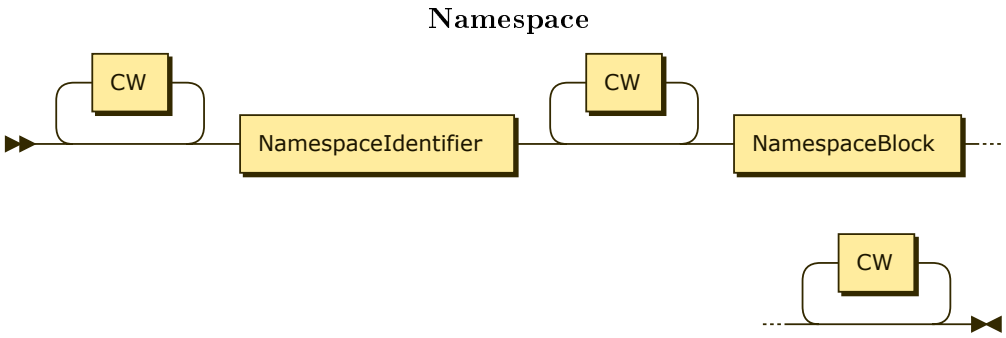


Figure 7: Non-terminals: NamespaceIdentifier, NamespaceBlock, CW

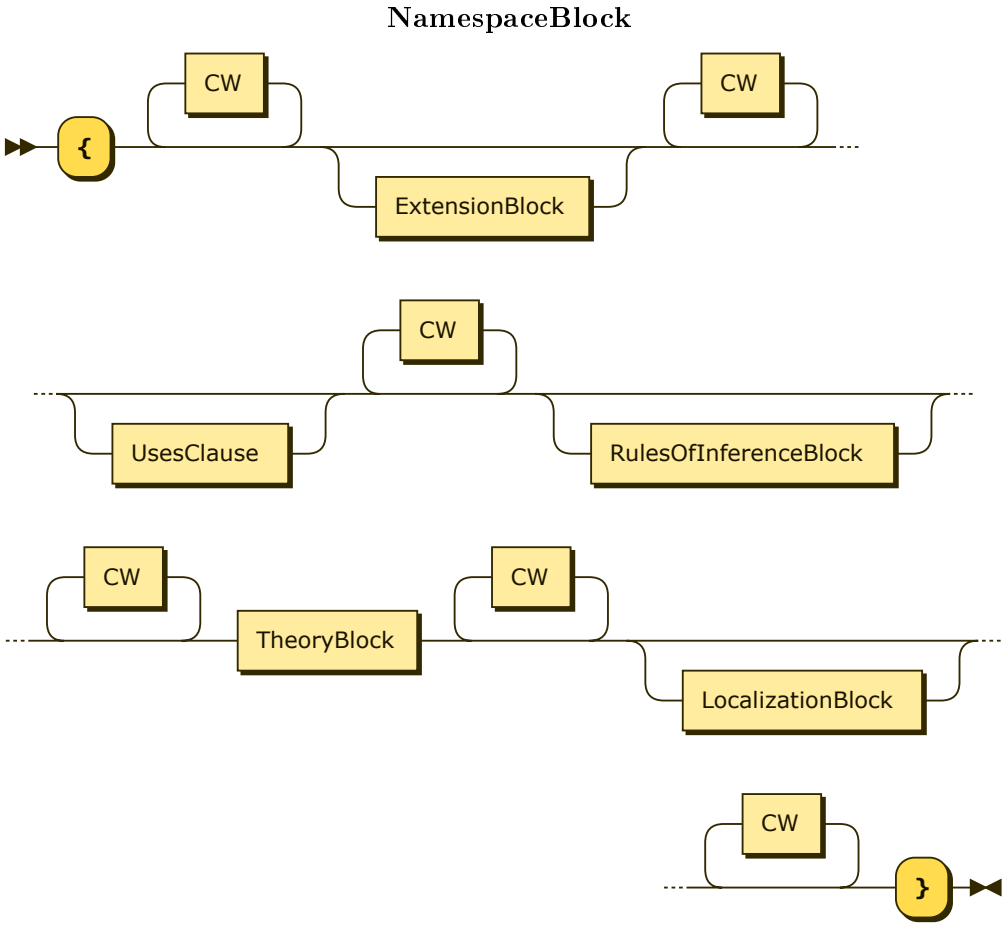


Figure 8: Terminals: `}`, `{`; Non-terminals: CW, TheoryBlock, LocalizationBlock, UsesClause, RulesOfInferenceBlock, ExtensionBlock

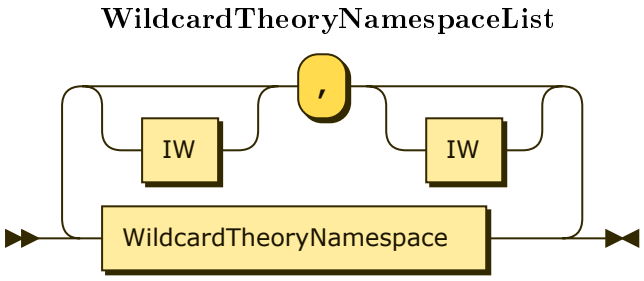


Figure 9: Terminals: `,`; Non-terminals: IW, WildcardTheoryNamespace

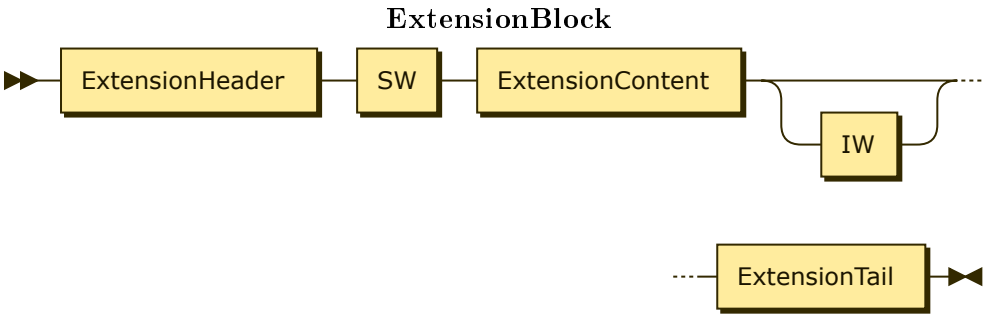


Figure 10: Non-terminals: ExtensionTail, IW, SW, ExtensionContent, ExtensionHeader

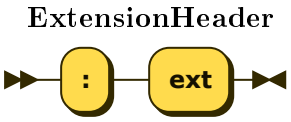


Figure 11: Terminals: `:`, `ext`

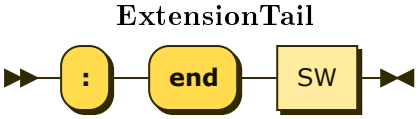


Figure 12: Terminals: `end`, `:`; Non-terminals: SW

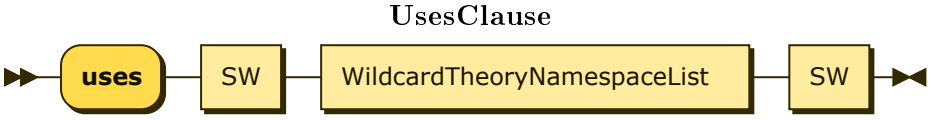


Figure 13: Terminals: `uses`; Non-terminals: WildcardTheoryNamespaceList, SW

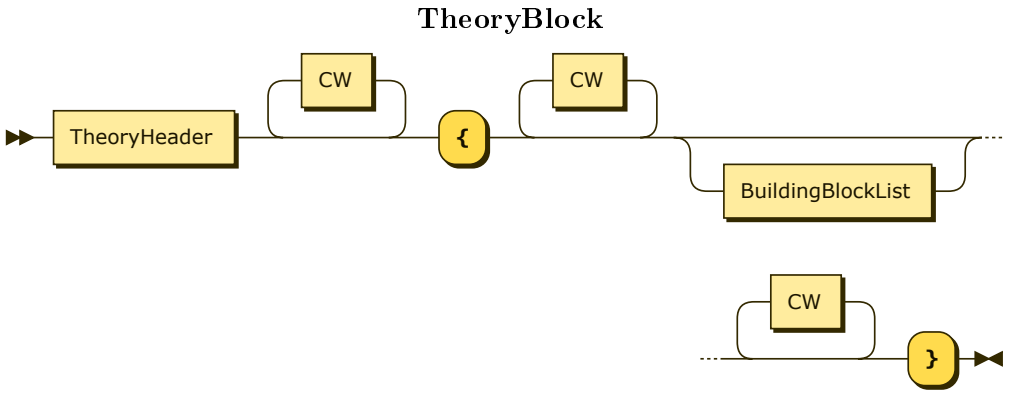


Figure 14: Terminals: `}`, `{`; Non-terminals: BuildingBlockList, TheoryHeader, CW

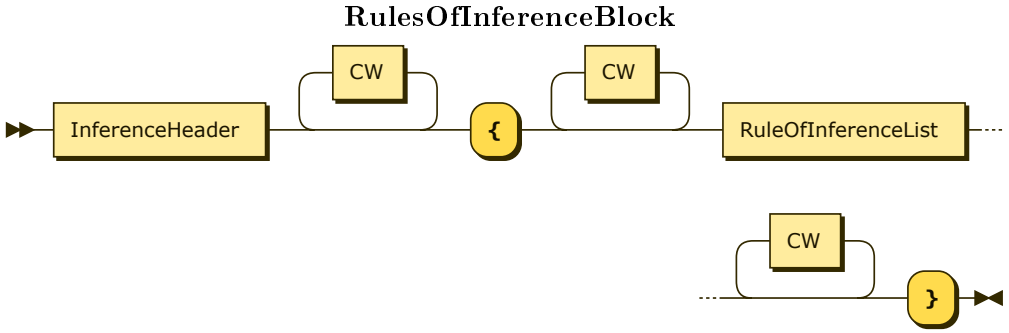


Figure 15: Terminals: `}`, `{`; Non-terminals: RuleOfInferenceList, CW, InferenceHeader

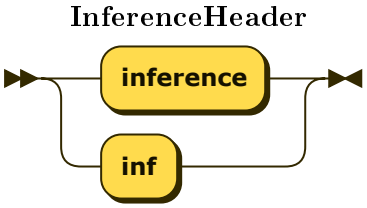


Figure 16: Terminals: `inf`, `inference`

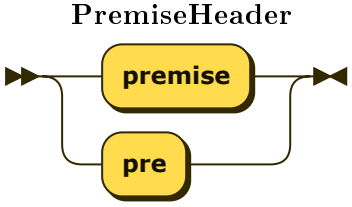


Figure 17: Terminals: `premise`, `pre`

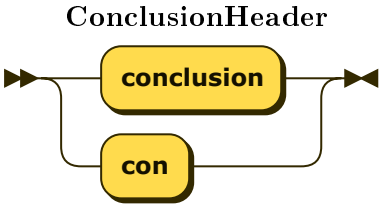


Figure 18: Terminals: `conclusion`, `con`

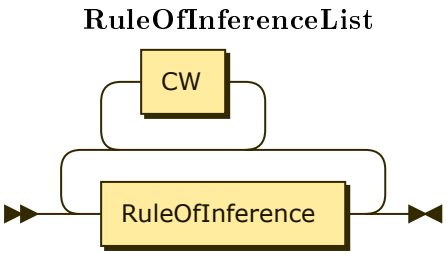


Figure 19: Non-terminals: CW, RuleOfInference

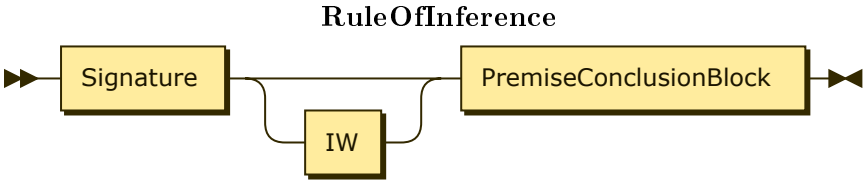


Figure 20: Non-terminals: PremiseConclusionBlock, IW, Signature

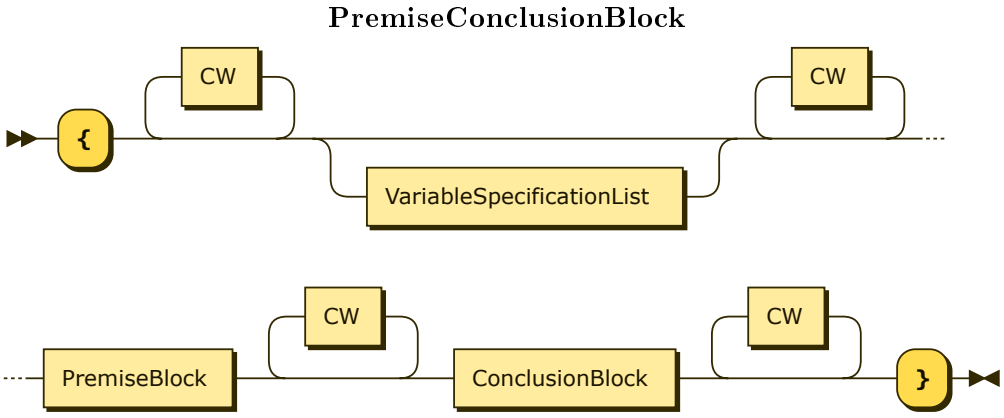


Figure 21: Terminals: `}`, `{`; Non-terminals: PremiseBlock, VariableSpecificationList, CW, ConclusionBlock

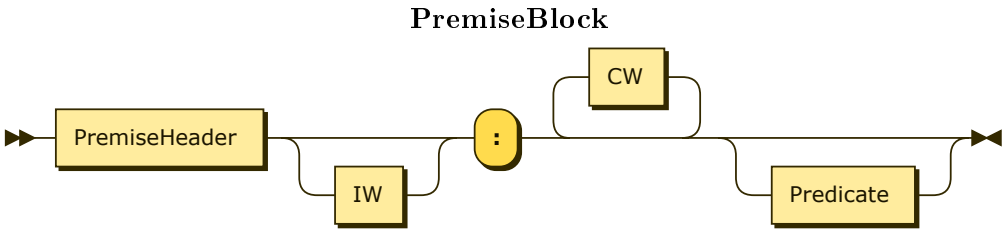


Figure 22: Terminals: :; Non-terminals: IW, PremiseHeader, CW, Predicate

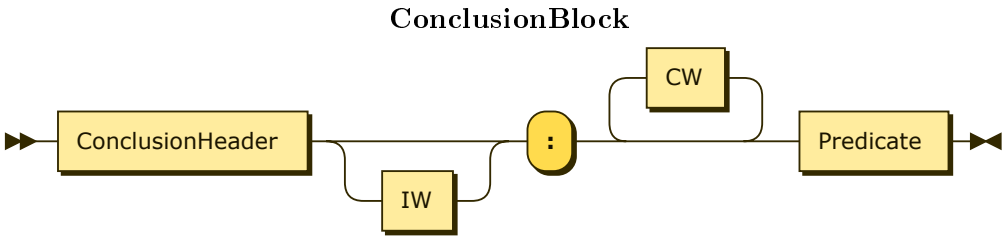


Figure 23: Terminals: :; Non-terminals: Predicate, IW, ConclusionHeader, CW

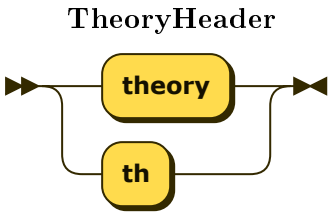


Figure 24: Terminals: `theory`, `th`

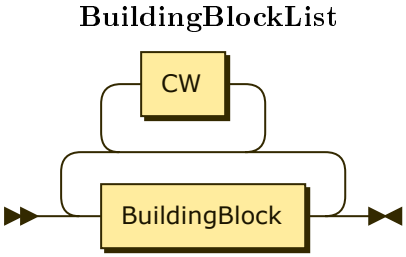


Figure 25: Non-terminals: BuildingBlock, CW

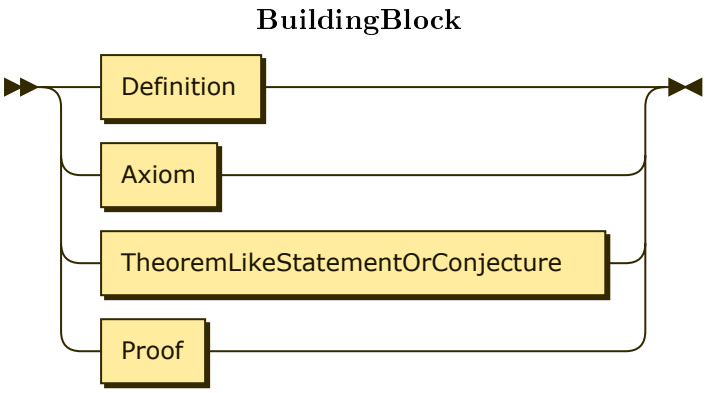


Figure 26: Non-terminals: Axiom, TheoremLikeStatementOrConjecture, Proof, Definition

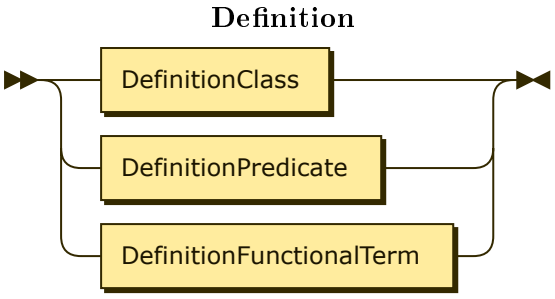


Figure 27: Non-terminals: DefinitionPredicate, DefinitionFunctionalTerm, DefinitionClass

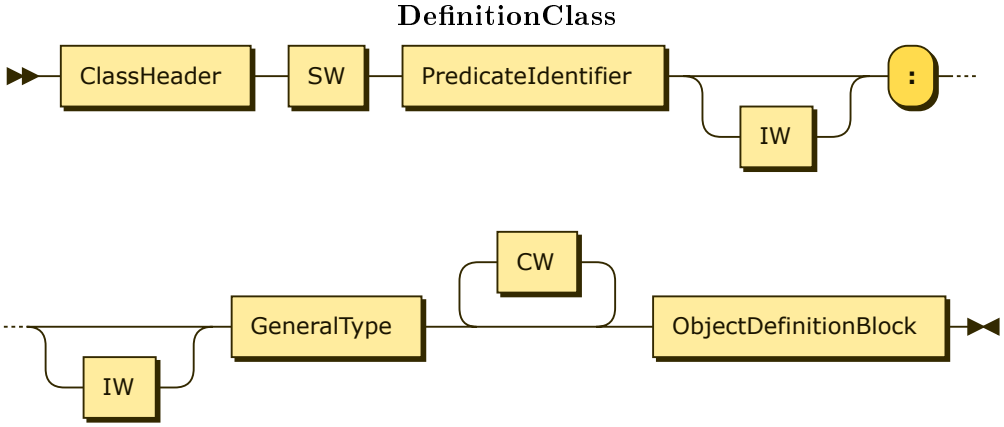


Figure 28: Terminals: :; Non-terminals: CW, ObjectDefinitionBlock, IW, SW, ClassHeader, GeneralType, PredicateIdentifier

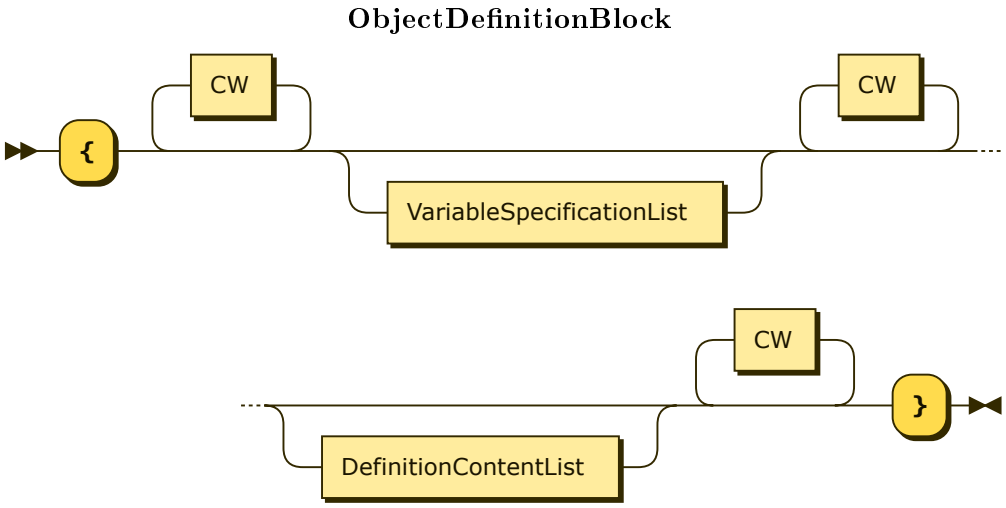


Figure 29: Terminals: }, {; Non-terminals: VariableSpecificationList, DefinitionContentList, CW

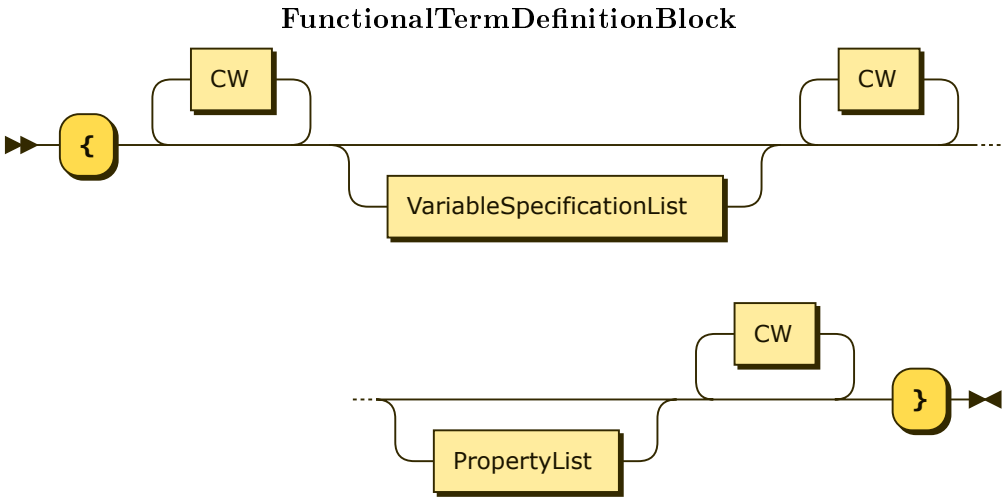


Figure 30: Terminals: }, {; Non-terminals: VariableSpecificationList, CW, PropertyList

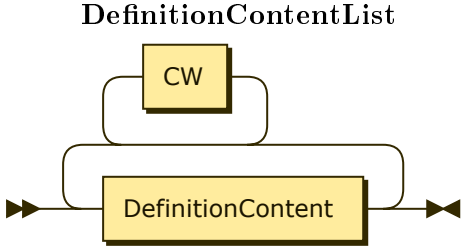


Figure 31: Non-terminals: CW, DefinitionContent

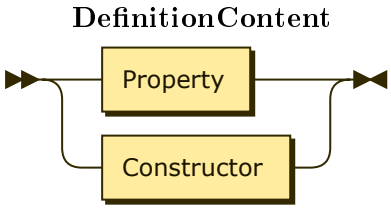


Figure 32: Non-terminals: Constructor, Property

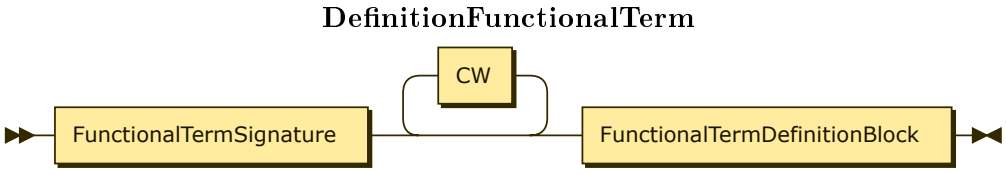


Figure 33: Non-terminals: FunctionalTermDefinitionBlock, FunctionalTermSignature, CW

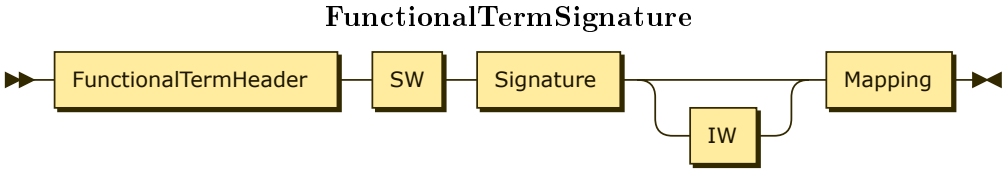


Figure 34: Non-terminals: Signature, IW, Mapping, SW, FunctionalTermHeader

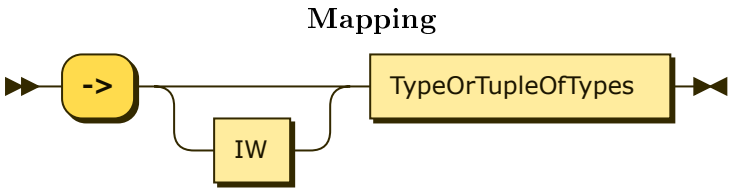


Figure 35: Terminals: >; Non-terminals: IW, TypeOrTupleOfTypes

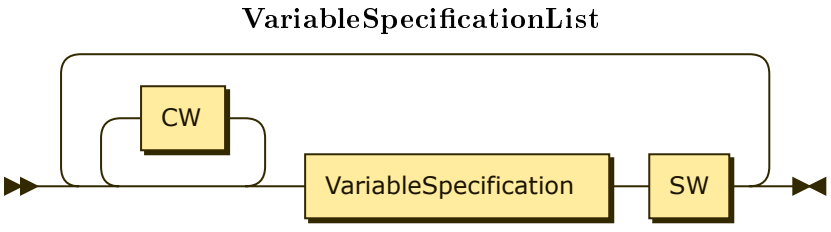


Figure 36: Non-terminals: VariableSpecification, CW, SW

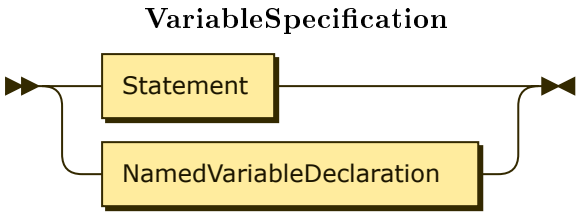


Figure 37: Non-terminals: NamedVariableDeclaration, Statement

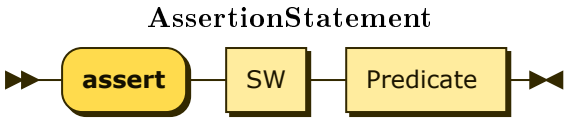


Figure 38: Terminals: `assert`; Non-terminals: Predicate, SW

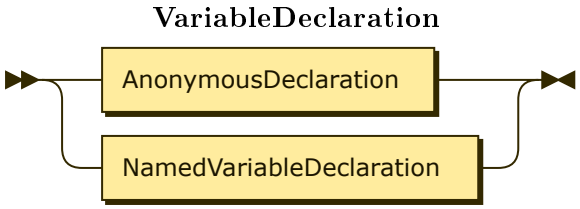


Figure 39: Non-terminals: NamedVariableDeclaration, AnonymousDeclaration

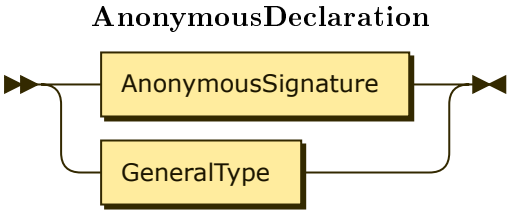


Figure 40: Non-terminals: GeneralType, AnonymousSignature

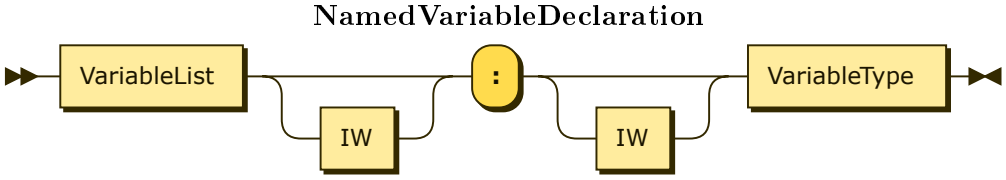


Figure 41: Terminals: `:`; Non-terminals: VariableType, IW, VariableList

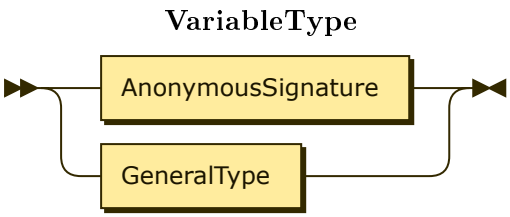


Figure 42: Non-terminals: GeneralType, AnonymousSignature

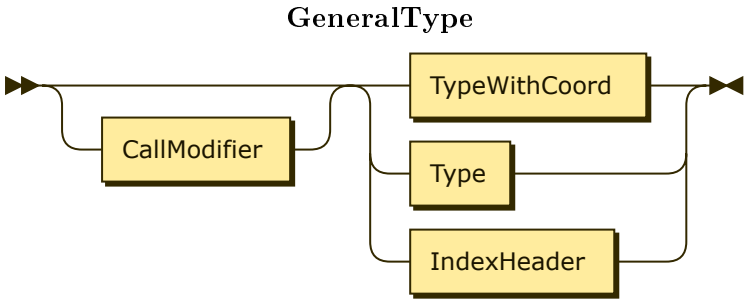


Figure 43: Non-terminals: Type, CallModifier, TypeWithCoord, IndexHeader

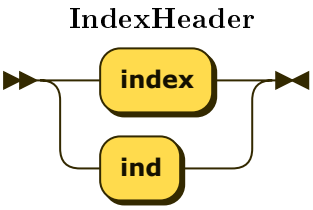


Figure 44: Terminals: `index`, `ind`

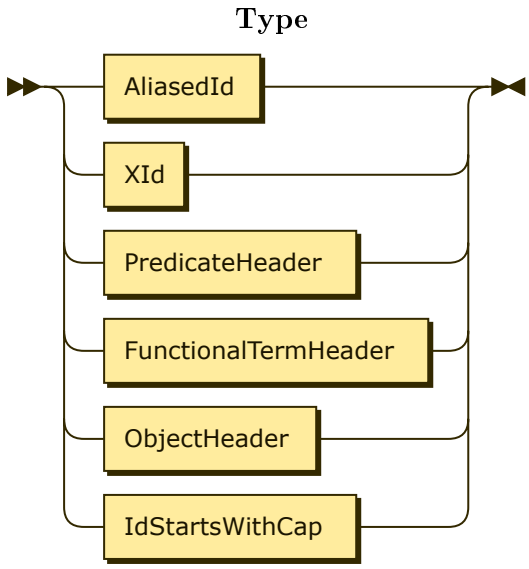


Figure 45: Non-terminals: XId, PredicateHeader, FunctionalTermHeader, IdStartsWithCap, ObjectHeader, AliasedId

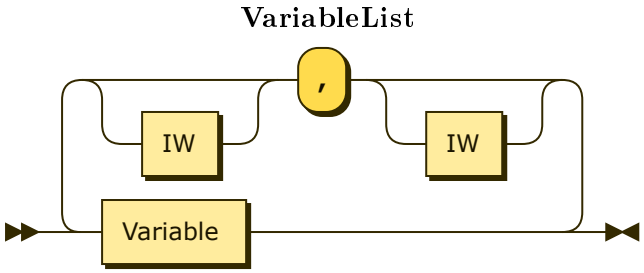


Figure 46: Terminals: ,; Non-terminals: IW, Variable

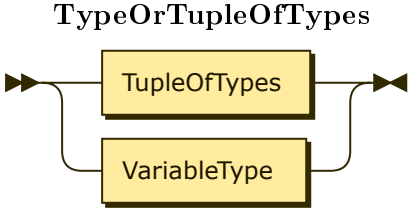


Figure 47: Non-terminals: VariableType, TupleOfTypes

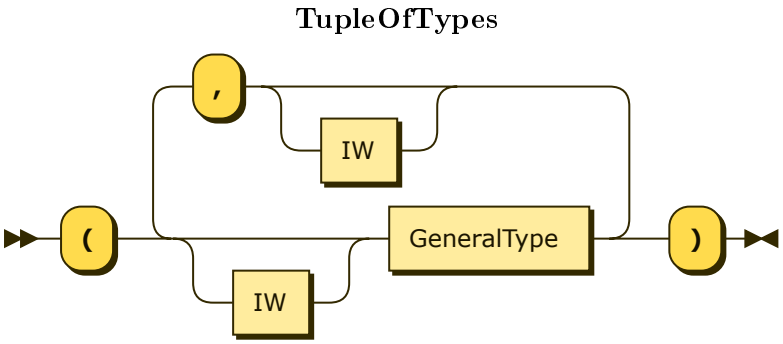


Figure 48: Terminals: `,`, `(`, `)`; Non-terminals: IW, GeneralType

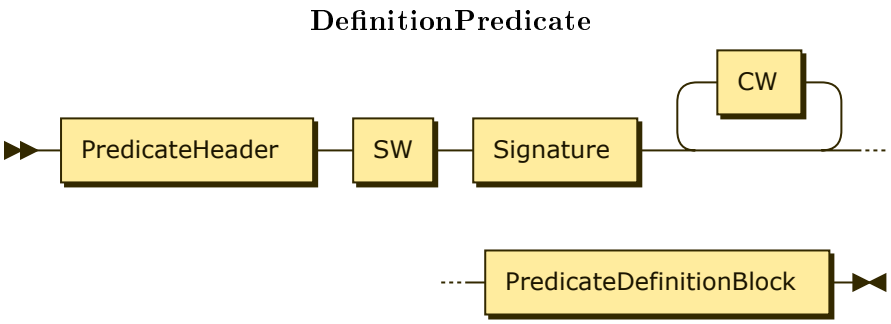


Figure 49: Non-terminals: CW, Signature, PredicateHeader, SW, PredicateDefinitionBlock

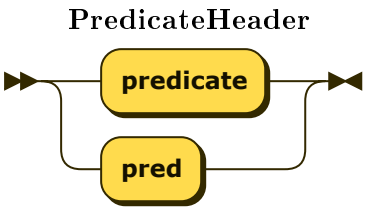


Figure 50: Terminals: `pred`, `predicate`

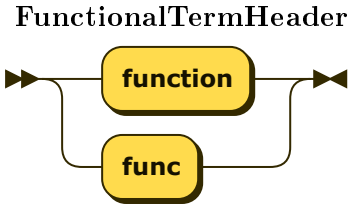


Figure 51: Terminals: `function`, `func`

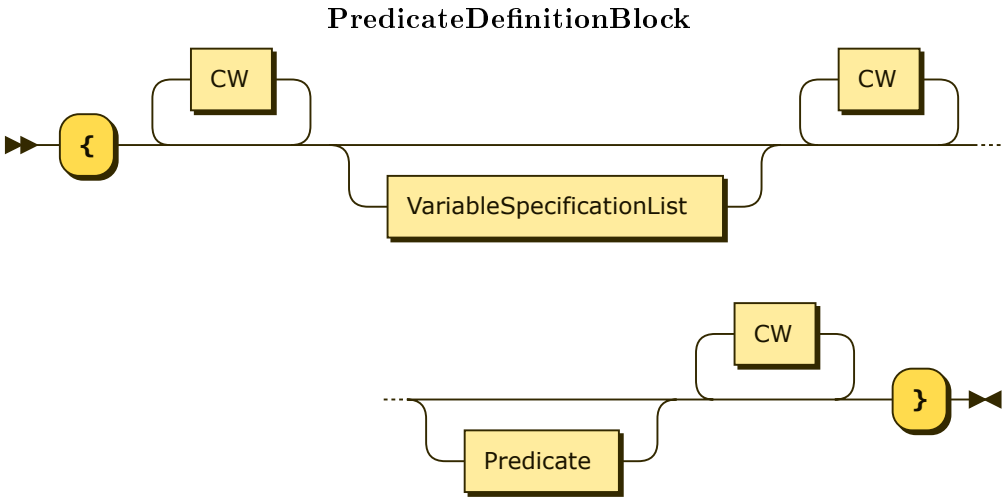


Figure 52: Terminals: `}`, `{`; Non-terminals: Predicate, VariableSpecificationList, CW

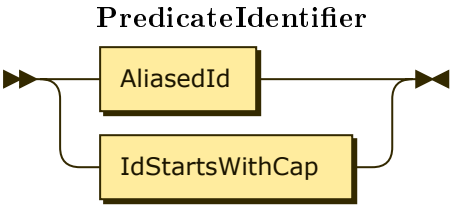


Figure 53: Non-terminals: IdStartsWithCap, AliasedId

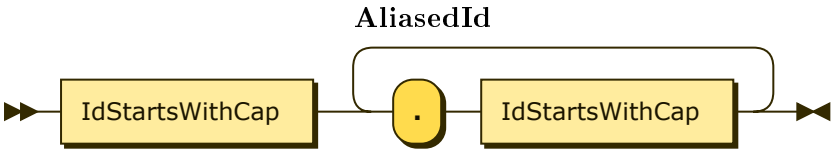


Figure 54: Terminals: `.`; Non-terminals: IdStartsWithCap

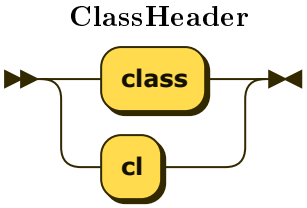


Figure 55: Terminals: `class`, `cl`

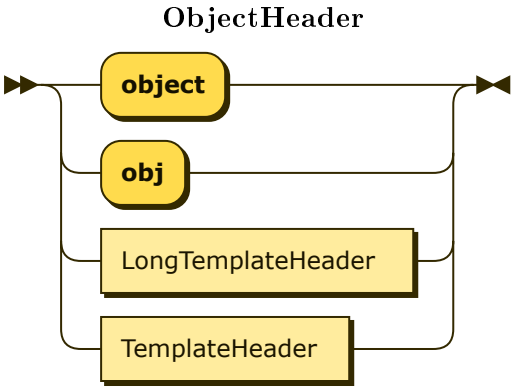


Figure 56: Terminals: `obj`, `object`; Non-terminals: TemplateHeader, LongTemplateHeader

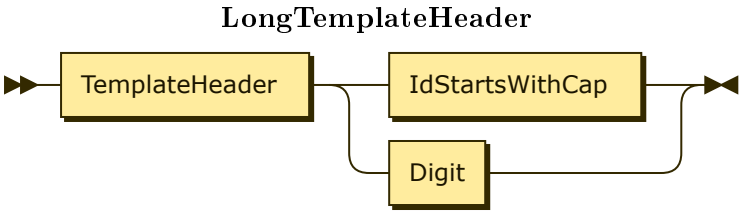


Figure 57: Non-terminals: TemplateHeader, Digit, IdStartsWithCap

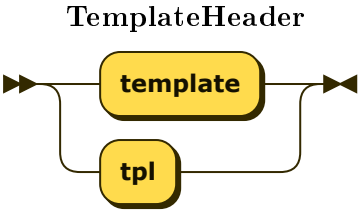


Figure 58: Terminals: `template`, `tpl`

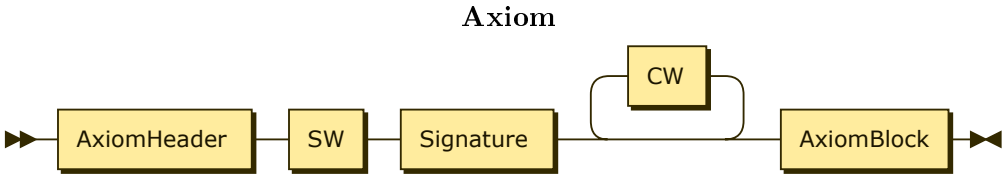


Figure 59: Non-terminals: Signature, CW, AxiomBlock, SW, AxiomHeader

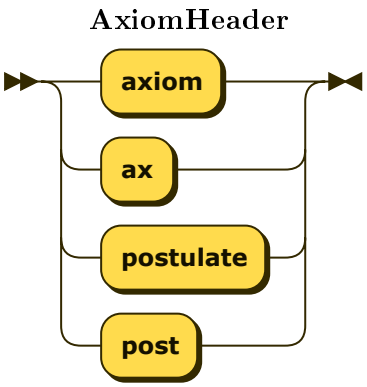


Figure 60: Terminals: `postulate`, `axiom`, `post`, `ax`

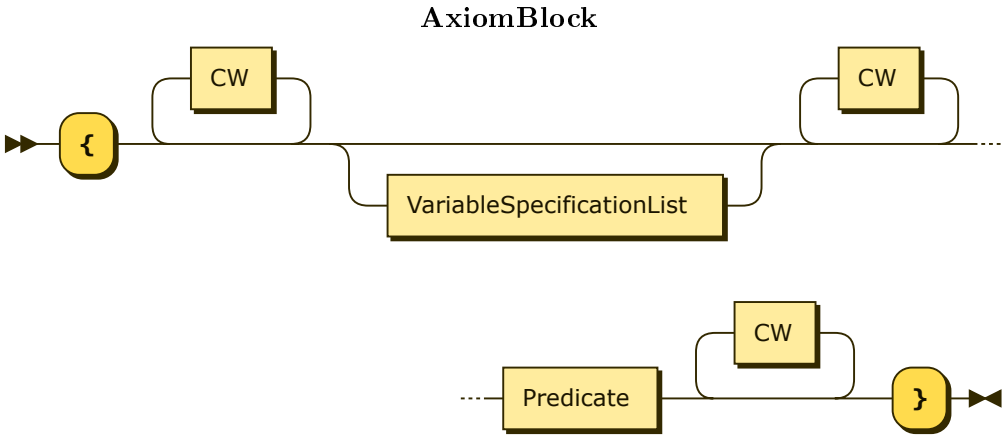


Figure 61: Terminals: `}`, `{`; Non-terminals: Predicate, VariableSpecificationList, CW

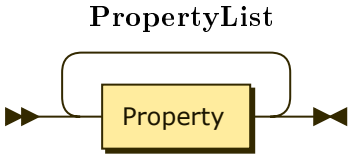


Figure 62: Non-terminals: Property

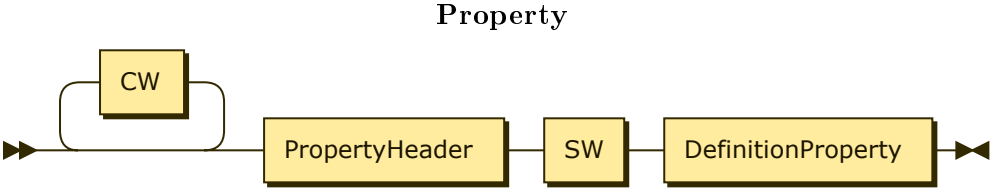


Figure 63: Non-terminals: DefinitionProperty, CW, SW, PropertyHeader

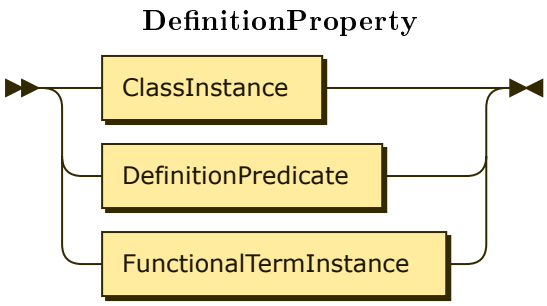


Figure 64: Non-terminals: DefinitionPredicate, FunctionalTermInstance, ClassInstance

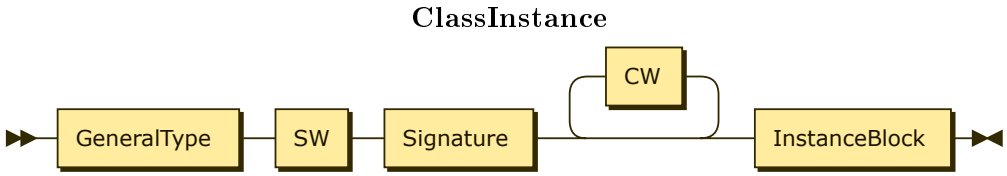


Figure 65: Non-terminals: CW, Signature, SW, InstanceBlock, GeneralType

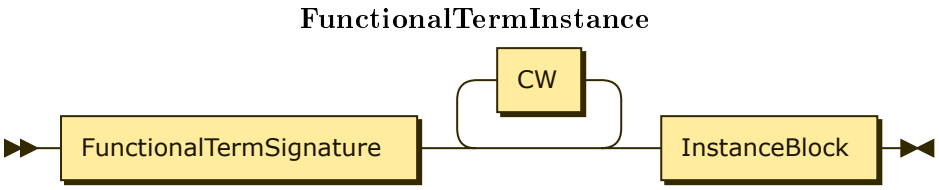


Figure 66: Non-terminals: FunctionalTermSignature, CW, InstanceBlock

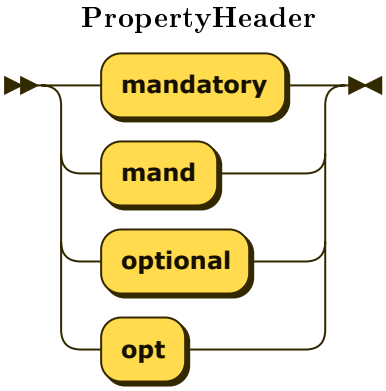


Figure 67: Terminals: `optional`, `mand`, `mandatory`, `opt`

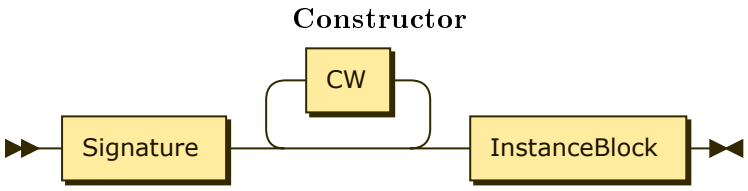


Figure 68: Non-terminals: InstanceBlock, Signature, CW

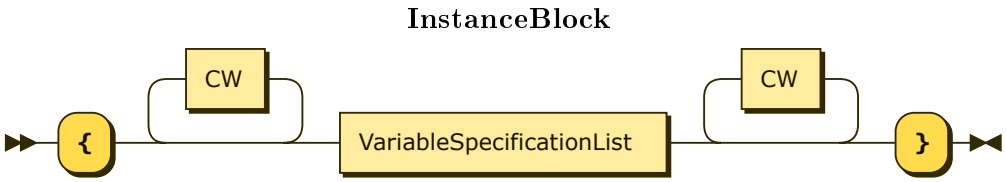


Figure 69: Terminals: `}`, `{`; Non-terminals: VariableSpecificationList, CW

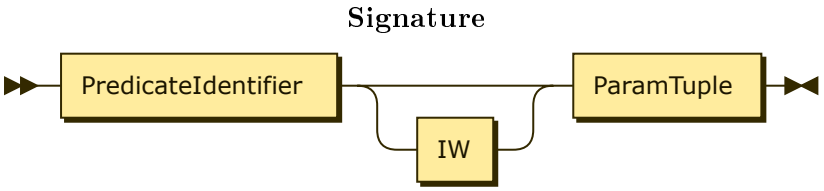


Figure 70: Non-terminals: IW, PredicateIdentifier, ParamTuple

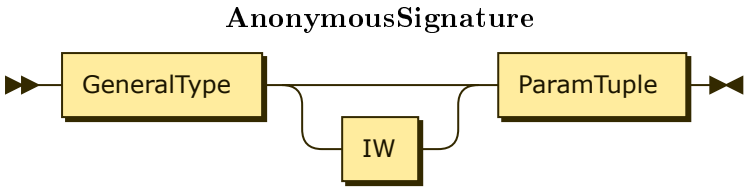


Figure 71: Non-terminals: IW, GeneralType, ParamTuple

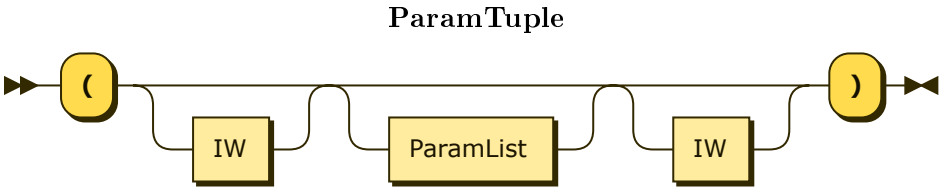


Figure 72: Terminals: `(`, `)`; Non-terminals: IW, ParamList

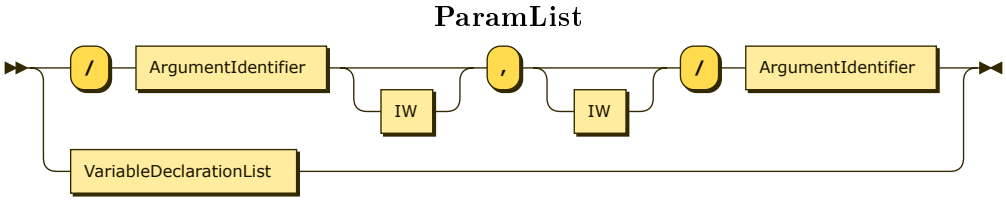


Figure 73: Terminals: `,`, `/`; Non-terminals: IW, VariableDeclarationList, ArgumentIdentifier

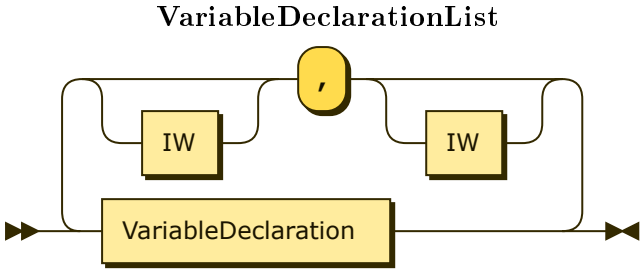


Figure 74: Terminals: `,`; Non-terminals: IW, VariableDeclaration

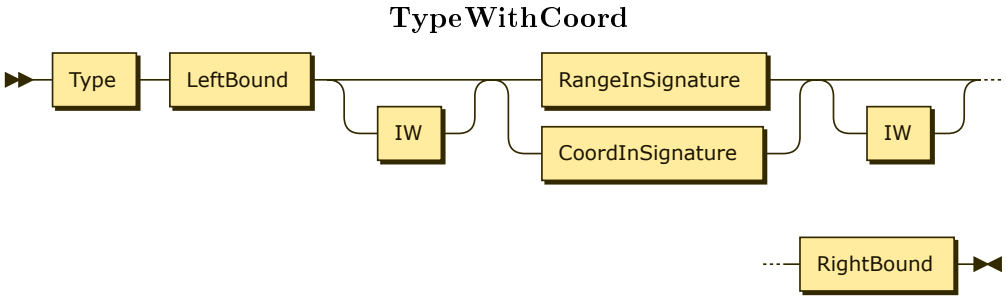


Figure 75: Non-terminals: IW, Type, LeftBound, CoordInSignature, RightBound, RangeInSignature

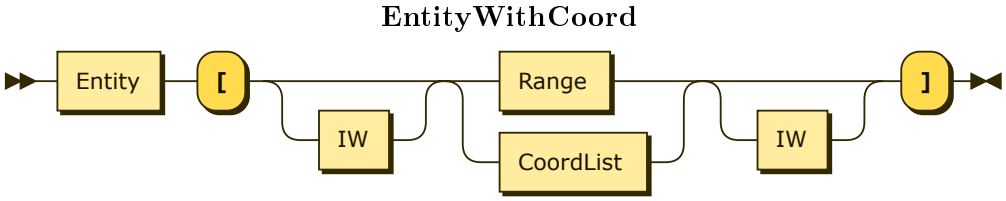


Figure 76: Terminals: `[`, `]`; Non-terminals: IW, Range, Entity, CoordList

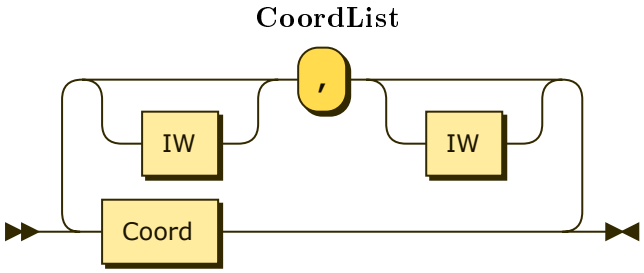


Figure 77: Terminals: `,`; Non-terminals: Coord, IW

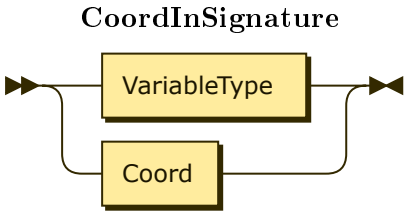


Figure 78: Non-terminals: Coord, VariableType

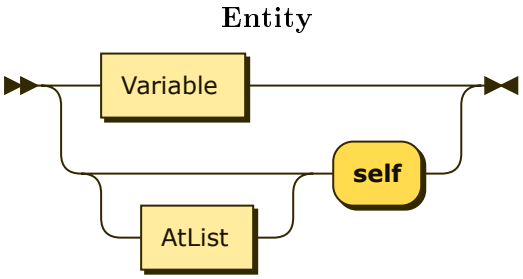


Figure 79: Terminals: `self`; Non-terminals: Variable, AtList

Figure 80: Terminals: `@`

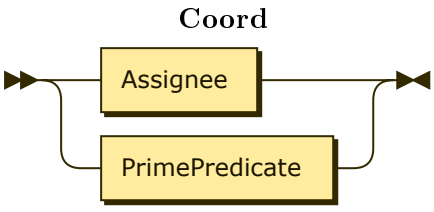


Figure 81: Non-terminals: PrimePredicate, Assignee

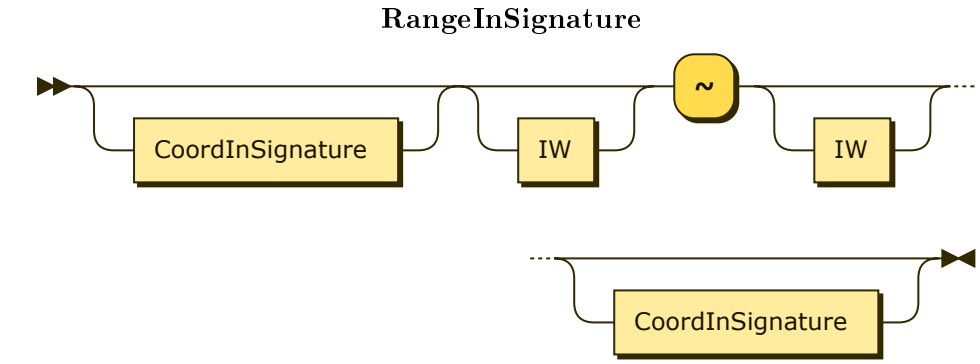


Figure 82: Terminals: ; Non-terminals: CoordInSignature, IW

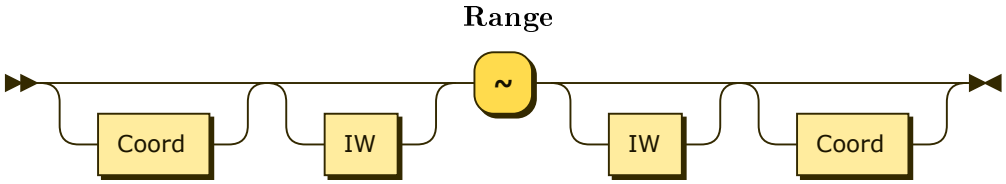


Figure 83: Terminals: ; Non-terminals: Coord, IW

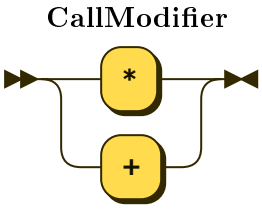


Figure 84: Terminals: `*`, `+`

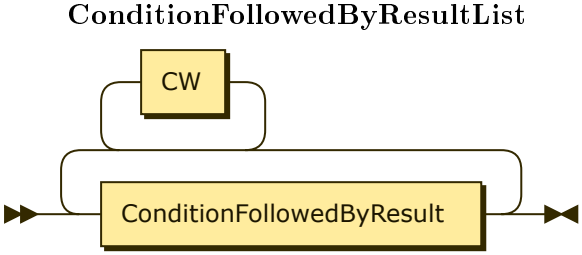


Figure 85: Non-terminals: ConditionFollowedByResult, CW

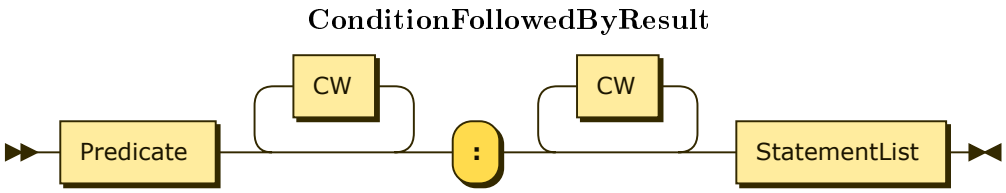


Figure 86: Terminals: :; Non-terminals: Predicate, CW, StatementList

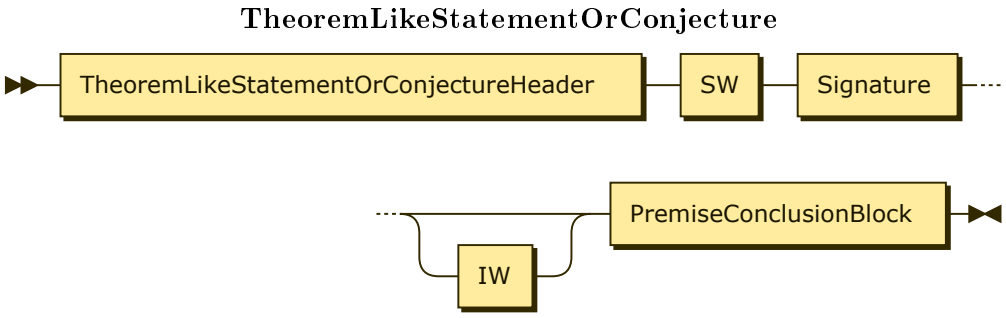


Figure 87: Non-terminals: Signature, IW, SW, TheoremLikeStatementOrConjecture-Header, PremiseConclusionBlock

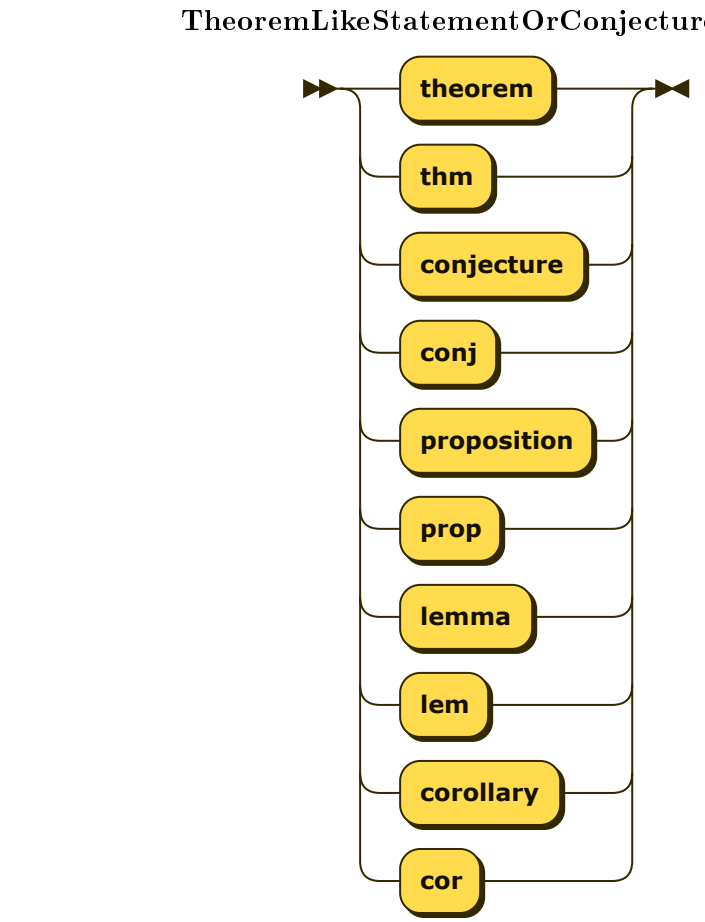


Figure 88: Terminals: `conj`, `proposition`, `cor`, `corollary`, `conjecture`, `thm`, `lem`, `theorem`, `prop`, `lemma`

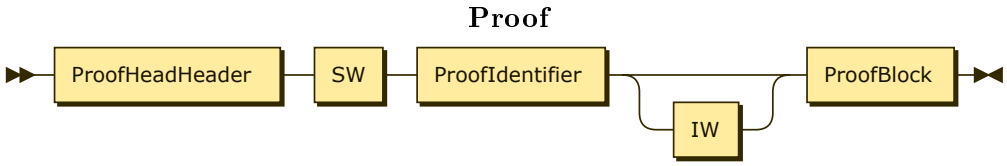


Figure 89: Non-terminals: IW, SW, ProofHeadHeader, ProofIdentifier, ProofBlock

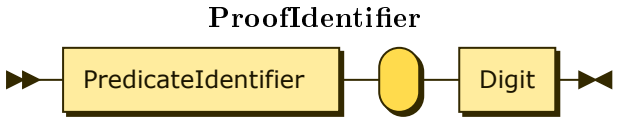


Figure 90: Non-terminals: Digit, PredicateIdentifier

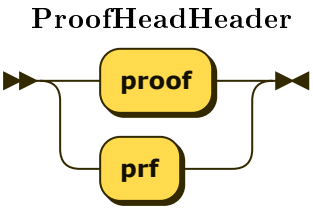


Figure 91: Terminals: `prf`, `proof`

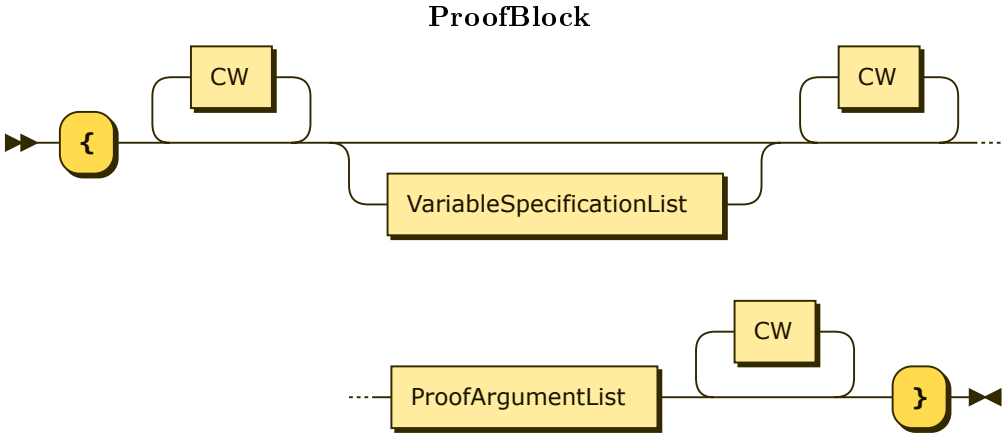


Figure 92: Terminals: `}`, `{`; Non-terminals: VariableSpecificationList, ProofArgumentList, CW

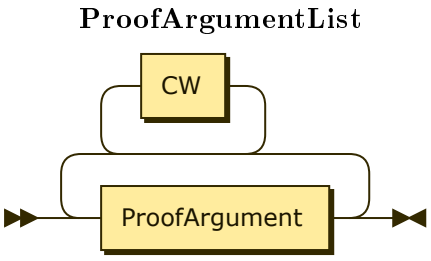


Figure 93: Non-terminals: ProofArgument, CW

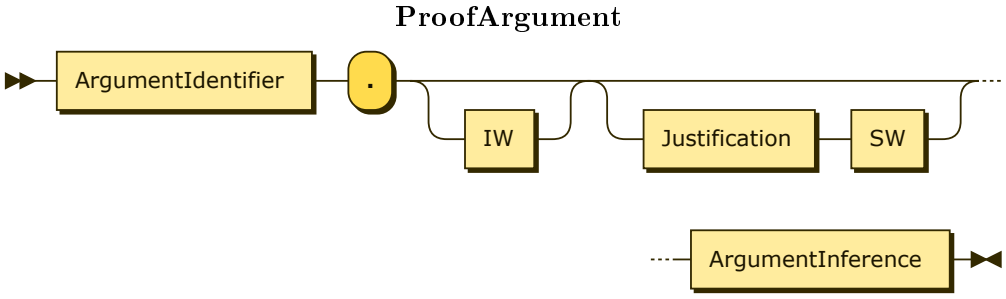


Figure 94: Terminals: `.`; Non-terminals: IW, ArgumentInference, ArgumentIdentifier, SW, Justification

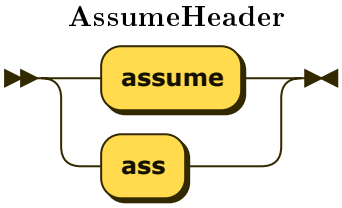


Figure 95: Terminals: `assume`, `ass`

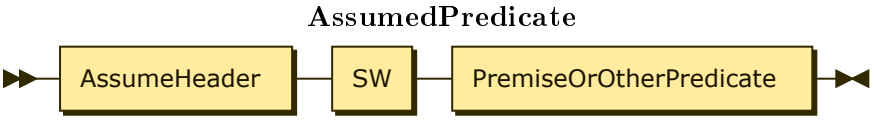


Figure 96: Non-terminals: PremiseOrOtherPredicate, AssumeHeader, SW

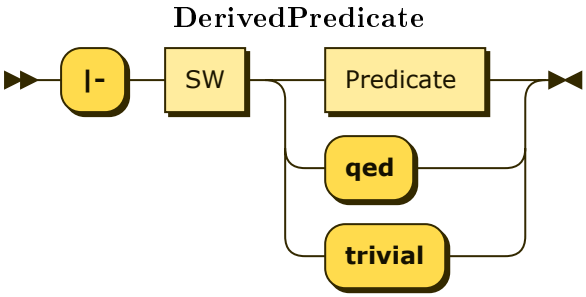


Figure 97: Terminals: `|-`, `qed`, `trivial`; Non-terminals: Predicate, SW

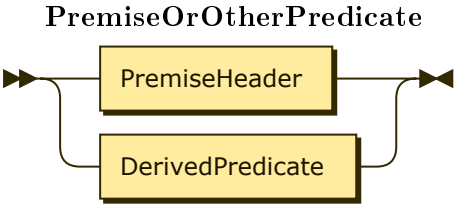


Figure 98: Non-terminals: PremiseHeader, DerivedPredicate

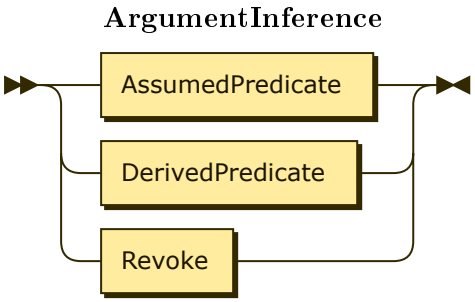


Figure 99: Non-terminals: Revoke, AssumedPredicate, DerivedPredicate

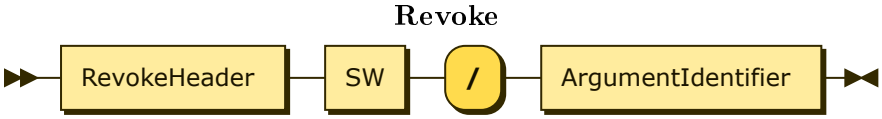


Figure 100: Terminals: `/`; Non-terminals: RevokeHeader, ArgumentIdentifier, SW

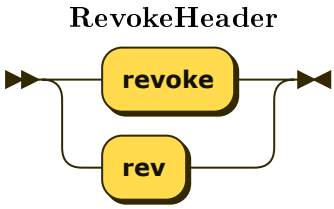


Figure 101: Terminals: `revoke`, `rev`

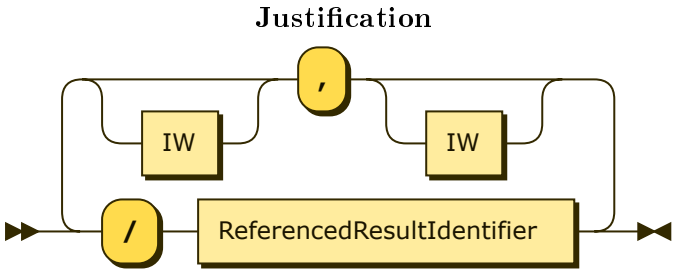


Figure 102: Terminals: `,`, `/`; Non-terminals: IW, ReferencedResultIdentifier

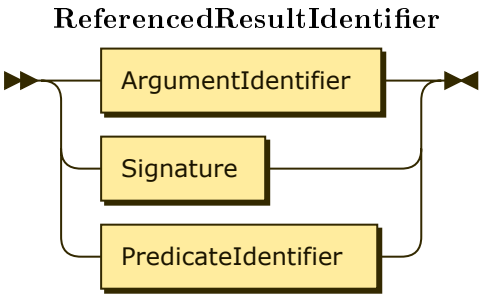


Figure 103: Non-terminals: PredicateIdentifier, ArgumentIdentifier, Signature

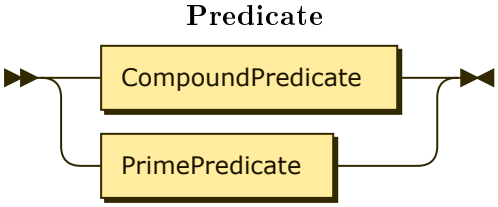


Figure 104: Non-terminals: CompoundPredicate, PrimePredicate

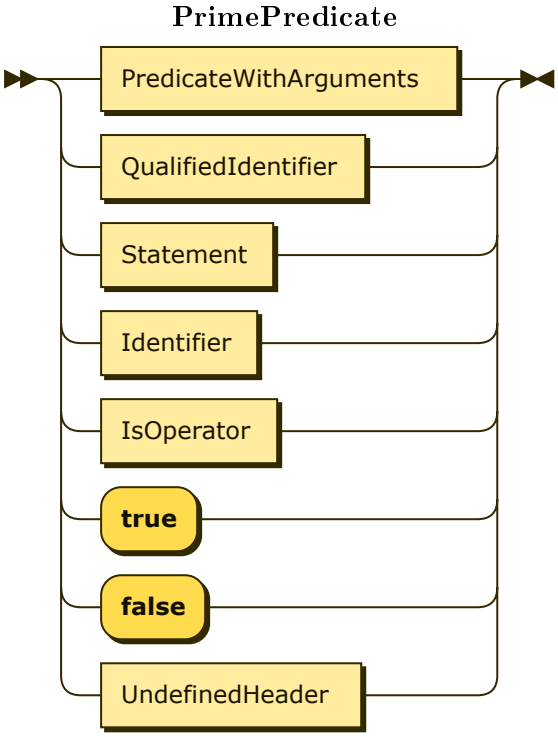


Figure 105: Terminals: `false`, `true`; Non-terminals: Identifier, IsOperator, UndefinedHeader, PredicateWithArguments, QualifiedIdentifier, Statement

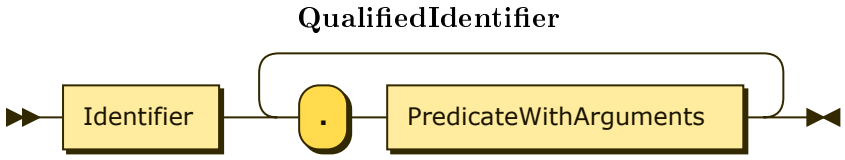


Figure 106: Terminals: `.`; Non-terminals: PredicateWithArguments, Identifier

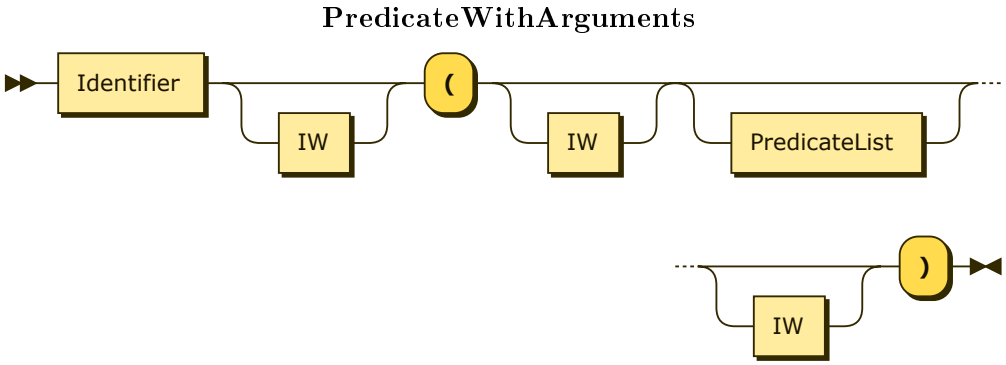


Figure 107: Terminals: `(`, `)`; Non-terminals: IW, PredicateList, Identifier

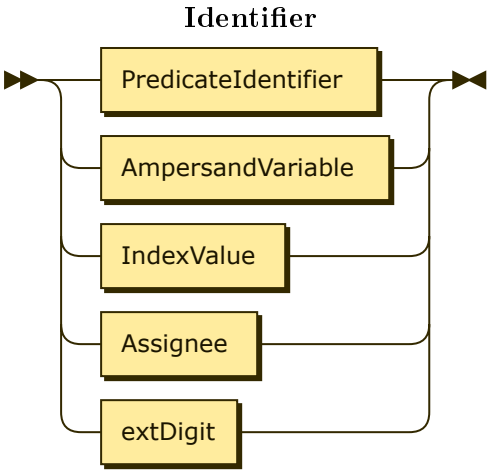


Figure 108: Non-terminals: AmpersandVariable, IndexValue, Assignee, extDigit, PredicateIdentifier

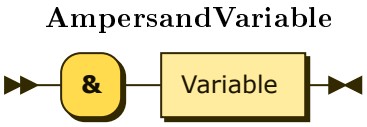


Figure 109: Terminals: `;`; Non-terminals: Variable

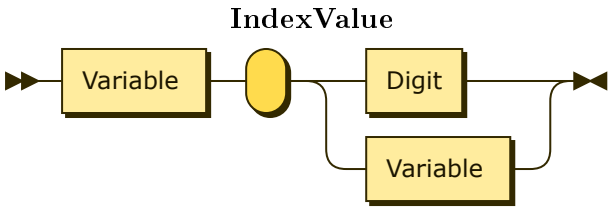


Figure 110: Non-terminals: Digit, Variable

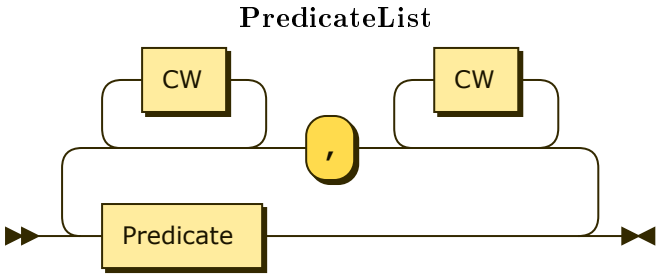


Figure 111: Terminals: ,; Non-terminals: Predicate, CW

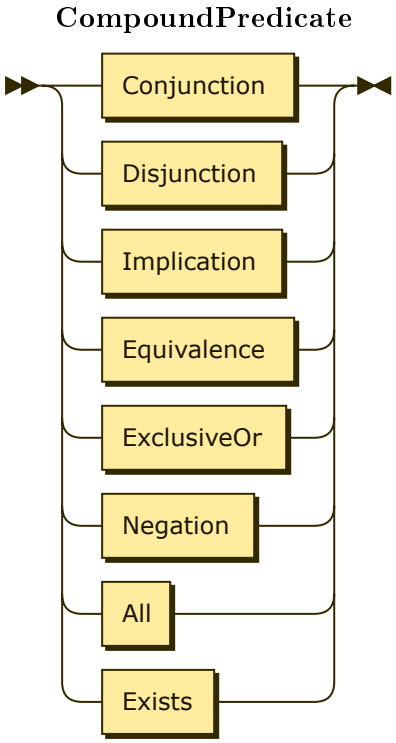


Figure 112: Non-terminals: Exists, Negation, Conjunction, ExclusiveOr, All, Equivalence, Implication, Disjunction

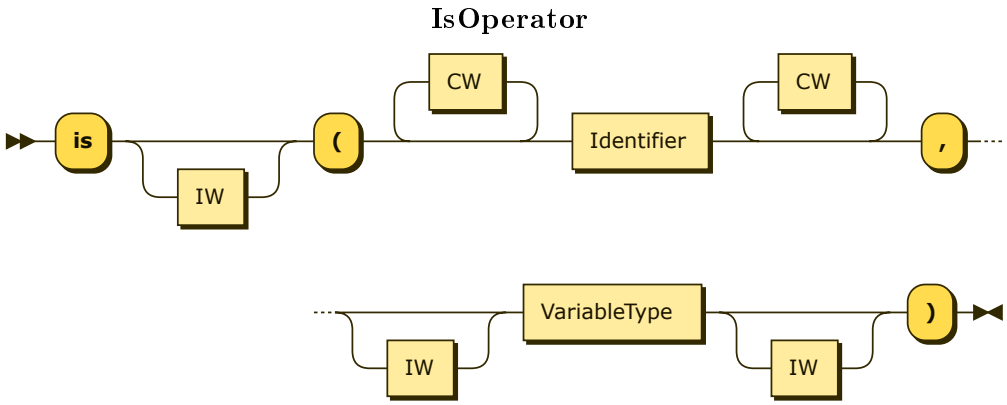


Figure 113: Terminals: `,`, `is`, `(`, `)`; Non-terminals: IW, VariableType, CW, Identifier

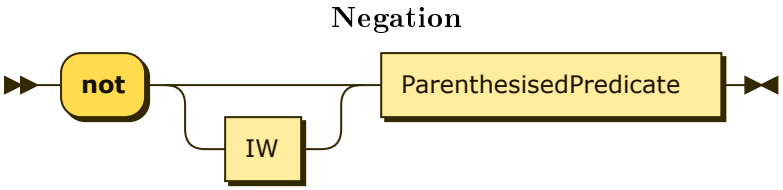


Figure 114: Terminals: `not`; Non-terminals: ParenthesisedPredicate, IW

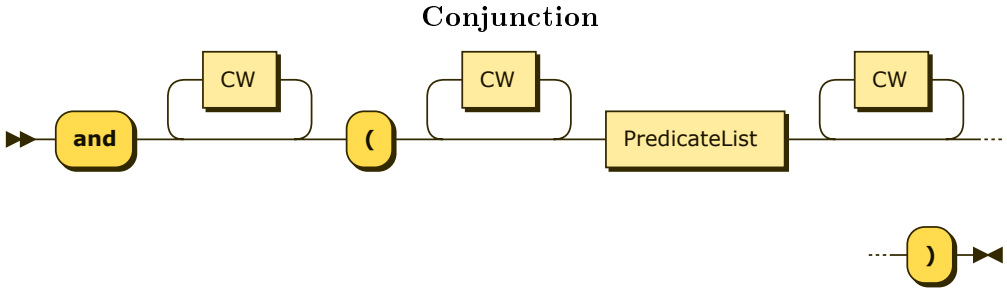


Figure 115: Terminals: `and`, `(`, `)`; Non-terminals: PredicateList, CW

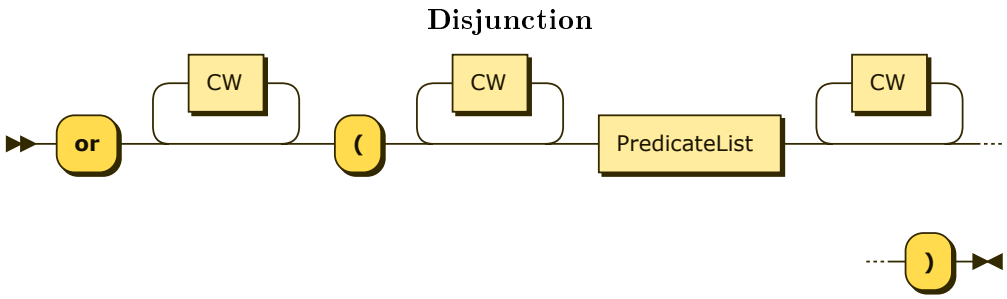


Figure 116: Terminals: `or`, `(`, `)`; Non-terminals: PredicateList, CW

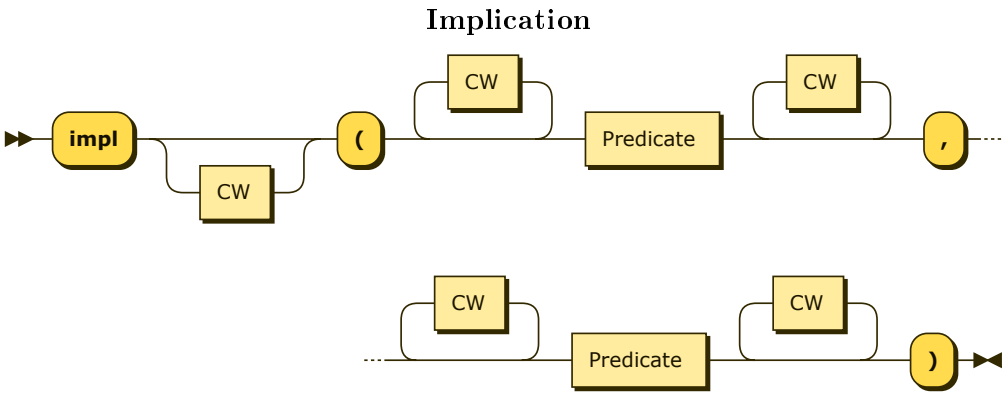


Figure 117: Terminals: `,`, `impl`, `(`, `)`; Non-terminals: Predicate, CW

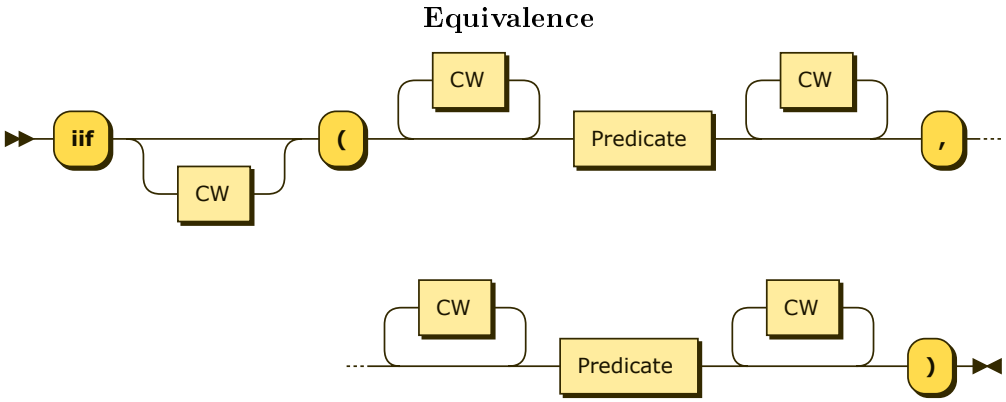


Figure 118: Terminals: `,`, `iif`, `(`, `)`; Non-terminals: Predicate, CW

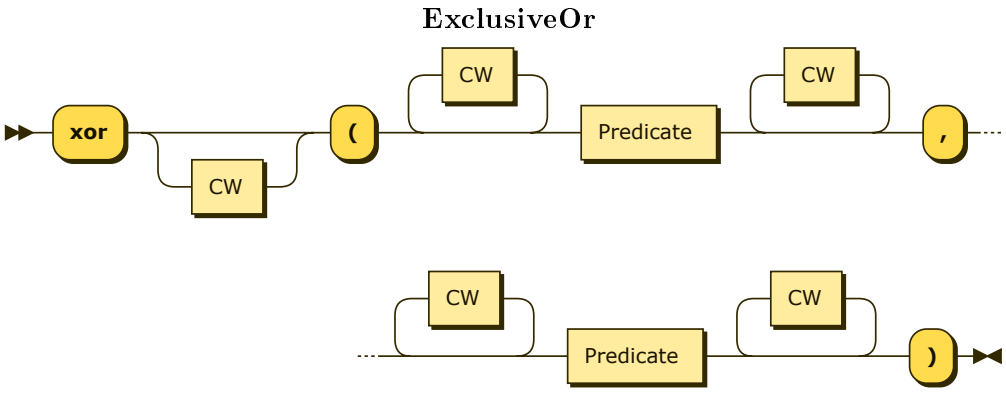


Figure 119: Terminals: `,`, `)`, `(`, `xor`; Non-terminals: Predicate, CW

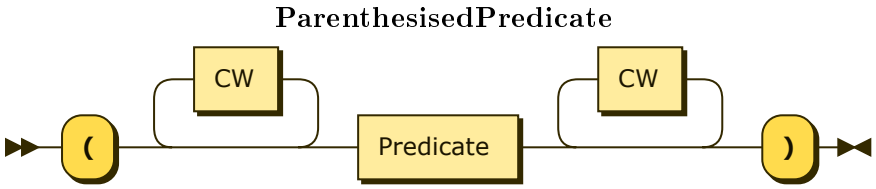


Figure 120: Terminals: `(`, `)`; Non-terminals: Predicate, CW

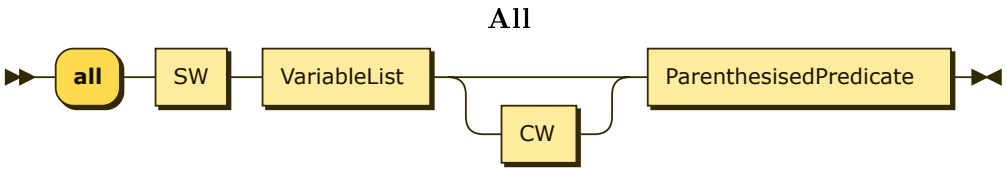


Figure 121: Terminals: `all`; Non-terminals: ParenthesisedPredicate, VariableList, SW, CW

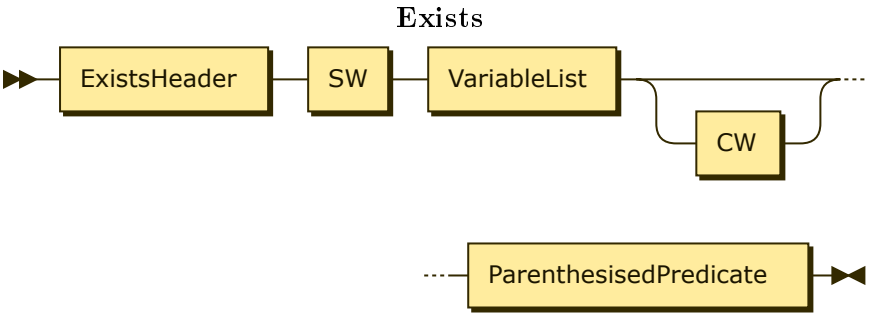


Figure 122: Non-terminals: VariableList, CW, ParenthesisedPredicate, SW, ExistsHeader

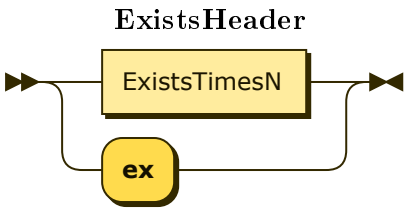


Figure 123: Terminals: `ex`; Non-terminals: ExistsTimesN

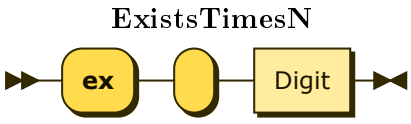


Figure 124: Terminals: `ex`; Non-terminals: Digit

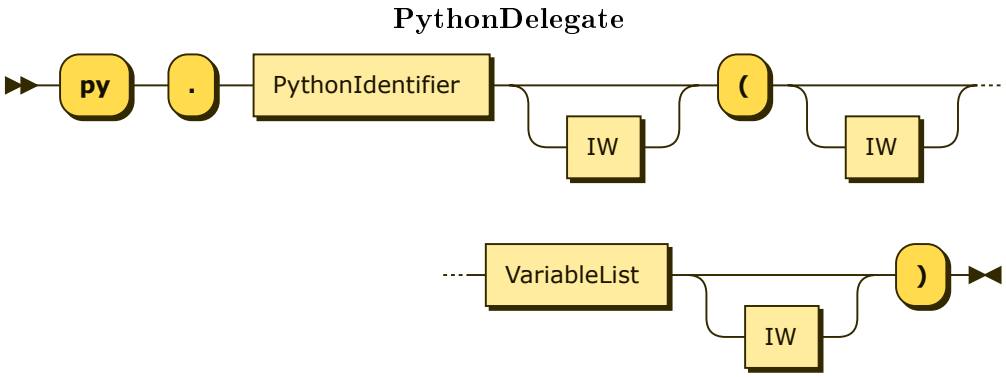
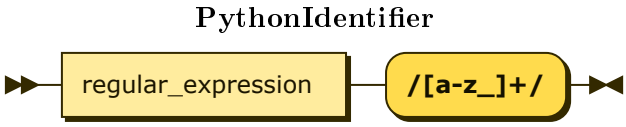


Figure 125: Terminals: `py`, `.`, `(`, `)`; Non-terminals: IW, VariableList, PythonIdentifier



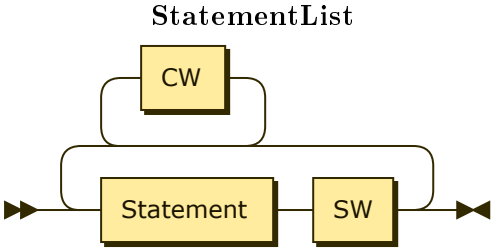


Figure 126: Non-terminals: SW, CW, Statement

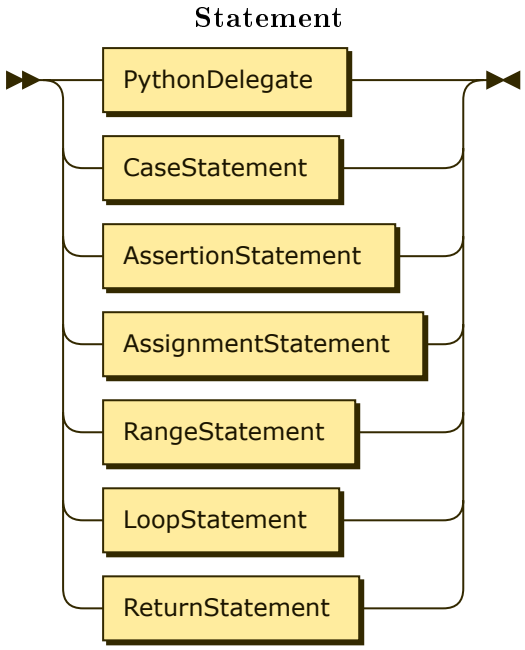


Figure 127: Non-terminals: AssertionStatement, RangeStatement, PythonDelegate, ReturnStatement, LoopStatement, CaseStatement, AssignmentStatement

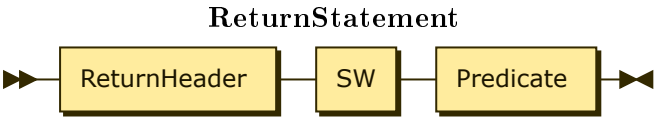


Figure 128: Non-terminals: Predicate, ReturnHeader, SW

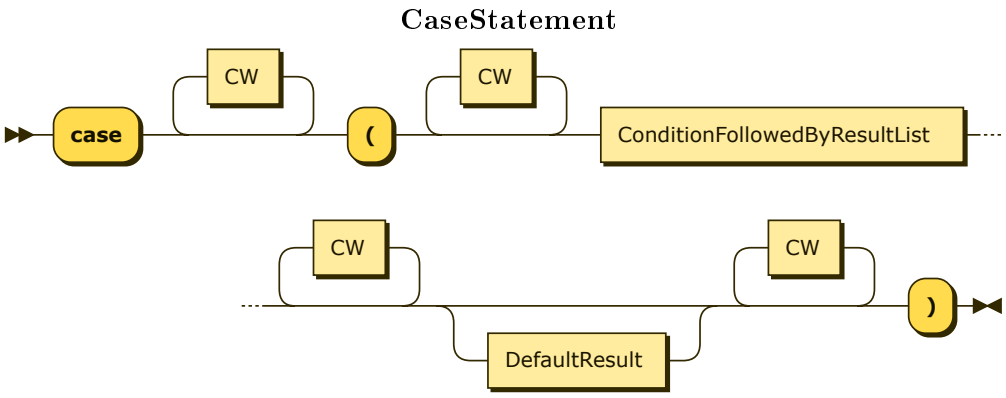


Figure 129: Terminals: `case`, `(`, `)`; Non-terminals: ConditionFollowedByResultList, DefaultResult, CW

LoopStatement

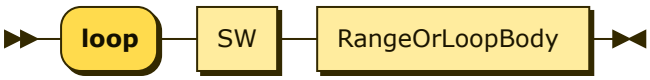


Figure 130: Terminals: `loop`; Non-terminals: RangeOrLoopBody, SW

RangeStatement

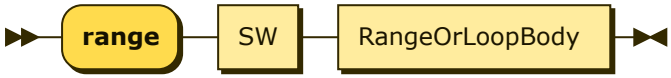


Figure 131: Terminals: `range`; Non-terminals: RangeOrLoopBody, SW

RangeOrLoopBody

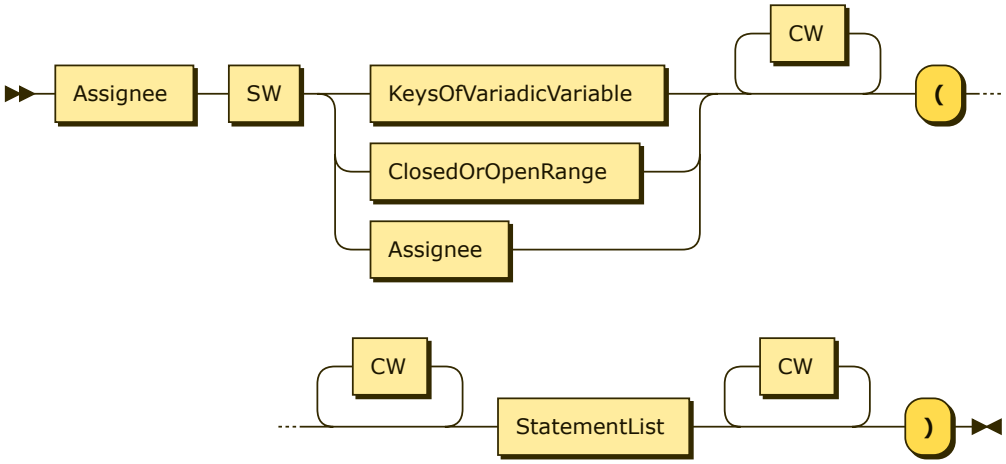


Figure 132: Terminals: `(`, `)`; Non-terminals: ClosedOrOpenRange, CW, KeysOfVariadic-Variable, SW, Assignee, StatementList

KeysOfVariadicVariable

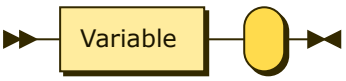


Figure 133: Non-terminals: Variable

ClosedOrOpenRange

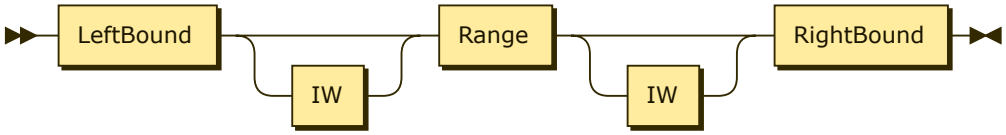


Figure 134: Non-terminals: IW, Range, RightBound, LeftBound

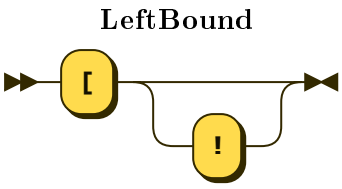


Figure 135: Terminals: `[`, `!`

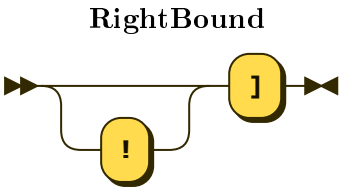


Figure 136: Terminals: `]`, `!`

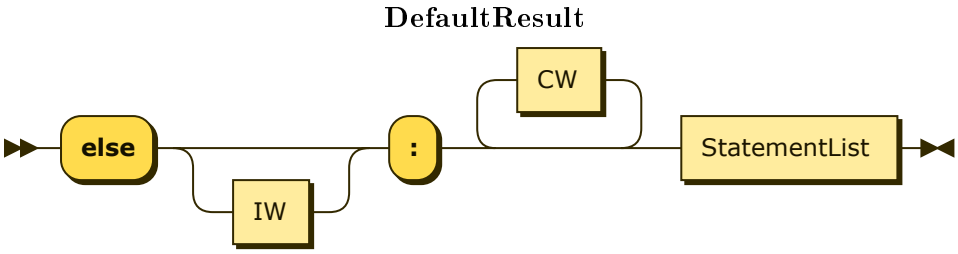


Figure 137: Terminals: `else`, `:`; Non-terminals: IW, CW, StatementList

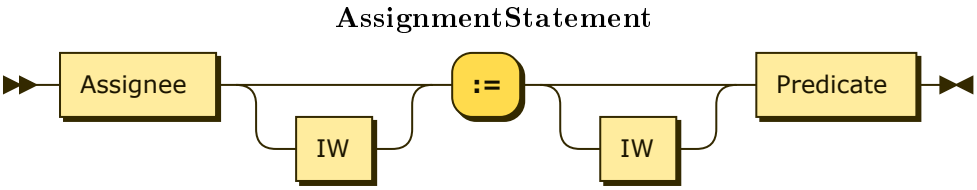


Figure 138: Terminals: `:=`; Non-terminals: Predicate, IW, Assignee

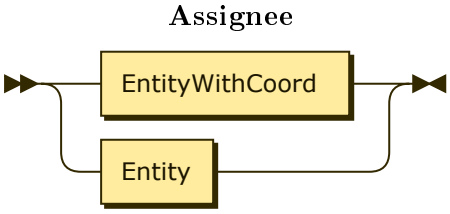


Figure 139: Non-terminals: Entity, EntityWithCoord

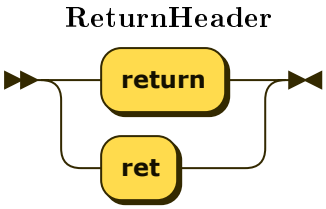


Figure 140: Terminals: `ret`, `return`

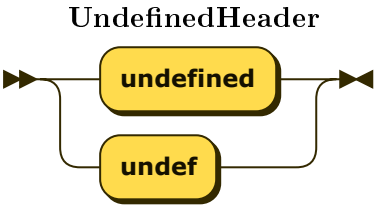


Figure 141: Terminals: `undefined`, `undef`

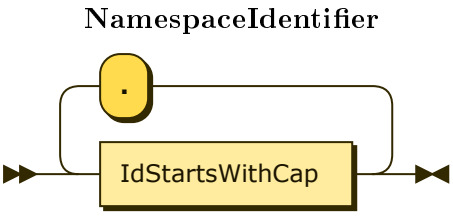


Figure 142: Terminals: `.`; Non-terminals: IdStartsWithCap

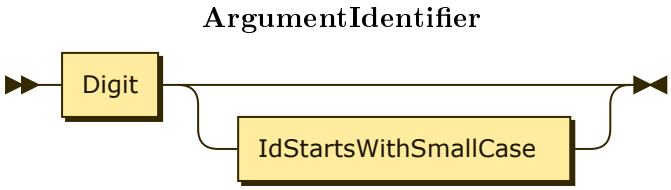


Figure 143: Non-terminals: Digit, IdStartsWithSmallCase

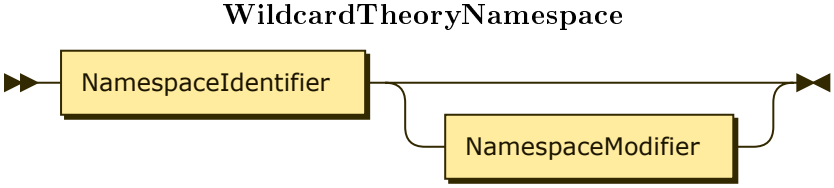


Figure 144: Non-terminals: NamespaceIdentifier, NamespaceModifier

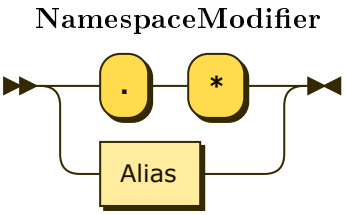


Figure 145: Terminals: `*`, `.`; Non-terminals: Alias

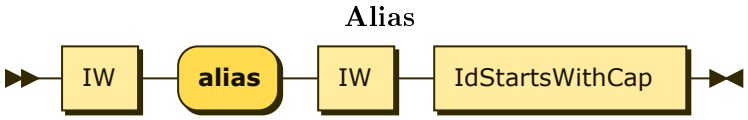


Figure 146: Terminals: `alias`; Non-terminals: IW, IdStartsWithCap

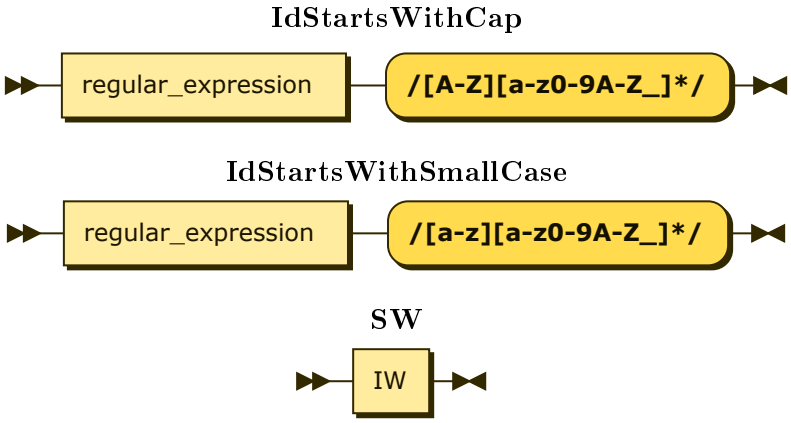


Figure 147: Non-terminals: IW

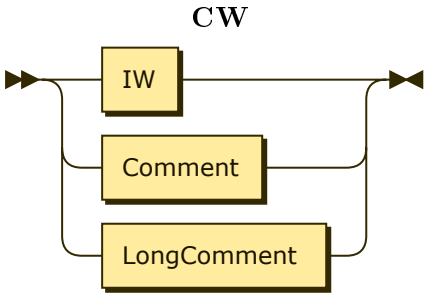
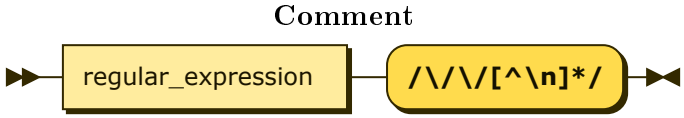


Figure 148: Non-terminals: IW, LongComment, Comment



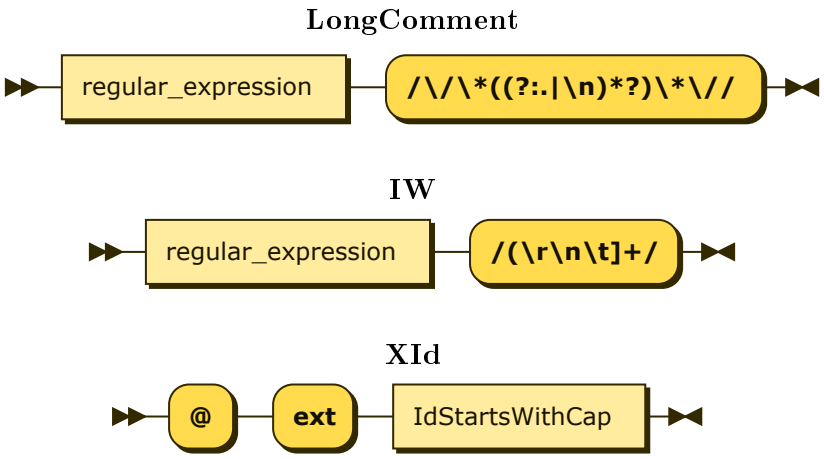


Figure 149: Terminals: `@`, `ext`; Non-terminals: IdStartsWithCap

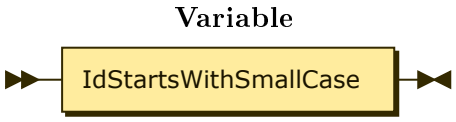


Figure 150: Non-terminals: IdStartsWithSmallCase

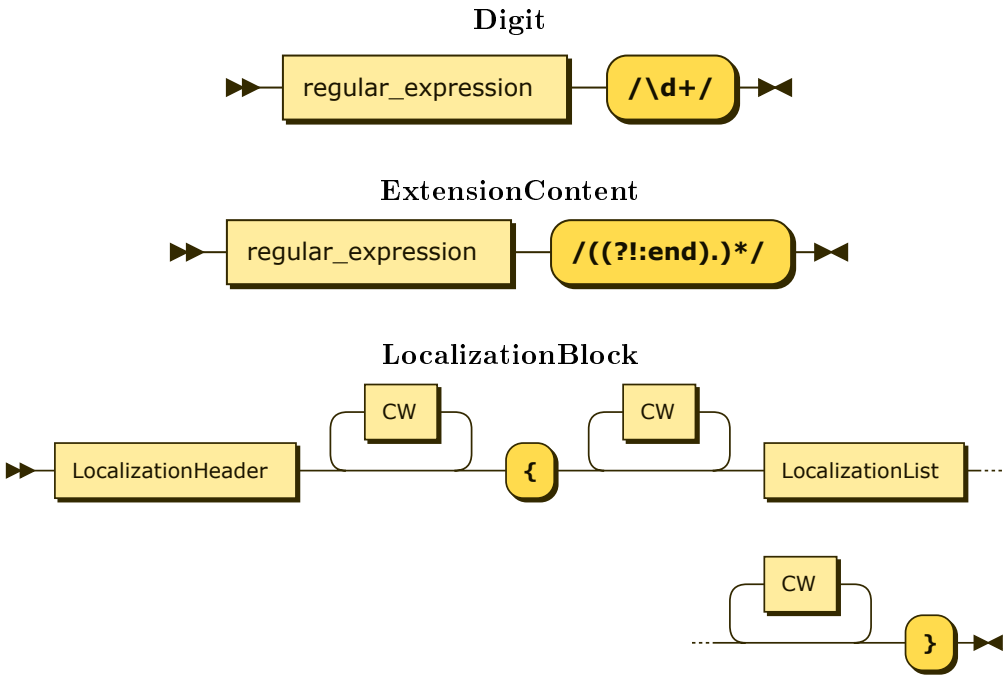


Figure 151: Terminals: `}`, `{`; Non-terminals: LocalizationList, CW, LocalizationHeader

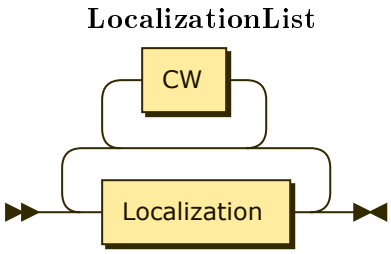


Figure 152: Non-terminals: CW, Localization

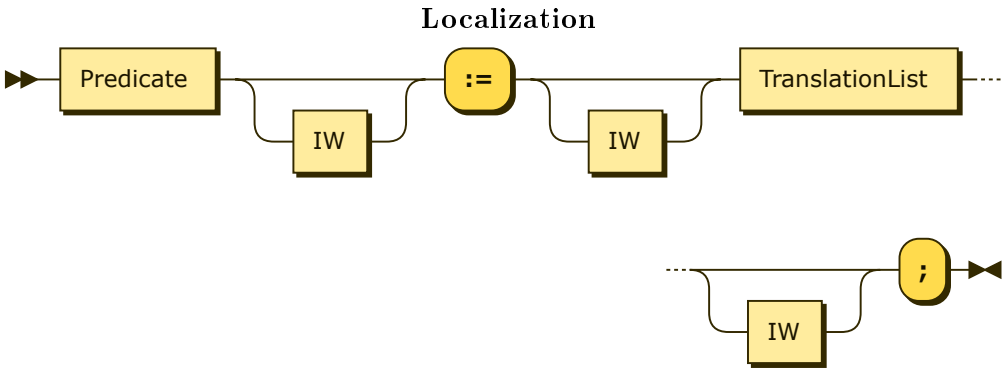


Figure 153: Terminals: >, :=; Non-terminals: Predicate, IW, TranslationList

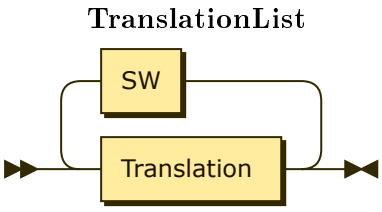


Figure 154: Non-terminals: Translation, SW

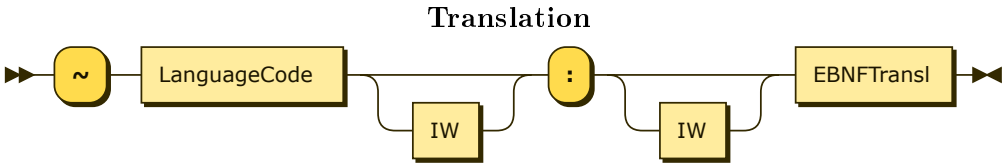
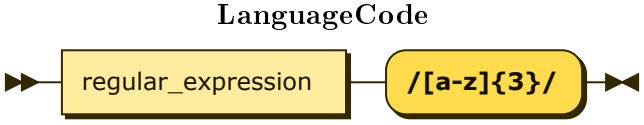


Figure 155: Terminals: :, ; Non-terminals: IW, EBNFTransl, LanguageCode



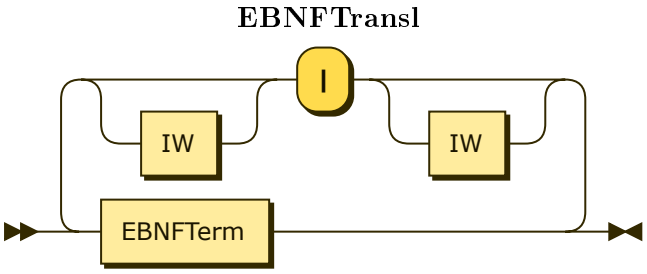


Figure 156: Terminals: `|`; Non-terminals: IW, EBNFTerm

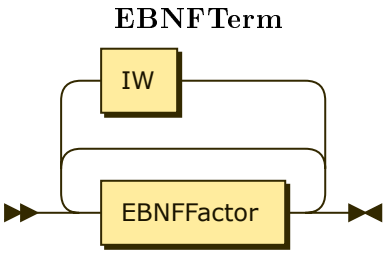


Figure 157: Non-terminals: EBNFFactor, IW

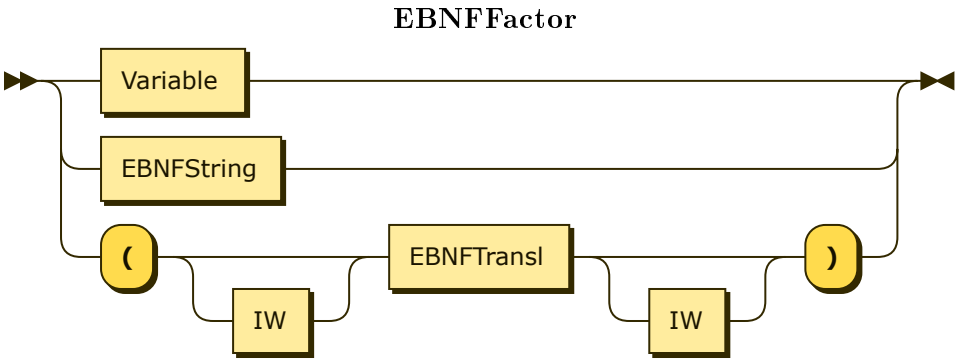


Figure 158: Terminals: `(`, `)`; Non-terminals: IW, EBNFString, Variable, EBNFTransl

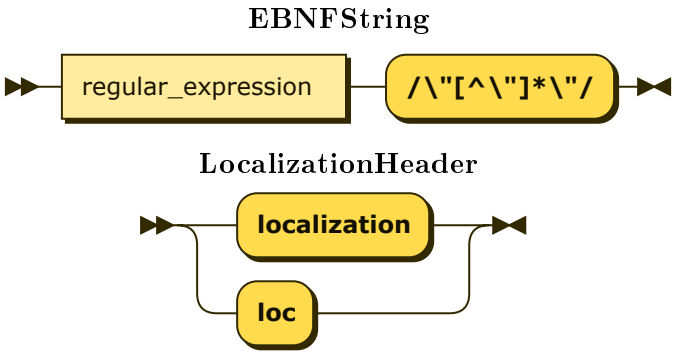


Figure 159: Terminals: `loc`, `localization`

extDigit

FPL Keywords

Concepts

Names

References

[1] K.P. Grotemeyer, *Topologie*, *B I Hochschultaschenbücher*, *Band 836*, 1969

[2] L.A. Steen, J.A. Seebach, Jr. *Counterexamples in Topology*, *Dover Publications, Inc.*, 1995

[3] Steve Awodey, *Category Theory*, *Oxford Logic Guides*, *2nd Ed.*, 2010

[4] C. Beierle, G. Kern-Isberner, *Methoden wissensbasierter Systeme*, *Vieweg*, 2000

[5] J.J. Buckley, E. Eslami, *An Introduction to Fuzzy Logic and Fuzzy Sets*, *Physica-Verlag*, 2002

[6] D. Cryan, S. Shatil, B. Mayblin, *Logic: A Graphic Guide*, *Icon Books Ltd.*, 2001 Stanford Encyclopedia of Philosophy, 2011

[7] H.-D. Ebbinghaus, *Einführung in die Mengenlehre*, *BI Wissenschaftsverlag*, 1994

[8] Rudolf Carnap, *Logical Syntax of Language*, *Psychology Press*, 1937

[9] Mohan Ganesalingam, *The Language of Mathematics*, *Springer*, 2013

[10] Thimothy Govers (Ed.), *The Princeton Companion to Mathematics*, *Orinceton University Press*, 2008

[11] Richard Knerr, *Knaurs Buch der Mathematik*, *Droemer Knaur*, 1989

[12] Reinhard Diestel, *Graph Theory*, *Springer*, 2000

[13] U.C. Merzbach, C.B. Boyer, *A History of Mathematics*, *Wiley*, 3rd. Ed., 2010

[14] Helmuth Gericke, *Mathematik in Antike, Orient und Abendland*, *fourierverlag*, 6rd. Ed., 2003

[15] Horst Hischer, *Grundlegende Begriffe der Mathematik: Entstehung und Entwicklung*, *Springer Spektrum*, 2012

[16] Immanuel Kant, *Kritik der reinen Vernunft, Werke 2*, *Könemann*, 11th Edition, 1995

[17] Translation by F.M. Müller of Immanual Kant's *Critique of Pure Reason*, *The Macmillan Company*, 1922,
`http://files.libertyfund.org/files/1442/0330_Bk.pdf`

[18] Harro Heuser, *Lehrbuch der Analysis, Teil 1*, *B. G. Teubner*, 11th Edition, 1994

[19] Dirk W. Hoffmann, *Theoretische Informatik*, *Hanser*, 2015

[20] Dirk W. Hoffmann, *Die Gödel'schen Unvollständigkeitssätze*, *Springer Spektrum*, 2013

[21] Dirk W. Hoffmann, *Forcing*, *Books on Demand*, 2018

[22] Dirk W. Hoffmann, *Grenzen der Mathematik*, *Spektrum Akademischer Verlag*, 2011

[23] Jonathan M. Kane, *Writing Proofs in Analysis*, *Springer*, 2016

[24] Ekkehard Kraetzel: "Studienbücherei Zahlentheorie", *VEB Deutscher Verlag der Wissenschaften*, 1981

[25] Richard Kohar, *Basic Discrete Mathematics, Logic, Set Theory, & Probability*, *World Scientific*, 2016

[26] Sheldon Axler, *Linear Algebra Done Right*, *Springer*, 3rd Edition, 2015

[27] Serge Lang, *Algebra*, *Springer*, 3rd Edition 2002

[28] Merzbach, Boyer, *A history of mathematics*, *Wiley*, 2011

[29] Bertram Maurer, *Mathematik, Die faszinierende Welt der Zahlen*, *NGV*, 2018

[30] J. Arbonés, P. Milrud, *Die Mathematik der Musik*, *Libero*, 2010

[31] Michael Meyer, *Entdecken und Begründen im Mathematikunterricht - zur Rolle der Abduktion und des Arguments (transl. from German "Discovering and Justifying in Mathematics Class - Role of Abduction and Arguments")*, *Journal für Mathematikdidaktik*, 28(2007)3/4

[32] Frédéric Mynard, *An Introduction to the Language of Mathematics*, *Springer*, 2018

[33] F. W. Lawvere, S. H. Schnanuel, *Conceptual Mathematics, A first introduction to categories*, *2nd Ed.*, *Cambridge*, 2009

[34] Hartshortn et. al., editor, *Collected Papers of C. Sanders Peirce*, *2nd Vol.*, *Harward University Press, Cambridge*, 1931

[35] G. H. Hardy, E. M. Wright, *An Introduction to the Theory of Numbers*, *4th Edition*, *Oxford at the Clarendon Press*, 1975

[36] Charles C. Pinter, *A Book of Set Theory*, *Dover Publications Inc.*, 2017

[37] Ulrich Knauer, *Diskrete Strukturen - kurz gefasst*, *Spektrum Akademischer Verlag*, 2001

[38] Otto Forster, *Analysis 1*, *vieweg Studium*, *4th Ed.* 1983

[39] Hermann Engesser (Hrsg.) *Informatik*, *Duden*, *2nd Ed.* 1993

[40] Karl Bosch, *Elementare Einführung in die Wahrscheinlichkeitsrechnung*, *vieweg studium*, *6th Ed.*, 1995

[41] Wolfgang Rautenmberg, *Einführung in die Mathematische Logik*, *Vieweg*, *2nd Ed.*, 2002

[42] F. Reinhardt, H. Soeder, *dtv-Atlas zur Mathematik*, *Deutscher Taschenbuch Verlag*, *10th Ed.*, 1994

[43] Thomas L. Heath Translation (Editor Data Densmore), *Euclid's Elements*, *Green Lion Press*, 2013

[44] Daniel N. Robinson, *The Great Ideas of Psychology, Lectures*, *The Teaching Company*, 1997

[45] H. D. Young, R. A. Freedmann, *University Physics with Modern Physics*, *Pearson*, 2016

[46] Klaus Jänich, *Topologie*, *Springer*, *7th Ed.*, 2001

[47] J.F. Nash (Jr.), M.Th. Rassias *Open Problems in Mathematics*, *Springer*, 2016

[48] J. Kramer, A.-M. von Pippich, *Von den natürlichen Zahlen zu den Quaternionen*, *Springer Spektrum*, 2013

[49] Florian Bechtold*Geometrie*, *Springer*, 2017

[50] John M. Lee*Axiomatic Geometry*, *American Mathematical Society*, 2013

[51] Max Koecher, *Lineare Algebra und analytische Geometrie*, *Springer*, *3rd Ed.*, 1992

[52] Detlef D. Spalt, *Eine kurze Geschichte der Analysis*, *Springer Spektrum*, 2017

[53] Hans Wußing, *Vorlesungen zur Geschichte der Mathematik*, *VEB Deutscher Verlag der Wissenschaften*, 1979

[54] Dirk. J. Struik, *Abriß der Geschichte der Mathematik*, *VEB Deutscher Verlag der Wissenschaften*, 1976

[55] Max Tegmark, *Our Mathematical Universe*, *Penguin Books*, 2014

[56] John Stillwell, *Reverse Mathematics*, *Princeton University Press*, 2018

[57] H.-P. Tuschik, H. Wolter, *Mathematische Logik - Kurzgefasst*, *BI Wissenschaftsverlag*, 1993

[58] M. Wenzel et al.*The Isabelle/Isar Reference Manual*, `https://isabelle.in.tum.de/dist/Isabelle2020/doc/isar-ref.pdf`, 2020

[59] Bruno Buchberger *Theorema Project*, https://risc.jku.at/pj/theorema-project/, 2020

[60] In BookOfProofs, *Euclid's Elements*, https://www.bookofproofs.org/branches/euclids-elements/, 300 BC

[61] In BookOfProofs, *Zermelo-Fraenkel Axioms*, https://www.bookofproofs.org/branches/zermelo-fraenkel-axioms/, 2018

[62] E. Zermelo, *Collected Works*, https://epdf.pub/mengenlehre-varia-schriften-der-mathematisch-naturwissenschaftlichen-wissenschaf.html, 2010

[63] In BookOfProofs, *Putting It All Together - Syntax and Semantics of a Logical Calculus*, https://www.bookofproofs.org/branches/putting-it-all-together-syntax-and-semantics-of-a-logical-calculus/, 2018

[64] Online resource: Book Of Proofs https://www.bookofproofs.org/

[65] Online resource ProofWiki https://proofwiki.org/

[66] Ross Cameron, *Infinite Regress Arguments*, https://plato.stanford.edu/entries/infinite-regress/, 2018

[67] In Wikipedia. *Five Ws*, https://en.wikipedia.org/wiki/Five_Ws/, 2021

[68] Creative Commons, *CC BY-SA 4.0*, https://creativecommons.org/licenses/ by-sa/4.0/, 2019

[69] Coq Developers *The Coq Proof Assistant*, https://coq.inria.fr/, 2020

[70] J. Corneli et al. *Modelling the Way Mathematics Is Actually Done*, http://www.newton.ac.uk/files/ preprints/ni17003.pdf, 2017

[71] Marcos Cramer, *The Naproche system: proof-checking mathematical texts in controlled natural language*, https://korpora-exp.zim.uni-duisburg-essen.de/naproche/downloads/2014/SDV.pdf 2014

[72] Mary Domski, *Descartes' Mathematics*, https://plato.stanford.edu/entries/descartes-mathematics/,

[73] *The Association for Logic, Language and Information*, http://www.folli.info/,

[74] M. Ganesalingam, W. T. Gowers *A Fully Automatic Theorem Prover with Human-Style Output*, https://link.springer.com/article/10.1007/s10817-016-9377-1, 2016

[75] Herman Geuvers *Proof assistants: History, ideas and future*, https://www.ias.ac.in/article/fulltext/sadh/034/01/0003-0025, 2009

[76] Siegfried Gottwald, *Many-Valued Logic*, https://plato.stanford.edu/entries/logic-manyvalued/, 2015

[77] David Hilbert *The Foundations of Geometry*, Transl. from German original from 1899 *"Grundlagen der Geometrie"* to English by E.J. Townsend https://math.berkeley.edu/∼wodzicki/160/Hilbert.pdf, 1950

[78] Leon Horsten, *Philosophy of Mathematics*, https://plato.stanford.edu/entries/philosophy-mathematics/, 2017

[79] Isaac Newton Institute *Big proof Programme*, http://www.newton.ac.uk/event/bpr, 2017

[80] Isabelle Developers *Isabelle*, https://isabelle.in.tum.de/, 2020

[81] Leslie Lamport *The TLA+ Home Page*, https://lamport.azurewebsites.net/ tla/tla.html, 2019

[82] Leslie Lamport *How to Write a 21st Century Proof*, https://lamport.azurewebsites.net/pubs/proof.pdf, 2011

[83] Mizar Developers*Mizar Project*, http://mizar.org/project/, 2020

[84] Arnold Neumaier *FMathL Project*, https://www.mat.univie.ac.at/∼neum/FMathL.html since 2017

[85] Arnold Neumaier *The FMathL mathematical framework*, https://www.mat.univie.ac.at/∼neum/ms/fmathl.pdf, 2009

[86] A. Neumaier, University of Vienna *Systems related to the FMathL vision*, https://www.mat.univie.ac.at/∼neum/FMathL/Related.pdf, 2010

[87] OED *Retrieved August 17, 2020*, https://www.oxfordlearners dictionaries.com/ definition/english/, 2020

[88] Aarne Ranta, *Structures grammaticales dans le français mathématique : II - (suite et fin), Mathématiques et sciences humaines, tome 139 (1997), p. 5-36*, http://archive.numdam.org/article/ MSH_1997__139__5_0.pdf, 1994

[89] Bradner, *RFC 2119*, https://www.ietf.org/ rfc/rfc2119.txt, 1997

[90] Anonymous *The QED Manifesto*, http://www.cs.ru.nl/∼freek/qed/qed.html, 1994

[91] Goodreads *Albert Einstein Quotes, retrieved August 16, 2020*, https://www.goodreads.com/author/quotes/ 9810.Albert_Einstein, 2020

[92] McMaster University *The Bertrand Russell Archives, retrieved August 16, 2020*, https://www.mcmaster.ca/russdocs/brquotes.htm, 2014

[93] *Pascal Case*, http://wiki.c2.com/?PascalCase, 2014

[94] Chris Drew, *Shannon Weaver Model of Communication*, https://helpfulprofessor.com/shannon-weaver-model/, 2020

[95] In Wikipedia. *Retrieved August 13, 2020*, https://en.wikipedia.org/wiki/Linear_code, 2020

[96] James Young, *The Coherence Theory of Truth*, https://plato.stanford.edu/ entries/truth-coherence/, 2018

[97] Eugene Wigner, *The Unreasonable Effectiveness of Mathematics in the Natural Science*, http://www.dartmouth.edu/∼matc/MathDrama/reading/Wigner.html, 1960

[98] J. C. Villanueva, *How Many Atoms Are There in the Universe?*, https://www.universetoday.com/36302/atoms-in-the-universe, 2009

[99] Avery Thomson, *In 1928, One Physicist Accidentally Predicted Antimatter*, https://www.popularmechanics.com/science/a27049/in-1928-one-physicist-accidentally -predicted-antimatter/, 2017

[100] In Wikipedia. *Retrieved August 26, 2020*, https://en.wikipedia.org/wiki/Extended_Backus, 2020

[101] Andrzej Indrzejczak, University of Lodz *Natural Deduction*, https://iep.utm.edu/nat-ded/, 2020

[102] W.N. Molodschi, *"Studien zu philosophischen Problemen der Mathematik"*, *VEB Deutscher Verlag der Wissenschaften*, 1977

[103] Vaibhav Kumar Rai, GeeksforGeeks *Rules of Inference*, https://www.geeksforgeeks.org/mathematical-logic-rules-inference/, 2020

[104] Patrick Devine, *Case Styles: Camel, Pascal, Snake, and Kebab* https://betterprogramming.pub/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841, 2018

[105] In w3schools.com. *C# Tutorial*, https://www.w3schools.com/cs/default.asp, 2020

[106] Michael Lahanas *Pythagoras: The whole thing is a number*, http://www.hellenicaworld.com/Greece/Science/en/PythagorasNumber.html, 2020

[107] In BookOfProofs, *Building the Successors of Ordinal Numbers*, https://www.bookofproofs.org/branches/building-the-successors-of-ordinal-numbers/, 2020

[108] In BookOfProofs, *FPL - the Formal Proving Language*, https://www.bookofproofs.org/branches/fpl-formal-proving-language/, 2020

[109] C. Cohen-Tannoudji, B. Fiu, F. Laloë, *Quantum Mechanics*, https://personal.utdallas.edu/∼son051000/comp/postulates.pdf, 2020

[110] In *C# Reference*, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface, 2020

[111] In *C# Reference*, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/, 2020

[112] Alexander Bogomolny *Is Mathematics Beautiful*, http://www.cut-the-knot.org/manifesto/beauty.shtml/, 2018

[113] Allesandro Languasco *Efficient computation of the Euler–Kronecker constants of prime cyclotomic fields. Research in Number Theory*, https://www.researchgate.net/publication/331728655_Efficient_computation_of_the_Euler-Kronecker_constants_of_prime_cyclotomic_fields, 2021

[114] Antal Járai, K.-H. Indlekofer *Largest known twin primes and Sophie Germain primes*, https://www.researchgate.net/publication/220576589_Largest_known_twin_primes_and_Sophie_Germain_primes, 1999