Analysis of a parsing algorithm suitable for parsing ambiguous mathematical text

Final Year Dissertation

Author: Luka Vrečar

BSc Mathematics and Computer Science, Year 4

Supervisor: Dr Joe Wells

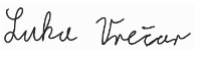
2nd reader: Prof Mike Chantler

Heriot-Watt University, Edinburgh

April 20, 2022

Declaration

I, Luka Vrečar, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included. [1]

Signed: 

Date: April 20, 2022

[1] This declaration was taken from Appendix C.1 in the Project Handbook. Everything apart from my name is copied word for word.

Abstract

This dissertation presents a formalisation framework and the beginnings of a formalism of DynGenPar [21–23], a parser that is potentially suitable for parsing mathematical text. This dissertation also identifies and discusses some of the challenges that mathematical text poses to traditional parsers, namely the dynamic nature of their grammar, and the ambiguities which are present on all levels of mathematical text. DynGenPar claims to be an exhaustive parser (and as such produces all valid parse trees for a given input), which can also handle dynamic rule addition. This means it would be able to add rules to its grammar during parsing, which is crucial when parsing mathematical text, and expensive with other existing parsers like LR [20], GLR [39], or LL [36], which rely on pre-compiled tables that need to be recomputed when the underlying grammar changes.

In my endeavours to formalise DynGenPar, I also managed to produce a minimal version of the original implementation, essentially removing all the additional functionality that Kofler implemented to make DynGenPar more usable in practice. Those features also need formalisation and verification in order for the original implementation of DynGenPar to be used for parsing ambiguous mathematical text.

Contents

Chapter 1

Introduction

The digital representation of mathematics has been an effort of researchers for many years. While there are many different options for representing human-written mathematics digitally to make it understandable by humans, such as LaTeX for publications and presentation MathML for web browsers, we are yet to create a widely used system to make human-written mathematics (and mathematics in general) understandable by machines[1]. Here, we mean a system for computer representation of mathematical documents. A computer system could parse these documents unambiguously and with the correct meaning[2], and then check it for logical soundness, or build new mathematics on top of that. We will refer to this as *computerising mathematics*. Computerising mathematics poses a particular challenge for multiple reasons.

One reason is that, as reported in the findings of the Semantic Representation of Mathematical Knowledge Workshop [18], mathematicians have not yet agreed as to what such a system would be capable of or how to implement it. Even the discussion on what foundation of mathematics is de facto the best to represent mathematics (in publications, let alone digitally) is ongoing (and beyond the scope of this project).

Several additional reasons why the computerisation of mathematics is a particularly challenging problem can also be found in the language of mathematics. It has been the topic of detailed discussion, by e.g., Mohan Ganesalingam in his PhD thesis and subsequent book based on it [11], and in Mihnea Iancu's PhD thesis, Towards Flexiformal Mathematics [17]. Both works highlight a number of linguistic phenomena found in the language of mathematics.

[1] Systems (which will be discussed later on) already exist to represent human-written mathematics in a format understandable by machines, but their input and output languages are not necessarily similar to that of human-written mathematics.

[2] The correct meaning is the meaning intended by the author of a mathematical document.

The final reason we will mention here is the scale of the problem of representing mathematics digitally. Making a computer system that understands arbitrary mathematics, can potentially reason about it and even produce new mathematics is difficult, due to the many parts required. It would need, among other things, a representational language for both input and output, a parsing mechanism that can handle on the spot additions to its grammar, a disambiguation component, some form of reasoning capability, and a database of mathematical knowledge the system has encountered already. This dissertation focuses on the problem of unambiguous representation of mathematical texts, which is a part of a field of research known as mathematical knowledge management.

In the beginning, the goal was to parse a specific example of ambiguous mathematical text. However, during the background research of the topic and shortly after, it became clear that the state of the art of mathematical language representation and parsing software does not enable us to achieve said goal. Nevertheless, we can make good progress toward improving the state of the art, which this dissertation does. Below is a brief summary of what was achieved in the scope of this dissertation, to improve the state of the art of parsing software which is necessary both for the short term goal of parsing the aforementioned example, and the long term goal of computerising mathematics.

1.1 Summary of contributions

1. In Section 2.3.6 I determined that Ganesalingam's algorithm does not exist in any sort of implementation.

2. In Section 2.3.2 I identified DynGenPar, a part of Concise [8], which could be used, as it handles ambiguities and allows for dynamic grammar rule addition.

3. In Section 3.2, I explained why Concise would not be useful as an interface with DynGenPar.

4. In Section 4.1.3, I figured out that the description of DynGenPar found in the literature does not correspond with the C++ implementation and that there is no formal description of either.

5. In Section 4.1.3, I also found a significant difference between the non-deterministic DynGenPar algorithm described in the literature, and the C++ implementation, which con-

currently makes all the non-deterministic choices at once. I realised that the details of this concurrent execution and the fact that it even occurs were omitted from the papers written about DynGenPar [21–23].

6. In Section 4.1.3, I also identified the need to produce a formal description of DynGenPar in order to be able to reason about it and make claims about what it does. The closest thing to a formal specification before my work was the description found in the literature [21–23], which, as mentioned, turned out not to be what the implementation does.

7. I realised formalising the entire algorithm would take too much time, so I tried to find anything that could be removed.

8. I hypothesised that there is a core of the C++ implementation that could be isolated, understood, and formalised. In Section 4.4, I describe how I successfully extracted a minimal core of the original DynGenPar implementation that could still parse ambiguous context-free grammars.

9. In Section 4.2.1, I presented a framework within which I could then write a formal specification of the DynGenPar algorithm.

10. In Section 4.2.2, I worked out and presented many of the details of the formal specification of the DynGenPar algorithm.

11. In Section 4.4, I created a list of everything that I removed from the original DynGenPar implementation to extract a minimal core.

12. I created some example applications (see Section 4.5) and a test suite (see Section 4.3) for understanding DynGenPar.

13. I worked out a parsing of an example input with a very simple grammar by hand and presented it in Section 4.2.4, in order to help illustrate how the algorithm works.

1.2 Document structure

The document begins with a review of relevant literature and how it ties into the project in Chapter 2. Some related work using different approaches to disambiguate mathematical text is also mentioned. Following the literature review, Chapter 3 contains an assessment of two pieces

of software encountered while researching the background of the problem. Chapter 4 contains a partial formalism of the DynGenPar algorithm, and a framework used to build the formalism. Furthermore, it includes a detailed computation of a small example using the DynGenPar algorithm, and details how a core was extracted from the implementation. In Chapter 5 the main achievements and what is left to future work is listed.

Chapter 2

Background

The language of mathematics has been studied for many years. While interesting purely from a linguistic perspective, a motivation for studying mathematics has also been the automation of mathematical research. Rigorous computation (especially when several cases need to be considered to complete a proof), formalisation of proofs, and checking of logical soundness are difficult and prone to errors along the way when done by a human. Computerising the process could yield desired results (namely, formalised proofs and a verification of logicalness) faster and with fewer errors, thus increasing the quality of research produced.

A different, more ambitious goal is autonomous reasoning. By autonomous reasoning, we mean a computer system "thinking" about mathematics, given a collection of knowledge and presented with either a problem to be solved, or a new piece of mathematical knowledge. Such a system would emulate the way humans think about mathematics when applying their knowledge to problems or reading a mathematical document they have not encountered before. Without true artificial intelligence, this is likely only possible using systems specialised for mathematics, and in a limited capacity[1]. There are multiple reasons why that might be beneficial, and aside from the increased quality mentioned before, we should point out the time and effort it would save researchers. It takes humans years of study to understand mathematics well enough to reason about it efficiently, so creating a computer system which could learn mathematics and reason about it would be beneficial. Such a system could work faster, around the clock (humans need to eat and sleep whereas a computer system does not), and with fewer mistakes, which would speed up mathematical research greatly as well.

A goal which will not be further discussed in this document is of a more philosophical nature.

[1] We refer here to the areas of mathematics such a system could cover.

The study and understanding of the language of mathematics could answer some questions from philosophy and perhaps even psychology, such as "What objects are we referring to when we say 'the number 3'?", "What does the ability to read and write mathematical language tell us about the human intelligence and ability to reason?", "Why are some mathematical texts considered more 'beautiful' or 'elegant' than others?", and numerous others.

The final motivation to study the language of mathematics we will mention here is an ambitious one. In the last century, the output of published mathematics has grown enormously.[2] To understand all of it and see how it all connects, we would first have to read through all of it, which is a task too big for any human or group of humans. A computer system could do it, since it can read faster, more precisely, and around the clock, but computer systems do not understand mathematics well enough to read through the centuries of mathematical texts that currently exist. There is potential, though, for a system to read, digitise and explain any past, present, and future piece of mathematical writing. In order to create such a system, we need to ensure it understands mathematical documents in the way it was intended by their authors, and that requires precise and in-depth knowledge of the language(s) of mathematics. We use the plural of "language" here, since mathematics is being published in many languages (although this project focuses only on English), and there are differences in mathematical convention (among other things like grammar and vocabulary) between them. Regardless of which natural language we use to represent it, the language of mathematics is a source of several phenomena we will discuss now.

2.1 Linguistic phenomena in the language of mathematics

The language of mathematics contains many linguistic phenomena that make parsing it different from natural languages. Ganesalingam [11] and Iancu [17], among others, discuss these phenomena in great detail. We will follow the classification outlined by Iancu [17]. Iancu also notes an important phenomenon that appears at all levels of mathematical language: varying formality. The term *flexiformality* is used for this phenomenon. This term originates from Kohlhase's work on the Flexiformalist Manifesto [25]. Kohlhase opens this manifesto with an observation, that while formalising mathematics is vital for computer support[3], mathematical

[2] Roughly 360 000 mathematical publications were written in 2020 alone compared to roughly 75 000 in 1996 (the oldest statistic available at the same source) [37].

[3] Proof checking systems require fully formalised documents, while other computer systems for mathematics might not.

texts are rarely fully formalised.

Kohlhase notes later on, that some computer systems for mathematics (such as screen readers or formula search tools) do not require fully formalised mathematical documents to function correctly. Kohlhase proposes a flexiformal system of representing mathematics, where a document can be entirely informal, fully formalised, or have a degree of formality somewhere in between (but still be accurately represented in the flexiformal system). This would be achieved using step-by-step formalisation, allowing for formalisation to stop at any time when the degree of formality of a document is good enough for what we wish to do with it. An illustrative example of functionality in terms of formality of documents for various types of computer tools is shown. The desire for a digital library of flexiformal mathematical knowledge is also introduced as a key point. The discussion of the concept of flexiformality in detail is beyond the scope of this dissertation, so we refer the reader to the Flexiformalist Manifesto [25] for further reading.

With the phenomenon of varying formality, which is present on all levels of text, let us now focus on phrases and sentences, some of the smallest building blocks of mathematical documents.

2.1.1 Phrasal level

On the phrasal level of mathematical texts, the main issues and phenomena arise from the use of symbols and formulas when writing mathematics. Text and formulas are intertwined within documents, and sometimes even within the same sentence. For example: "Let $x \in \mathbb{R}$ be positive..." or "$P = \{n \mid n \text{ is prime}\}$". A computer system for mathematics must handle both textual and symbolic mathematics correctly separately and, as seen in these examples and Ganesalingam's book [11], interdependently as well. Another occurrence of formulas in text which can be difficult to understand for a computer system, is when formulas replace parts of speech. For example, in "Let $x \in \mathbb{R}$ be positive...", the formula $x \in \mathbb{R}$, which is a statement, serves as a noun phrase in place of "some x such that x is a real number".

Formulas could conceivably be removed through a process known as *verbalisation*. Verbalisation is a term used by Iancu [17] to refer to the process of assigning natural language representation to mathematical objects, for example "plus" for $+$ and "Euler's number" for e. Since formulas make texts more concise and therefore (in general) easier to read, they cannot be removed without making documents harder to process for human readers. As noted by Ganesalingam [11], at least variables must be left in, since anaphoric referencing can become unwieldy for human readers. An anaphor is "a word or phrase that refers to a word or phrase

used earlier in a text and replaces it, for example the word 'it' in the sentence 'Joe dropped a glass and it broke'"[4]

On the other side of verbalisation is *notation*. Notation is a symbol or combination of symbols that encodes a formal mathematical object, for example $\int$, $\cup$, π, etc. One issue with notation is its overloading. The same symbol can mean an entirely different thing in different fields of mathematics. For example, $\times$, which usually denotes multiplication, denotes the vector product in the context of vectors, while denoting the Cartesian product in the context of sets. The same can happen with verbalisation, where "prime" has a different meaning in number theory, where it refers to prime numbers, than it does in calculus, where f', read as "f prime", denotes the first derivative of the function f. It can also introduce ambiguity, especially with implicit notation like multiplication is in most cases[5].

Another source of ambiguity with notation is complex notation such as $\sum$, lim, etc. They introduce a particular type of ambiguity called *elision*, which refers to ambiguity as a consequence of omitting arguments. There are multiple types of elision, which pose different challenges. For example, when talking about a sequence, say $a_1, a_2, ..., a_k$, we often see the sum of this sequence denoted as $\sum a_i$, instead of $\sum_{i=1}^{k} a_i$. Another example, which arguably poses an even greater challenge, is the use of ... when listing items, like in $\mathbb{N} = \{1, 2, 3, ...\}$. A human reader can usually infer the precise meaning of such notation based on previous experience and knowledge, and a computer system could conceivably do the same. An additional challenge is also that the meaning of the omitted notation sometimes differs based on context. For example, $\log n$ can mean $\log_{10} n$ in one case, $\log_2 n$ in another, and $\log_e n = \ln n$ again in a third case.

Another issue is the naming of mathematical objects locally. That can be, for example, variables which are used in a definition, theorem, proof, etc. Sometimes, these locally bound objects can be used across multiple sentences and their binders can either be symbolic ("...$\forall x$...") or textual ("...for all x..."). Often, restrictions on them are introduced, sometimes even after the object itself ("Let the following hold for all x, such that $x > 0$..."). All of this poses a challenge for a computer system reading mathematics, as it needs some kind of memory of what these bound objects are, their properties, and when to "forget" about them. The main

[4]Everything between the double quotation marks is taken word for word from the online version of Cambridge Dictionary [1].The only change are the single quotation marks within the copied text, which were changed from double quotation marks, for clarity.

[5]For example, $f(a(b+c))$ could be understood as the function f applied to the product of a and $b+c$, or as f applied to the function a applied to $b+c$. Context would play a crucial role in automatic disambiguation here, but human readers usually disambiguate this based on the statistical fact that f is most often used to denote functions, while a, b, and c are usually used to denote numbers.

difficulty, however, is figuring out the binding structure, i.e., what the bindings refer to. It may seem relatively straightforward to a human reader, but certain not too complex sentences can already pose a problem, since the bindings can often "leak" into the next few sentences. Therefore, just parsing mathematical text sentence by sentence without keeping track of the broader context is impossible. Iancu [17] provides several examples of this, but we will only list two, to help illustrate this "leaking":

- Let $x \in G$. The element y such that $xy = yx = e$ is uniquely determined.
- If $n > 10$ is prime, then it is odd. Moreover, n ends in one of the digits 1, 3, 7 or 9.

Here, we can see that x and n are referred to in the second sentence. However, to understand what they mean, the previous sentence must be parsed and remembered. The second example poses an additional challenge due to an anaphoric reference (highlighted in bold) to n ("... then **it** is odd."). This can be relatively straightforward to resolve and determine that the "it" refers to n. However, the example can further be complicated by introducing an anaphoric reference in the second sentence as well. That can be done by rewriting the example to be "If $n > 10$ is prime, then it is odd. Moreover, **it** ends in one of the digits 1, 3, 7 or 9". Such referencing can make figuring out the binding of mathematical objects difficult and is a potential source of ambiguity.

A phenomenon that could potentially be seen as an advantage to natural language is the fact that every new notion or piece of notation in mathematics is defined precisely, and that is done in clearly marked parts of the document (see below). When such a definition is encountered, a computer system could learn it and remember it for later. These on-the-spot additions and their handling are discussed by Kofler in his PhD thesis [21], and also by Ganesalingam in his book [11].

2.1.2 Discourse structure

Mathematical language and its documents introduce specific passages (known as "blocks") for important pieces of text, such as definitions, theorems, lemmas, and proofs. Within a document, they are often denoted by special typesetting and sometimes even numbered or named[6] (e.g., "Definition 7.2.", or "Mean Value Theorem"). A special case here are definitions, which

[6]This is usually done to make referring to specific definitions and theorems later on easier. Sometimes, with named theorems, it can even happen that the theorem is nowhere in the document but is still referred to. This is mostly the case with well-known theorems.

are specified in mathematical language in ways that are rarely used elsewhere.[7] Definitions introduce new concepts and notation and usually build on previous knowledge (either from the same document, or other documents).

Definitions

A definition block is particularly useful, since it clearly marks the beginning and ending of a definition. When parsing text with a computer system, we need to update the parsing grammar when a definition is encountered, as it introduces concepts and notation the system might not have encountered yet. Knowing precisely where in the text to start and stop searching for new grammar rules simplifies this process greatly. Here is an example of a definition block, taken from the Linear Algebra lecture notes by Dr Alexandre Martin[8] [27]:

Definition 1.7. An augmented matrix is in *(row) echelon form* if the first non-zero entry in a row (the *pivot* of that row) appears strictly to the right of the pivots of the previous rows.

As is often the case in mathematical definition, the concept being defined or named is further specially typeset (usually with bold or, as is also the case here, italic case). If a computer system were reading these lecture notes, it would have to parse the definition and remember it, since almost immediately after it, the following sentence is encountered:

"From the row echelon form, it is straightforward to see whether a system of linear equations is consistent." [27]

Furthermore, this one-sentence definition actually defines two concepts, *(row) echelon form*, which is the main concept being defined, and *pivot*, which is essentially mentioned on the side just to inform the reader that a specific name is given to "the first non-zero entry in a row of a matrix". The word pivot is then immediately used in the same sentence to help define the concept of "(row) echelon form". Such (grammatically) nested definitions (i.e., definitions inside other definitions) are commonplace in mathematical language.

We see that parsing the definitions encountered in mathematical texts immediately is crucial for the understanding of subsequent text. As seen in the example, definitions can also be nested which must be taken into account when parsing them.

[7] An example of other texts which use precise definitions would be legal documents, and in particular, laws. There, precise definitions are key to eliminate multiple interpretations of the same text.

[8] These notes are quoted with the authors permission.

Assertion blocks

The other important sections, namely theorems, lemmas, proofs, and corollaries are implicitly interconnected. For example, the word "theorem" implies that a proof exists (and could potentially be encountered later in the document), a corollary arises as a consequence of a theorem, and lemmas can help with proofs. Here we will also mention conjectures and axioms. They are often present in mathematical documents, but a proof is not expected. A conjecture is essentially a theorem, which has not yet been proven (or disproven!). Axioms are statements, assumed to be true without proof. More complex mathematics is then derived from them in the form of theorems and proofs. Usually, a small number of axioms is used. For example, Euclidean geometry in the plane only requires 5 axioms, from which other results like the Pythagorean Theorem or the fact that the sum of internal angles of a triangle equals 180° can be derived.

Proofs are also a key part of mathematical language, since they convince readers of the truthfulness of the statements a document presents. They usually consist of multiple steps, with each step making an assertion and providing justification for it. Proofs can range from a single word ("Trivial"), to instructions ("Proof is left to the reader"), paragraphs (which most proofs are), or in extreme cases, multiple chapters or even publications[9].

Here is an example of a theorem and subsequent proof, again taken from the Linear Algebra notes by Martin [27]

Theorem 1.14. A homogeneous system of linear equations with more unknowns than equations has infinitely many solutions.

Proof. Suppose a homogeneous system of linear equations with m equations in n unknowns, $m < n$. Using Gaussian elimination on the corresponding augmented matrix with m rows and $n + 1$ columns, we obtain a matrix in row echelon form with $k \leq m < n$ rows with non-zero entries. Hence at least one of the variables is a free variable, which implies that there are infinitely many solutions. □

The empty square at the end of the proof is a shorthand way of informing the reader that the proof has concluded. Sometimes, the square is filled in, and occasionally, especially in older texts, it is replaced by the letters *Q.E.D.*, an abbreviation of the Latin phrase "*quod erat demonstrandum*", meaning "which was to be shown/demonstrated".

[9] A notable example of such a proof was the classification of small finite groups, which took nearly 50 years and hundreds of articles.

2.1.3 Document level

On the document level, mathematical language is separated into books, articles, etc. There is no singular text containing all of mathematics. Within mathematical publications, topics are often grouped via paragraphs, sections, or chapters, which can help separate unrelated concepts, (and for example, make searching easier[10]) or break up the text into manageable chunks to read through at a time.

Another phenomenon on this level, which poses a challenge, is what Iancu [17] calls *framing*. This refers to presenting mathematical objects in terms of objects the reader already knows (for example, sets of integers as groups under addition, functions as sets of ordered pairs, ...). Therefore, any computer system for mathematics would need some database of all previously encountered mathematical knowledge in order to handle framing. A related challenge to framing is inter-document referencing. If a mathematical document builds on previous work, that work could be referenced in it. Sometimes, that referencing is explicit, and other times the document is written with the assumption that the reader is familiar with the precursor work (for example, lecture notes, which build on the material covered in courses in the past). An additional problem with such referencing can occur when the author changes notation for the concepts covered in the precursor work, without explicitly stating they have done so. It is up to the reader, which is in our case a computer system, to realise this change occurred.

The final phenomenon we will mention on the document level is foundational ambiguity. Often, foundational and logical assumptions are left out. This can be a source of ambiguity when reading the document, since one can potentially show that a document is true in different foundations of mathematics. There are, however, benefits to this since, by not providing a specific foundation, the contents of the document are automatically true in any foundation of mathematics, provided the mathematical objects described in said document are definable in those foundations and the reasoning principles used are compatible.

2.1.4 Representational language

In this section, we will outline some requirements set out by Iancu in [17] for a language that could adequately represent mathematics (referred to as a *representational language*), and some observations made by Iancu regarding the current state of such languages. Any representational

[10] When looking for information, say, in a book on linear algebra, one might just be interested in matrices and their operations, but not vectors, vector spaces, etc.

language must be able to represent imprecise and incomplete documents. That is because (as noted before) things are often left out in mathematical documents since the reader is assumed to know them, or assumed to be able to infer them from the context using their existing mathematical knowledge. A good example of this is writing $\frac{a}{b}$ in a mathematical document without explicitly stating that b should be non-zero. A representational language should also be simple and minimal. This makes it easier to be verified/formalised, managed, and spread through the mathematical community as a means of writing mathematics. There should also be systems in place to deal with any ambiguity that arises.

The main observation Iancu made about the current state of representational languages was that none of the current languages and systems for representing mathematics have the handling of the linguistic phenomena of the language of mathematics as a direct goal. Every time the currently existing attempts at creating a representational language implement the handling of any of these phenomena, it is out of necessity in pursuit of their goal, which is (automatic) formalisation. Another observation is the difference in terminology for the same concepts used by these different systems. This follows from a lack of standardisation and agreement, which is a problem highlighted in the Semantic Representation of Mathematical Knowledge Workshop [18].

2.1.5 Conclusions

This section shows that ambiguities are present on all levels of mathematical text, from small phrases to entire documents. Therefore, to achieve any long-term goal of computerising mathematical text, we need to handle ambiguities efficiently, and in a way that can be formalised and proven to be correct. Another phenomenon that needs to be handled to computerise mathematical text is the dynamic nature of the grammar of a representational language. With every new definition, the grammar of a language, in which mathematics is written, changes. In certain cases, rules could potentially need to be removed as well. Therefore, parsing software we could use to parse mathematics, must support changing its grammars (by either adding or removing rules) during parsing. That is what led me to study DynGenPar in the first place, instead of some other algorithm that is more established and has more documentation and research connected to it.

2.2 Proof-checking

There are currently a number of proof-checking systems in existence, which all differ in capability and implementation. Some were even used to formally prove conjectures, instead of verifying existing proofs. A notable example is the Kepler conjecture on sphere packing, which was formalised using HOL Light and Isabelle [14]. We will discuss some of these proof-checking systems with respect to whether or not they support something resembling natural language as input and output (for example, a CNL - Controlled Natural Language). For a side-by-side comparison of some of these proof systems, we refer the reader to the "Formalizing 100 Theorems" website [10], where formalisations of proofs of many famous theorems in mathematics are listed.

2.2.1 Without natural language support

These proof-checking systems are mentioned in order to paint a clearer picture of the current state of computer-based proof-checking. They require a language with a particular syntax that bears little to no resemblance with natural language. Using them requires experience with the input and output languages of the individual systems, and in order to check any mathematics with them, translation is required. Translation can be a potential source of errors and is also a time-consuming endeavour.

The first system we will mention is the aforementioned HOL Light [15]. HOL stands for higher order logic and serves as a basis for many provers. What sets HOL Light apart from the other HOL-based provers is the simplified logical core. It also provides several automated tools and pre-proved theorems, saving the user time by not having to implement them[11]. HOL-Light is built in OCaml and offers the option for users to program their own inference rules and add new theorems to the collection of proved knowledge.

Another system built in OCaml is called Coq [38], which started off in 1984 with the Calculus of Constructions [4] by Thierry Coquand and Gérard Huet. Coq is still under active development, with over 200 contributors so far. It provides a collection of tools for formalisation of proofs, such as interactive proof methods, decision algorithms and a *tactic* language for defining custom proof techniques. It allows for custom plugins written in OCaml and even interfaces with other theorem provers and computer algebra systems.

[11] Every bit of mathematical truth that is not part of the foundation a proving system is built on must be proven and provided to the system in order for it to be used.

The final proof-checking system we will mention in this section is the Mizar[12] system [29]. The development of Mizar started in 1973 as an attempt to recreate a mathematical vernacular[13] in a computer environment. Since 1989, the focus of the project, aside from improvements to the system, has been the development of a database of mathematics, called the Mizar Mathematical Library. It contains 13244 definitions and 65082 theorems as of November 14$^{\text{th}}$ 2021 [12].

2.2.2 With natural language support

The proof-checking systems described in this section are of particular interest to this dissertation, since we wish to deal with mathematics written in (something resembling) natural language. Having unambiguous input is key for a proof-checking system, so a mechanism to produce it could be useful for mathematicians using proof-checking systems.

The first system we will mention is the Naproche[14] (NAtural language PROof CHEcking) project [31][15]. The development of Naproche stopped in 2014[16]. The system uses a custom CNL (Controlled Natural Language) as the input language, which was designed to be similar to the language used in mathematical texts. Naproche is written in Prolog and its inner working, together with the algorithms used are described in Marcos Cramer's PhD thesis [5].

A system with little in common with Naproche besides the name is Naproche-SAD (NAtural language PROof CHEcking - System for Automated Deduction) [6]. It was designed by Adrian de Lon et al., a few years after the development of Naproche stopped. As the name suggests, it is built on the System for Automated Deduction [41] and uses ForTheL as its input and output language. ForTheL offers some support for formulas written in LaTeX which makes it possible to typeset the results in a way even more readable by humans. Naproche-SAD is written in Haskell and supports various proving techniques, such as proof by contradiction and proof by induction.

Recently, Naproche-SAD and the proof assistant called Isabelle have been integrated together. This new system combines proving and reasoning capabilities of Naproche-SAD with the interactivity of the Isabelle graphical user interface [6] (but none of the proving capabilities).

[12]There are some who would argue that the language of Mizar is close to mathematical language. However, we are classifying the proof-checking systems here based on how close their language is to natural language (which Mizar's is not).

[13]This is a term introduced by de Bruijn [7] to refer to the language used by mathematicians to communicate their ideas in papers.

[14]Sometimes also spelled NaProChe.

[15]Not to be confused with Naproche-SAD (discussed later), which is also sometimes referred to as Naproche.

[16]That is the year when the last update was released. Development might have stopped at a different point in time.

2.2.3 Conclusions

To successfully parse mathematical documents written by humans, we need to have a way of representing the mathematical content of these documents in a formal system. This survey of proof checkers served as an exploration of the state of the art of different ways that is achieved. It quickly became apparent that no existing system can convert mathematical documents in a natural language to a formal system. The focus then became trying to identify proof checking systems, whose input and output language is the closest to natural language. This is what led me to Naproche-SAD and the CNL (Controlled Natural Language) ForTheL that it uses, and also to Concise.

2.3 Comparison of representations of mathematical language

There exist many different representations of mathematical language. They all offer different approaches to the problem and were created for different purposes. Here we outline a few of them which are relevant to this dissertation.

2.3.1 MathNat

MathNat [16] is a CNL (Controlled Natural Language) developed by Muhammad Humayoun and Christophe Raffalli. It supports some more complex parts of language, such as anaphoric referencing, rephrasing sentences, and handling of distributive and collective readings[17]. This CNL is then automatically translated into MathAbs, a system independent formal language. The authors hope MathNat can be made accessible to any proof-checking system. MathAbs is then translated into first-order logic so that it can be checked for soundness. MathNat has been implemented in the Grammatical Framework and can potentially be used in conjunction with other pieces of software mentioned in this chapter.

[17]Distributive and collective readings are different ways to read a sentence, which contains a plural that refers to multiple objects (e.g., the pronoun "they"). That plural can be read *distributively* and thus bind the text after it to individual elements the plural refers to (e.g., "if $P(a) = P(b) = 0$ for some polynomial $P(x)$ and some real numbers a and b, *they* are roots of the polynomial", which we understand as "a is a root of $P(x)$ and b is a root of $P(x)$"). When reading it *collectively*, this binds the text after the plural to the collection of those multiple objects (e.g., "if the dot product of $\vec{v}$ and $\vec{w}$ is 0, we say *they* are orthogonal", which we understand as "$\vec{v}$ and $\vec{w}$ are orthogonal (to each other)").

2.3.2 FMathL, DynGenPar, and Concise

The FMathL (Formal Mathematical Language) project [32] of the University of Vienna aims to create a modelling and documentation language for mathematics. Their long term goal is to produce an automatic research assistant system, which could be used to retrieve and edit mathematical documents, and check for their correctness. The final product would be a system, in which arbitrary problems can be expressed. It would also make use of the typesetting capabilities of LaTeX, be semantically precise (i.e., unambiguous), and be user-friendly. The authors emphasise this last point since they want to create the system with the current habits of mathematicians in mind.[18] As part of the FMathL project, two pieces of software that are of interest to this dissertation were produced.

DynGenPar

DynGenPar [21] stands for Dynamic General Parser and was developed by Kevin Kofler. This parser has two key characteristics. Firstly, it is dynamic, which means that the grammar it is using to parse text can be modified at any time, even during parsing. When parsing mathematical texts, this is an essential feature, since, as described by Ganesalingam [11], definitions change the grammar of mathematical language. When encountered, they must therefore be added to the grammar. The second characteristic is its generality. This means it can parse text using any grammar which is specified in the correct format. DynGenPar was also embedded into Concise (see below), where it is now used as the main parser.

Concise

Concise [8] is a tool, which helps the user visualise and edit semantic graphs. Semantic graphs are labelled directed graphs which carry semantic information. Concise enables 4 different visualisations of these graphs, and both graphical and programmatic editing of them. A feature that was of particular interest to this project was the disambiguation of parsed text, described in Kofler's PhD thesis [21], Section 6.5. After installing Concise and running it, I realised that it is not easy to use, especially because I lack prior experience with it and am not familiar with its inner workings. Since it is unlikely I would be able to achieve the level of knowledge of Concise necessary to use it as part of the project in the limited time available, I decided to abandon it

[18] A system, which would require mathematicians to adapt too much might not catch on. An example of this is provided later on.

as a means of disambiguating text in favour of a different solution.

2.3.3 GF - the Grammatical Framework

The Grammatical Framework [35] is a programming language for multilingual grammars. It works by using an abstract grammar to define grammatical rules, and concrete grammars, which define vocabulary. Due to this structure, it has uses as translation software (for limited fragments of language). GF also supports custom libraries introducing grammatical concepts for specific languages (such as gender, cases, etc.). Grammars and libraries for mathematics such as MGL and MGF have been created to parse mathematical text.

MGF [28] (Mathematical Grammatical Framework) is a collection of grammars to use within GF, built by the KWARC group. It contains an English and German grammar and supports translation between the two to some extent. It also contains the means to formalise sentences in a logical representation. It uses LaTeXML[19] to convert LaTeX source code into HTML.

MGL [3] (Mathematical Grammar Library) is another collection of grammars built within GF. It was produced as part of the MOLTO (Multilinqual Online Translation) project [30] and contains different modules for various areas of mathematics. It was originally designed to generate mathematical exercises in multiple languages. A demo application was built but has since been taken down. Currently, the grammar supports translation into English, French, Spanish, Catalan, and Finnish, with the long-term goal of encapsulating all of the (then) 23 official languages of the EU.

2.3.4 ForTheL

ForTheL (Formal Theory Language) [34] is a controlled natural language created by Andrey Paskevich and Konstantin Vershinin [42]. It built on the work done previously by Vershinin (in the 1970s) as part of the "Algoritm Ochevidnosti" (Evidence Algorithm) project, which started in the 1960s. ForTheL was designed with readability in mind and is thus closer to the English language than most languages used by theorem provers. It was initially used in SAD (System for Automated Deduction) [41], and is now a key component of Naproche-SAD as well.

[19] A tool to convert LaTeX code into a tree-like representation such as XML.

2.3.5 MathLang

MathLang [19] is a project, started by Fairouz Kamareddine and Joe Wells. They aimed to create a computer representation of mathematics, which could connect other aspects of computerising mathematics, such as typesetting systems (LaTeX and presentation MathML), proof assistants (discussed earlier), and systems dealing with the semantics of mathematics (discussed later on). It has three core aspects. The Core Grammatical aspect is a weak type system which assigns categories (such as 'term' and 'statement') to different parts of text, and checks that the text makes sense grammatically. The Text and Symbol aspect deals with connecting mathematical meaning to symbols and associating natural language text to meaning. The third aspect, called the Document Rhetorical aspect, identifies chunks of text, assigns them a role (definition, theorem, proof, ...), and indicates the relations between these chunks.

2.3.6 Discourse representation structures

In his book [11], Ganesalingam discusses the use of Discourse Representation Structures (DRS) to represent the language of mathematics, provided some modifications are made to the original theory. A detailed discussion on how to handle linguistic phenomena (both from mathematical and natural language) follows. In particular, Ganesalingam shows how to handle anaphors, variables, plurals, types, definitions, and more. Since many examples have been showed to work using DRS, I hoped that Ganesalingam managed to create some software (or at least the beginnings of it) to encode the language of mathematics into DRS, but I was unable to find it. I tried contacting him but got no answer, so I decided to abandon DRS as a viable representation candidate for the project. Another argument in favour of abandoning DRS can be found in the history of Naproche. At some point during its development, DRS was used as a basis of the Naproche system. DRS have since been replaced by dynamic predicate logic, which shows other researchers also concluded that DRS is, in practice, not a viable representation of mathematical language.

2.3.7 Semantic markup

Semantic markup, more generally, refers to encoding the meaning of content (or the reference to it) into the content itself. In the context of writing mathematics in LaTeX, which is the most commonly used software for typesetting mathematics, that would entail writing the formulas in math mode in a way that renders as intended for readers but encodes the meaning of the

formula in the source code as well. As an example, consider the formula $\sin(x^2 + \theta)$. Most mathematicians would represent this in their LaTeX code as `\sin(x^2+\theta)`, but this gives no clear indication as to what any of the symbols mean. A parser could infer that `\sin` stands for the sine function, since LaTeX math mode has an explicit command to insert it, but the meaning is not clear for `x^2` and `\theta`. To semantically markup this formula, one would rewrite it as, say, `\sin(\power{\var{x}}{2}+\realnum{\theta})` using predefined commands (examples here are power, var, realnum), which would render the input to produce $\sin(x^2 + \theta)$. A parser could then, assuming there is a standard way of semantically marking up formulas, be able to infer the meaning of the entire formula, leaving nothing ambiguous. A number of existing systems deal with semantic markup of mathematics, such as Content MathML, OpenMath, OMDoc, and sTeX. MathML, OpenMath, and OMDoc are XML based, while sTeX is designed specifically for LaTeX documents.

Semantic TeX - sTeX

Semantic TeX (abbreviated as sTeX) [26] is an attempt to semantically markup mathematical formulas. It is a collection of LaTeX macro packages which encode the reference to the meaning of formulas into the writing of the formulas themselves. Then there is a separate mapping between these references and their mathematical meaning. The software is freely available on GitHub [24], but is not widely used. There are several reasons as to why that might be, but we will not list them all here.

One reason for it not being widespread might be, to put it plainly, because people are not used to it. Most mathematicians would, when typesetting formulas, just typeset the symbols required so that the formulas render as intended once LaTeX is exported to PDF format. This is simpler and faster than semantically marking up the formulas, especially if one has never tried sTeX before. Since most mathematical documents are intended for human readers, which requires rendering them as PDFs, semantic markup is not a priority for the general mathematical research community. Another reason, which becomes clear quickly, is the difficulties encountered in setting it up, which will be discussed in detail later on.

2.3.8 Conclusions

This section is a survey of approaches different research groups took to represent mathematics in a form that is as close to natural language as possible. While some achieve that by limiting

themselves to a strict subset of natural language (the CNL approach), others try to formally represent the semantics of language (such as MathLang, Concise, and sTeX). However, there is currently still no way to represent arbitrary natural language in a formal system.

2.4 Machine learning

Machine learning as a means of parsing natural language is not a new idea. However, I have not encountered a lot of research done in the area of parsing the language of mathematics with machine learning. One paper I found which was related to our research was a paper by Pagael and Shubotz [33]. They used machine learning to try and find the meaning of formulas in mathematical text (as an example, they wanted to figure out that the E in $E = mc^2$ stands for energy). There are certain issues which arise when trying to parse mathematical text with machine learning, though. The most clear reason are the foundational issues that come with it, since machine learning is probability based. 99 % certainty that a sentence has the meaning which the machine learning model predicted is not satisfactory for mathematical truths, which we always state with 100 % certainty. The idea of user interaction as a means of disambiguating text (if a user confirms something is true, we can trust it) was partially also inspired by Pagael and Shubotz's work [33].

2.4.1 Conclusions

In the end, I decided not to use machine learning. Aside from the foundational issues that using probability to parse mathematics has, there is also a lack of experience from both my supervisor and me. As learning how to apply machine learning to help us achieve our goals would take too much time, it was abandoned in favour of a different approach. We should note here, however, that machine learning could potentially be used to augment some of the things mentioned earlier in this chapter, but that is beyond the scope of this dissertation.

2.5 Parsing

Parsing is the process of figuring out the structure of a sentence, given a collection of rules known as a *grammar*. A grammar is referred to here in a more general sense than just the grammar of a natural language. Programming languages, for example, also have grammars. The idea of using computers to parse text has been around for decades and is used in everyday

applications, such as code compilers. We will first look at some of the core components of a parser, and then list and evaluate some parsing algorithms based on whether or not they fulfil the requirements we set out earlier.

2.5.1 Context-free grammars

A context-free grammar $G = (N, T, P, S)$ is a quadruple, where N is the set of symbols referred to as *non-terminals*, T is the set of symbols referred to as *terminals*, P is a set of *rules* (sometimes referred to as *productions*), and $S \in N$ is known as the *start symbol*. There is an additional condition, which is that N and T are disjoint (i.e., $N \cap T = \{\}$).

In a context-free grammar, all rules are of the form $n \to s$, where $n \in N$ is a single non-terminal, and s is a (possibly empty) sequence of elements that belong to $N \cup T$ (referred to as the set of *symbols*). Conceptually, rules can be thought of as "replacing" or "rewriting" the non-terminal on the left-hand side with the sequence on the right-hand side.

Given a context-free grammar and an input sentence (we will assume here, for simplicity, that it is a finite sequence) consisting of terminals, parsing algorithms will try to find a sequence of rewritings starting with S which will reproduce the input sequence. The sequence of rewritings is usually represented in a data structure known as a *parse tree* with a node labelled with S as its root and the input terminals as its leaves, such that the order of the leaves is the same (when read from left to right) as the order of the terminals in the input. If a rule of the form $n \to s$ is used during parsing, each element of the sequence s is attached as a child to the node labelled with n.

2.5.2 Example

Consider a simple context-free grammar $G = (N, T, P, S)$, where $N = \{S, A\}$, $T = \{a, b\}$, and $P = \{S \to A\,b, A \to a\}$. We will refer to $S \to A\,b$ as rule 1, and $A \to a$ as rule 2. Given an input $a\,b$, a parser will try to find the sequence of rewritings which starts with S and ends with the sequence $a\,b$. In this example, it is clear to see (since there is only one option) that we can replace S with $A\,b$, as per rule 1. Since this is not yet equal to the input, we need to keep going. Replacing A with a, as per rule 2, yields $a\,b$, which is equal to the input, so we're done. The parse tree in Figure 2.1 represents this parsing.

Note that this example was simple and straightforward to parse due to the fact that each non-terminal that needed to be replaced had only one choice of replacement. Adding more rules

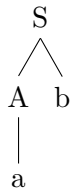


Figure 2.1: An example of a parse tree

can make finding a parsing of a given input more challenging (e.g., by adding the rule $S \to A\,c$, which gives us two options to rewrite S, with only one leading to the right parse tree), or leads to ambiguities (e.g., by adding the rule $S \to a\,b$, which gives us two possible parsings). Various parsing algorithms exist, which deal with eliminating wrong rules and ambiguities in different ways.

2.6 Parsing algorithms

In this section we will consider some of the parsers which were already considered by Kofler in his thesis [21], and assess how suitable they are based on the requirements set out earlier.

- **LR**

  The LR algorithm was first introduced by Knuth in 1965 [20]. It is usually denoted LR(k) for some positive integer k. The k represents the number of terminals in the input that the algorithm can "see" in front of the current position. This is referred to as *lookahead*. Lookahead is then used to produce a parse tree using the *bottom-up* approach. In a bottom-up approach, if a sequence of k or fewer symbols forms the entire right-hand side of a rule, it is rewritten as (or *reduced to*) the non-terminal on the left-hand side of the same rule.

  It is highly efficient and can parse input strings in linear time. However, since it reduces as soon as the right-hand side of a rule is matched, other rules with longer sequences of symbols on the right-hand side which could still be matched (by considering a longer section of the input) are never considered. Since it would not find all possible parsings of a given input, the LR algorithm is not suitable to handle ambiguities.

- **GLR**

  The GLR (Generalized LR) algorithm was introduced by Tomita in 1985 [39]. The main improvement to LR that is made is the fact that all possible reductions are considered

(and not just the first one that the algorithm encounters, as is the case with LR), meaning that GLR can handle ambiguities. While GLR handles ambiguities efficiently, it relies on complex tables to represent its grammars, which need to be computed before parsing. If a grammar changes, these tables need to be recomputed, which can become very computationally expensive as grammars grow. As such, adding rules dynamically during parsing with a GLR algorithm is not practical, making GLR not meet our requirements.

- **LL**

  The LL algorithm [36] is similar to the LR algorithm, but instead of a bottom-up approach employs a *top-down* approach, also using k terminal lookahead. With a top-down approach, a symbol on the left-hand side of a rule is replaced with the sequence on the right-hand side. To figure out what rule to use, LL algorithms use a special table[20], which indicates which rule to use if the input contains a given terminal at the current position and the algorithm is expanding a given non-terminal. With k lookahead, all rules which match a right-hand side of length at most k will be considered, so ambiguities can be handled[21] using an LL(k) parser. The drawback of using a table is, as in the GLR case, that it needs to be recomputed each time the underlying grammar changes (either by adding or removing rules). As such, LL parsers do not meet the requirements we set out earlier.

[20] This table functions similarly to the neighborhood defined in DynGenPar [21].

[21] For a big enough value of k, this will cover any potential rule in the grammar, as long as rules are finite.

Chapter 3

Excluded software

In the background research for this dissertation, I encountered and tried several pieces of software, which proved challenging to use for various reasons. However, despite all the difficulty, they taught me some valuable lessons, mostly in what not to do when producing software.

3.1 sTeX

sTeX [26] looked like a good candidate for producing unambiguous symbolic mathematics. It was therefore of particular interest to me to try and set it up. After following the installation instructions on WSL (Windows Subsystem for Linux), which consisted of obtaining TeX live[1] and setting an environment variable, I tried getting a minimal example of sTeX running. A number of issues arose when trying to compile it. Solving them required lots of experience with TEX and LATEX and as such I could not solve them on my own. My supervisor, Joe Wells, helped me overcome these issues after we spent a few hours over two meetings trying to solve them.

Under the assumption that sTeX was now set up and working, I tried to typeset the examples provided by the sTeX source code. This proved difficult once again, since just running `pdflatex filename.tex`[2] from the WSL Ubuntu terminal did not work. The folder with the examples contained a `makefile`, which I could run in the terminal using the `make` command. Finally, some `.pdf` outputs were produced. It turned out, that one of the examples was a set of instructions on how to compile files with sTeX and an explanation for why certain files do

[1] TeX live is a TEXdistribution. There were no explicit instructions to obtain it, but since I wish to work with `.tex` files, it is rather obvious I would need it.

[2] Here, `filename` is just a placeholder. It was replaced with actual individual file names when trying to compile them.

not produce `.pdf` outputs when compiled. To further add to the confusion, the document also appears to be incomplete.

Then, I tried getting it to run on the sTeX test suite, which ships with the source code. Again, some files within the suite did not compile successfully, despite sTeX appearing to work (it worked on examples provided, and the minimal example I wrote myself). To conceivably use sTeX as a means of representing parts of mathematical texts unambiguously, more work is required.

3.1.1 Problems with sTeX

There are several reasons sTeX is problematic to use. As mentioned, I had trouble setting it up, and even after I did, I was not sure how to use it. Examples were provided with the source code but were, for some reason, incomplete. They were thus of little help when figuring out how to use sTeX. It became clear that when producing software of my own, I would have to include good examples on how to use the software, to spare the user as much difficulty as possible.

Another reason why sTeX was difficult to use is its incredibly complex internal dependency structure. There are packages relying on packages relying on packages. The complexity of this internal dependency structure makes debugging (when using, setting up, or upgrading the software) more difficult and also makes it harder for someone else (with no experience) to build new software or produce research based on it.

The final difficulty I will mention here is a lack of good documentation. The installation instructions were brief and OS-specific. There was no documentation on how to use it, and even the test suite did not compile successfully. Therefore, I strived to write good documentation and provide a working example for any software I produced.

3.2 Concise

As mentioned in the first deliverable, I reached out to Dr Kofler and Dr Neumaier, who worked on Concise [8]. A few days after the deadline for the first deliverable, they replied to me and eventually sent me the newest versions of both Concise and DynGenPar. Unlike sTeX, Concise was extremely easy to install and run. It ships as a standalone executable that installs Concise on your computer. I was hoping to use Concise both as an interface with DynGenPar (for input and output), and to understand DynGenPar better. I had some trouble getting started with

Concise and would have to spend lots of time trying to understand what Concise does and how it does it. That was one of the reasons I ultimately did not use Concise.

It also turned out that DynGenPar was very intertwined with Concise and could, from my understanding, only accept grammars in Concise's internal representation. Learning how Concise's internal grammar representation works would have taken too much time. In addition to the above reasons, my supervisor told me that Concise follows an approach to types which was likely to conflict with our long-term goals. These factors contributed to my decision to not try to build on top of Concise, but to shift my focus on understanding DynGenPar instead.

Chapter 4

DynGenPar

I first encountered DynGenPar early on in my background research, as part of a bigger project in the field of mathematical knowledge management, FMathL. The papers written on it [21–23] describe it as a parser that can handle any grammar and also supports adding rules to the grammar during parsing, without having to recompile a table. According to its authors, it can also find every parsing of a given input in case of ambiguities. As it can potentially handle ambiguities, which are present throughout mathematical texts, and can change the grammar during parsing (one of the key requirements set out by Ganesalingam [11]) I had great hopes that it will be usable.

4.1 Initial issues

There were some issues with DynGenPar that became apparent after I started studying it. These ultimately steered the project in a slightly different direction, as I realised that some work is required on DynGenPar in order to definitively assess whether or not is can be used as a means of disambiguating mathematical texts.

4.1.1 Setup difficulties

Once I downloaded the source code, I followed the installation instructions provided, using a distribution of Ubuntu 20.04 within WSL (Windows Subsystem for Linux). These are the steps I took to compile the source code successfully:

1. I obtained the packages `cmake`, `make`, and `qt5-default`. The instructions specified `qt4-devel` version 4.6.0 or higher. The changelog mentioned that DynGenPar supports Qt5,

but the installation instructions did not mention that anywhere. While the source code can be compiled using Qt4, the Qt4 packages are no longer present in the Ubuntu archive on Ubuntu 20.04 (as they are deprecated due to the existence of Qt5) and this caused a bit of confusion. Once I realised I can use Qt5 and figured out what the package is called, I installed `qt5-default` instead.

2. Alongside the three packages mentioned, I also had to install `extra-cmake-modules` and `build-essential` which were not mentioned in the installation instructions, but were required for the compilation to succeed.

3. According to the instructions, I should only have to run `cmake .` for a Makefile to be produced. However, the compiler ran into issues. It could not find the installation directory of Qt5. From my understanding, after hours spent browsing the internet in search of a solution, this is because cmake does not know where to find Qt5, and Qt5 does not update cmake's search path to enable that. Finally, I found a solution, after trying numerous others. Instead of running `cmake .`, I had to run `cmake . -DCMAKE_PREFIX_PATH=/usr/lib/x86_64-linux-gnu/cmake/Qt5`. Finally, a Makefile was produced.

4. Then, all I had to do was run `make` and the source code compiled without errors.

I think I could have spent less time trying to compile the source code if I had more (i.e., any) prior experience using cmake. A positive consequence of the numerous things I tried was figuring out how to compile C++ applications that make use of Qt, which proved to be useful later on.

4.1.2 Lack of grammars

Kofler's PhD thesis [21] mentions several grammars that were used with DynGenPar. Some of them, namely the ones for MathNat, Naproche, and the grammar for LaTeX formulas seemed potentially useful in pursuing of my goals. I looked through the source code, and aside from a few small examples could not find these grammars anywhere. From my understanding, they are packaged with Concise, and written in its internal representation as well. Thissetback meant I would need to write my own grammar, figure out a way to convert them from the Concise format (and understand Concise - which I decided not to try and do) or try to find a grammar elsewhere. As mentioned earlier, I decided studying Concise was not a good use of my

limited time, and while there are other grammars out there, such as the grammars produced by the MOLTO project [3] and the KWARC research group [28] for use within the Grammatical Framework. In the end, I encountered one more issue which led me to not use a grammar for parsing mathematics during this dissertation at all.

4.1.3 Theory and practice

After I managed to compile the DynGenPar source code, I could begin studying the algorithm. It soon became apparent, that the algorithm described in Kofler's thesis [21] and the C++ implementation were quite different. The thesis describes a non-deterministic algorithm, which can, if the right non-deterministic choices are made, find a parse tree for a given input. The C++ implementation, however, implements a version of this algorithm, which makes all of the non-deterministic choices and considers them concurrently by forking the parsing process using *continuations*.

This is an important difference which was not mentioned in any of the papers written on DynGenPar [21–23]. I want to point out here that while DynGenPar lacks a formal description, which is mathematically precise and matches either the algorithm described in Kofler's thesis or the C++ implementation, lots of features that were added to the algorithm to make it more usable in practice. The C++ implementation extends the basic algorithm description with functionality such as parsing prediction, support for PMCFGs (parallel multiple context-free grammars) [2], a variety of token sources, and several ways to optimise the algorithm's runtime.

Most importantly, by concurrently considering all the non-deterministic choices during parsing, the C++ implementation does not just find a single parse tree for a given input, but all of them. This is important in the presence of ambiguities, since they introduce multiple ways to parse the same input. The implementation also contains a way to potentially add rules to the grammar during the parsing process[1], and Kofler even provides a small example of using it in practice.

This difference between the theoretical description and the practical implementation is what led me to try and come up with a formal description of what the algorithm does. This formal description, along with everything else that was needed to produce it, is the main focus of the remainder of this chapter. I also work through the steps it takes to parse a very simple example

[1] Although due to the nature of its implementation (and the fact that the algorithm is non-deterministic) it can be tricky to define precisely how and when these rule additions affect the grammar.

to try and illustrate how the algorithm works.

4.2 Understanding DynGenPar

This section will first present a framework that is needed to explain the DynGenPar algorithm. Then, I will present a cleaned up version of the non-deterministic algorithm, as presented in the various publications relating to DynGenPar [21–23]. At this point, I'd like to note that this version of the algorithm is not the same as the one found in Kofler's C++ implementation. The C++ implementation is based on the non-deterministic version, as seen in Kofler's thesis [21], but at every non-deterministic choice, the process forks to consider all of the options concurrently. Following the cleaned up description of the algorithm from Kofler's thesis is a discussion of *continuations*, which are a crucial part of the C++ implementation, and a showcase of the algorithm in action on a simple example.

4.2.1 Framework

This section presents a framework required for the explanation of the DynGenPar algorithm later on in this chapter. I will use "w.r.t." to mean "with respect to". There will also be some mentions of *continuations*. They will be more precisely described later on, but for now, think of them as points in the algorithm, from which data is saved for later calculations that can then be resumed at any time. Kofler refers to these continuations as *stacks* and *stack items* in his thesis and the C++ implementation of the DynGenPar algorithm.

Sequences

- Let a *sequence* be a function $s = \{(0, x_1), (1, x_2), ..., (n, x_n)\}$. Alternatively, this can be written as $[x_1, x_2, ..., x_n]$.
- Let s range over sequences.
- Let $|s|$ denote the cardinality of s. Conceptually, this stands for the length of the sequence.
- Let $FSeq(A)$, where A is a set, be the set of all finite sequences that can be built using only elements of A.
- As a convention, elements of $FSeq(A)$ will be denoted as $\vec{A}$ throughout this document.

- For two sequences, $s = [x_1, x_2, ..., x_n]$, $s' = [x'_1, x'_2, ..., x'_l]$, let $s \cdot s'$ stand for concatenation, i.e., $s \cdot s' = [x_1, x_2, ..., x_n, x'_1, x'_2, ..., x'_l]$. Alternatively, $s \cdot s' = s \cup \{(i + |s|, s'(i)) \mid i \in (0..|s'|)\}$.

Symbols

Let N range over finite sets of symbols referred to as *non-terminals*, and let T range over finite sets of symbols referred to as *terminals* (or tokens as seen in Kofler's thesis [21]), such that $N \cap T =$ (i.e., N and T are disjoint sets). Let $\Gamma = N \cup T$ and let y range over Γ (we use y, for sYmbol). Let t range over T, and let η range over N.

Parse trees

- A *parse tree* (denoted $\mathcal{T}$) is an ordered pair $y\#\vec{\mathcal{T}}$, where $y \in \Gamma$, and $\vec{\mathcal{T}}$ is a sequence of parse trees.
- When $y \in \Gamma$ occurs in a parse tree (left of the operator), it is referred to as a *node label*.
- Let $\mathit{symbols}(y\#[\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_n]) = \{y\} \cup \bigcup_{i=1}^{n} \mathit{symbols}(\mathcal{T}_i)$.
- In this document, a parse tree $\mathcal{T} = y\#[\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_n]$ is called *non-recursive* if and only if all the subtrees (the $\mathcal{T}_i$s) are non-recursive and $y \notin \mathit{symbols}(\mathcal{T}_i)$ for all $i \in (1..n)$.

Rule

A *rule* is an ordered pair $(\mathit{lhs}, \mathit{rhs})$, where $\mathit{lhs} \in N$ and $\mathit{rhs} \in \mathit{FSeq}(\Gamma)$. Let P range over sets of rules.

Context-free grammar

A *context-free grammar* is a quadruple $G = (N, T, P, S)$, where N is the set of non-terminals, T is the set of terminals, P is the set of rules, and $S \in N$ is referred to as the *start symbol*. There is also a necessary condition that N and T are disjoint. We are fixing a particular context-free grammar G for the remainder of this section.

Parsing

Let *Parse* be the smallest relation (w.r.t. $\subseteq$) such that:

- $Parse(G, t, [t], t\#[\,])$ holds for all $t \in T$.
- For a rule $r = (\eta, rhs) \in P$, sequence s, and tree $\mathcal{T} = \eta\#[\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_k]$, $Parse(G, \eta, s, \mathcal{T})$ holds if and only if there exists $s_1 \cdot s_2 \cdot ... \cdot s_k \in FSeq(T) = s$ such that $Parse(G, rhs(i), s_i, \mathcal{T}_i)$ holds for all $i \in (1..k)$.
- We say a tree $\mathcal{T} = \eta\#\vec{\mathcal{T}}$ *parses* a sequence $\vec{t}$, if and only if $Parse(G, \eta, \vec{t}, \mathcal{T})$ holds.

Superfluous recursion

In this document, a parse tree $\mathcal{T}$ is said to contain *superfluous recursion*, if and only if $\mathcal{T}$ parses a sequence $\vec{t}$, there is a subtree $\mathcal{T}'$ in $\mathcal{T}$ such that $\mathcal{T}'$ also parses $\vec{t}$.

Nullable non-terminal

For a context-free grammar G, a set $A \subseteq N$ is said to be *null-closed* w.r.t. G if and only if the following holds: for every rule $(\eta, rhs) \in P$, if $(ran(rhs) \subseteq A)$, then $\eta \in A$.

Let $nullable(G)$ be the smallest (w.r.t. $\subseteq$) set that is null-closed w.r.t. G. If $\eta \in nullable(G)$, we say that η is *nullable*. If η is nullable, then there exists a tree $\mathcal{T} = \eta\#\vec{\mathcal{T}}$ such that $Parse(G, \eta, [\,], \mathcal{T})$ holds (i.e., $\mathcal{T}$ parses $[\,]$).

We say that a sequence $\vec{\eta}$ is nullable if and only if $ran(\vec{\eta}) \subseteq nullable(G)$.

Initial graph

The *initial graph* corresponding to G is a directed labelled multigraph whose vertices are symbols. For each rule $r = (v, left \cdot [u] \cdot right)$ with $ran(left) \subseteq nullable(G)$, there is an edge (u, r, v) connecting vertices u and v. Each edge in the initial graph is labelled by the rule r generating it. Let E be the set of all edges. Then, the initial graph of a grammar G, $initial(N, T, P, S)$ is the ordered pair $(N \cup T, E)$.

Neighborhood

For $y \in \Gamma$, $\eta \in N$, $neighborhood(y, \eta)$ is constructed by considering all the paths of length 1 or more from y to η in the initial graph, and taking the labels of the initial edges of those paths. This will give us a set of rules.

Parse state

A *parse state* is a quadruple of the form $(\vec{t}_{left}, \vec{t}_{right}, matches, continuations)$. The sequence $\vec{t}_{left}$ represents the sequence of tokens that have been shifted (the word "shifting" is used as in the parsing literature), and $\vec{t}_{right}$ represents the sequence of tokens that still need to be shifted. The set *matches* contains all the parse trees which have the symbol S at their root and parse $\vec{t}_{left}$. They are the results of the algorithm, if and only if $\vec{t}_{right}$ is empty, and are otherwise discarded. The set *continuations* contains the continuations that need to be processed. They are all processed at once when a new token is shifted.

4.2.2 The algorithm

The DynGenPar algorithm described in Kofler's thesis [21] is a non-deterministic algorithm that finds a tree that parses a given input which does not contain superfluous recursion. In particular, a sequence of non-deterministic choices can be made to produce each valid parse tree. In this section we present the four main operations used by the algorithm. They were introduced by Kofler in various publications [21–23], but here we attempt to present them a bit more clearly. We will use (ND) to denote a point or step in the algorithm, where a non-deterministic choice is made. To get around the non-determinism of the algorithm in the C++ implementation, all the non-deterministic choices are made and all options considered concurrently (this will be discussed in more detail in the next section).

We now try to describe the four main operations of the algorithm, with some comments and annotations on how the concurrent executions of the C++ implementation fit in:

This algorithm contains some notion of a state which includes two sequences, $\vec{t}_{right}$ and $\vec{t}_{left}$, and a fixed context-free grammar $G = (N, T, P, S)$.

- **Operation:** $match_\epsilon(\eta, M)$

  Given $\eta \in N$ and a set $M \subseteq N$, $match_\epsilon$ will find all the non-recursive parse trees $\mathcal{T} = \eta\#\vec{\mathcal{T}}$ such that $\mathcal{T}$ parses $[\,]$. It does so as follows:

  1. Let $result = \{\,\}$.
  2. Let $M' = M \cup \{\eta\}$.
  3. For each rule $(\eta, rhs) \in P$ such that rhs is nullable and $ran(rhs) \cap M' = \{\,\}$:
     (a) Let $\mathcal{T}_\eta = \eta\#[\,]$.

(b) Then, for each $\eta' \in rhs$, append $\mathcal{T}_{\eta'} \in match_\epsilon(\eta', M')$ as a child of $\mathcal{T}_\eta$. (ND) (Note that the implementation will create a separate copy of $\mathcal{T}_\eta$ for each $\mathcal{T}_{\eta'} \in match_\epsilon(\eta', M)$, producing $|match_\epsilon(\eta', M)|$ trees to be added to *result*.)

(c) Add $\mathcal{T}_\eta$ to *result*.

4. Return *result*

- **Operation:** *shift*()

  This operation moves the first token t from $\vec{t}_{right}$ to $\vec{t}_{left}$ and also returns $\{t\}$ to its caller. If $\vec{t}_{right}$ is empty, the empty set is returned.

- **Operation:** $match(y)$

  Given $y \in \Gamma$, $match(y)$ does the following:

  1. If $y \in nullable(G)$, compute $match_\epsilon(y, \{\})$. Non-deterministically choose $\mathcal{T}_\epsilon \in match_\epsilon(y, \{\})$. (ND) (In the next section we will talk about how the non-deterministic scheduling splits the parsing process and the subsequent steps are executed concurrently for each possible $\mathcal{T}_\epsilon$ in order to consider all options. If there is no choice to be made, this branch of the exploration fails.)
  2. (The continuation $match_0$ is created and added to the set of continuations now.)[2] Let S be the result of *shift*(). (When the continuation $match_0$ is invoked, calculations are resumed from here.)[2]
  3. Then, proceed in one of two ways:
     (a) If $S = \{t\}$,
        i. Let $\mathcal{T}_t = t\#[\,]$.
        ii. Proceed in one of two ways:
           A. If $y \in T$ and $y = t$, return $\mathcal{T}_t$.
           B. If $y \in N$, (The continuation $match_1$ is created now.)[2] return $\{\mathcal{T}_\epsilon\} \cup reduce(y, t, \mathcal{T}_t)$ (When the continuation $match_1$ is invoked, calculations are resumed from here.)[2].

     (b) Otherwise, return $match_\epsilon(y, \{\})$
  4.

[2]This will be explained in more detail in the next section.

- **Operation:** $reduce(y, z, \mathcal{T}_y)$

  Given $y, z \in \Gamma$, and a parse $\mathcal{T}_y$ for y, which will be what the algorithm has recognized so far. We will refer to z as the *target* of the *reduce* operation. Consider all the rules $(\eta, left \cdot [y] \cdot right) \in neighborhood(y, z)$. Non-deterministically choose a rule and do the following: (ND) (Note that there are $|neighborhood(y, z)|$ distinct choices to make.)

  1. For each $i \in (0..|left| - 1)$:
     - Pick a tree $\mathcal{T}_{\epsilon i} \in match_\epsilon(left(i))$ (ND)
  2. Let $\vec{\mathcal{T}_\epsilon}$ be the sequence constructed by taking the set of all ordered pairs $(i, \mathcal{T}_{\epsilon i})$. (Note that if $|left| = 0$, $\vec{\mathcal{T}_\epsilon} = [\,]$.)
     (Note that there are $match_\epsilon(left(i))$ options to choose a $\mathcal{T}_{\epsilon i}$, so there are $\prod_{i=0}^{|left|-1} |match_\epsilon(left(i))|$ distinct sequences $\vec{\mathcal{T}_\epsilon}$.)
  3. (The continuation $reduce_4$ is created now.)[2]. For each $j \in (0..|right| - 1)$:
     - (The continuation $reduce_3$ is created now.)[2]. Pick a tree $\mathcal{T}_{yj} \in match(right(j))$
       (When the continuation $reduce_3$ is invoked, calculations are resumed from here.)[2]
       (ND)

     (When the continuation $reduce_4$ is invoked, calculations are resumed from here.)[2]
  4. Let $\vec{\mathcal{T}_r}$ be the sequence constructed by taking the set of all ordered pairs $(j, \mathcal{T}_{rj})$. (Note that if $|right| = 0$, $\vec{\mathcal{T}_r} = [\,]$.)
     (Note that there are $|match(right(j))|$ choices for each $\mathcal{T}_{rj}$, so there are $\prod_{i=0}^{|right|-1} |match(right(i))|$ distinct sequences $\vec{\mathcal{T}_r}$.)
  5. Now, let $\mathcal{T}_\eta = \eta\#\vec{\mathcal{T}_\epsilon} \cdot [\mathcal{T}_s] \cdot \vec{\mathcal{T}_r}$
  6. Then, do one of two things:
     - If $\eta = z$, return $\mathcal{T}_\eta$.
     - (The continuation $reduce_2$ is created now.) Otherwise, return the result of $reduce(\eta, z, \mathcal{T}_\eta)$.(When the continuation $reduce_2$ is invoked, calculations are resumed from here.)

4.2.3 Continuations

The C++ implementation of DynGenPar implements the non-deterministic choices in the algorithm, by considering all the options concurrently. At each non-deterministic choice the

processing forks into several threads and a separate continuation is made for each of them. Every continuation then computes the consequences of making a particular choice. We will illustrate this behaviour on an example in the next section. To achieve concurrent execution, threads of computation are put into continuations, which are then all run after a *shift* occurs. If during computation the thread requires an additional token to be shifted, processing stops and the continuation is added to *continuations*, where it will wait for all other threads to also be similarly blocked before proceeding. In his C++ implementation, Kofler calls the continuations "stack items".

In the previous section, we put notations referring to $match_0$, $match_1$, $reduce_2$, $reduce_3$, $reduce_4$ (which we will refer to as *types* of the continuation)[3] in various places throughout the algorithm description. They indicate at what point during the execution a continuation of a given type is created, moved to *continuations* (part of the parse state 4.2.1) and from what point in the algorithm it will resume computation when processed. We can define continuations as sequences in one of 6 ways:

- $[done]$ - this continuation is a special case. When it is returned a tree, it will add said tree to *matches* (part of the parse state).
- $[match_0, parent, y]$ - here, $match_0$ is a constant used to identify a specific point in the algorithm (see previous section), *parent* is a continuation, and y is a symbol.
- $[match_1, parent, y]$ - here, $match_1$ is a constant used to identify a specific point in the algorithm (see previous section), *parent* is a continuation, and y is a symbol.
- $[reduce_2, parent]$ - here, $reduce_2$ is a constant used to identify a specific point in the algorithm (see previous section) and *parent* is a continuation.
- $[reduce_3, parent, \mathcal{T}, rhs, i]$ - here, $reduce_3$ is a constant used to identify a specific point in the algorithm (see previous section), *parent* is a continuation, $\mathcal{T}$ is a tree, *rhs* is the right-hand side of a rule, and i is a non-negative integer.
- $[reduce_4, parent, y]$ - here, $reduce_4$ is a constant used to identify a specific point in the algorithm (see previous section), *parent* is a continuation, and y is a symbol.

[3]The continuations $match_0$, $match_1$, $reduce_2$, $reduce_3$, and $reduce_4$ correspond to stack items of type 0, 1, 2, 3, and 4 from the C++ implementation, respectively.

Due to a lack of explanation in Kofler's thesis [21] and other DynGenPar related papers [22, 23], and the brevity of the comments in the code, figuring out what stack items do took a big portion of my time. Even once I realised they were continuations, more work was needed to fully understand the function of each of them. In fact, I only managed to work everything out once the code was significantly shortened (more on that later). I should also mention two key differences between this description of continuations and the original implementation:

- There are two continuations that were removed when extracting the core functionality of the algorithm (see 4.4)
- As a consequence of this extraction, some information that was carried by the remaining continuations (the ones described above) was also pruned

that Kofler's unmodified stack items carry more information than the continuations described here, because I derived their description from the trimmed down version of DynGenPar (see 4.4)

4.2.4 The algorithm on an example

Consider a simple context-free grammar $G = \{N, T, P, S\}$, with $N = \{S, A\}$, $T = \{a, b, c\}$, and $P = \{S \to a, S \to A\,c, S \to a\,b\,c, A \to a\,b\}$. This example is meant to illustrate some functionality of the algorithm and has little real-world application. We will showcase the steps the algorithm takes to parse the input $a\,b\,c$, which we will represent as a sequence, $[a, b, c]$. The initial graph for the grammar is computed as specified above and looks like this:

Note that $\mathit{nullable}(G) = \{\,\}$ which greatly simplifies the computation of this example and allows us to focus on showcasing the concurrent executions. A new continuation, $\mathit{cont}_1 = [\mathit{match}_0, \mathit{done}, S]$ is created. The initial parse state is:

$$([\,], [a, b, c], \{\,\}, \{\mathit{cont}_1\})$$

The next thing to do is to shift a token, which updates the parse state to:

$$([a], [b, c], \{\,\}, \{\mathit{cont}_1\})$$

Then, cont_1 is processed, resuming right after step 2 of $\mathit{match}(S)$. A leaf tree, $\mathcal{T}_a = a\#[\,]$ is created. Since $S \notin T$, a new continuation $\mathit{cont}_2 = [\mathit{match}_1, \mathit{done}, S]$ is created. Then,

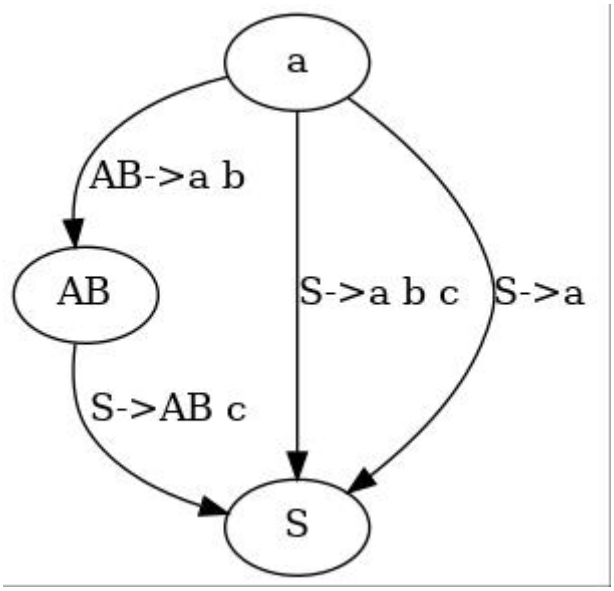


Figure 4.1: The initial graph of the example grammar, which I automatically generated.

reduce$(a, S, \mathcal{T}_a)$ is called.

We have *neighborhood*$(a, S) = \{S \to a, S \to a\,b\,c, A \to a\,b\}$. Therefore, the parsing process will split into three concurrent executions, one for each rule:

1. $S \to a$:

   There are no nullable η to the left of a in the rule, so step 1 of *reduce* is skipped. Therefore, we have $\vec{\mathcal{T}_\epsilon} = [\,]$. There are no symbols to the right of a in the rule, so step 3 is also skipped. We have $\vec{\mathcal{T}_y} = [\,]$. We can thus construct $\mathcal{T}_S = S\#\vec{\mathcal{T}_\epsilon} \cdot [\mathcal{T}_a] \cdot \vec{\mathcal{T}_y} = S\#[a\#[\,]]$. Since $S = S$ (i.e., the label of the root of $\mathcal{T}_S$ equals the target of *reduce*), $\mathcal{T}_S$ is returned. It is returned to $cont_2$, which continues processing at the very end of step 4b of *match* and returns the set $\{\mathcal{T}_S\}$. This set is then is returned to *done*, which computes the union of the set with *matches*. This effectively adds $\mathcal{T}_S$ to *matches*.

2. $S \to a\,b\,c$:

   There are no nullable η to the left of a in the rule, so step 1 of *reduce* is skipped. Therefore, $\vec{\mathcal{T}_\epsilon} = [\,]$. There are two symbols to the right of a in the rule, namely b and c. Thus a continuation $cont_3 = [reduce_4, cont_2, S]$ is created before processing b and c First, b is processed:

   A new continuation $cont_4 = [reduce_3, cont_3, \mathcal{T}_S, [a, b, c], 1]$ is created. Then *match*(b) is called. Since $b \notin nullable(G)$, step 1 of *match* is skipped. This produces yet another continuation, $cont_5 = [match_0, cont_4, b]$, which is added to *continuations*. Then, this

thread needs to wait for a token to be shifted so computation stops.

3. $A \to a\, b$:

   There are no nullable η to the left of a in the rule, so step 1 of *reduce* is skipped. Therefore, $\vec{\mathcal{T}}_\epsilon = [\,]$. There is a single symbol, b, to the right of a. A new continuation $cont_6 = [reduce_4, cont_2, A]$ is created before processing b. Then, b is processed:

   Another new continuation $cont_7 = [reduce_3, cont_6, \mathcal{T}_A, [a, b], 1]$ is created. Then, $match(b)$ is called. Since $b \notin nullable(G)$, step 1 of *match* is skipped. A new continuation, $cont_8 = [match_0, cont_7, b]$ is created and added to *continuations*. This thread now needs to wait for a token to be shifted so computation stops.

The continuation $cont_1$ has now finished processing and the parse state looks like:

$$([a], [b, c], \{\mathcal{T}_S\}, \{cont_5, cont_8\})$$

Then, a *shift* occurs, updating the parse state to

$$([a, b], [c], \{\,\}, \{cont_5, cont_8\})$$

at which point the two continuations are processed. Notice that the algorithm discarded $\mathcal{T}_S$, since there were still tokens in $\vec{t}_{right}$.

First, let us walk through the computation of $cont_5$. Since $b \in T$ and $b = b$ (i.e., the token that was just shifted is equal to the one we're matching), it returns a leaf tree $b\#[\,]$ to its parent, $cont_4$. Then, $cont_4$ resumes computation in the middle of step 3 of *reduce*. There is one more symbol in *right*, namely c. For that, a new continuation, $cont_9 = [reduce_3, cont_3, \mathcal{T}_S, [a, b, c], 2]$ is created. Then, $match(c)$ is called. Since $c \notin nullable(G)$, step 1 of *match* is skipped, and a new continuation, $cont_{10} = [match_0, cont_9, c]$ is created and added to *continuations*. This thread now needs to wait for a token to be shifted so processing stops here.

Now we will consider $cont_8$. Since $b \in T$ and $b = b$ (i.e., the token that was just shifted is equal to the one we're matching), it returns a leaf tree $b\#[\,]$ to its parent, $cont_7$. Then, $cont_7$ resumes computation in the middle of step 3 of *reduce*. There are no more symbols in *right* that need to be matched, so $cont_7$ returns $b\#[\,]$ to its parent, $cont_6$, which resumes computation at the end of step 3 in *reduce*. Then, we get $\vec{\mathcal{T}}_y = [b\#[\,]]$. We construct the parse tree $\mathcal{T}_A = A\#[\mathcal{T}_a] \cdot \vec{\mathcal{T}}_y = A\#[a\#[\,], b\#[\,]]$. Since $A \neq S$ (which was the original target for *reduce*),

a new continuation, $cont_{11} = [reduce_2, cont_2]$. Then, $reduce(A, S, \mathcal{T}_A)$ is called.

We have $neighborhood(A, S) = \{S \to A\, c\}$, so no forking will occur. There are no symbols to the left of A, so step 1 of *reduce* is skipped. We have $\vec{\mathcal{T}}_\epsilon = [\,]$. There is a single symbol, c, to the right of A. Therefore, a new continuation, $cont_{12} = [reduce_4, cont_{11}, S]$, is created. Immediately afterwards, yet another new continuation, $cont_{13} = [reduce_3, cont_{12}, [A, c], 1]$ is created. Then, $match(c)$ is called. Since $c \notin nullable(G)$, step 1 of *match* is skipped. Therefore, a new continuation, $cont_{14} = [match_0, cont_{13}, c]$ is created and added to *continuations*. This thread now needs to wait for a new token to be shifted so processing stops here.

After both $cont_5$ and $cont_8$ were processed, the parse state now looks like:

$$([a, b], [c], \{\,\}, \{cont_{10}, cont_{14}\})$$

Then, *shift* is called again, updating the parse state to

$$([a, b, c], [\,], \{\,\}, \{cont_{10}, cont_{14}\})$$

at which point $cont_{10}$ and $cont_{14}$ are processed.

Let us first consider the computation steps involved in processing $cont_{10}$. It resumes computation immediately after step 2 of *match*. Since $c \in T$ and $c = c$ (the symbol carried by $cont_{10}$ is equal to the symbol that was just shifted), the leaf tree $\mathcal{T}_c = c\#[\,]$ is returned to the parent of $cont_{10}$, which is $cont_9$. Then, $cont_9$ resumes its calculations in the middle of step 3 of *reduce*. It immediately returns $\mathcal{T}_c$ to its parent, $cont_3$, which resumes its calculations at the end of the loop in step 3 of *reduce*. Since there are no more symbols to process in *right*, we have $\vec{\mathcal{T}}_y = [\mathcal{T}_b, \mathcal{T}_c]$. Then, we construct $\mathcal{T}_S = S\#\vec{\mathcal{T}}_\epsilon \cdot [\mathcal{T}_a] \cdot \vec{\mathcal{T}}_y = S\#[\mathcal{T}_a, \mathcal{T}_b, \mathcal{T}_c]$. Since $S = S$ (the label of the root of $\mathcal{T}_S$ is equal to the target of *reduce*), we return $\mathcal{T}_S$ to the parent of $cont_3$, which is $cont_2$. Then, $cont_2$ resumes its calculations at the end of step 4b in $match_1$ and immediately returns the set $\{\mathcal{T}_S\}$ to its parent, *done*. This will compute the union of the set with *matches*, effectively adding $\mathcal{T}_S$ into *matches*.

Now, let us consider the computation steps involved in processing $cont_{14}$.It resumes computation immediately after step 2 of *match*. Since $c \in T$ and $c = c$ (the symbol carried by $cont_{14}$ is equal to the symbol that was just shifted), the leaf tree $\mathcal{T}_c = c\#[\,]$ is returned to the parent of $cont_{14}$, which is $cont_{13}$. Then, $cont_{13}$ resumes its calculations in the middle of step 3 of *reduce*. It immediately returns $\mathcal{T}_c$ to its parent, $cont_{12}$, which resumes its calculations at the end of the

loop in step 3 of *reduce*. Since there are no more symbols to process in *right*, we have $\vec{\mathcal{T}}_y = [\mathcal{T}_c]$. Then, we construct $\mathcal{T}'_S = S\#\vec{\mathcal{T}}_\epsilon \cdot [\mathcal{T}_A] \cdot \vec{\mathcal{T}}_y = S\#[\mathcal{T}_A, c\#[\,]]$. Since $S = S$ (the label of the root of $\mathcal{T}'_S$ is equal to the target of *reduce*), the tree $\mathcal{T}'_S$ is returned to the parent of $cont_{12}$. The parent, $cont_{11}$, resumes its calculations at the very end of step 6b of *reduce* and immediately returns $\mathcal{T}'_S$ to its parent, $cont_2$. Then, $cont_2$ resumes its calculations at the very end of *match*, returning the set $\mathcal{T}'_S$ to its parent, *done*, which computes the union of this set with *matches*, effectively adding $\mathcal{T}'_S$ to *matches*.

Both continuations have now been processed and the parse state now looks like:

$$([a, b, c], [\,], \{\mathcal{T}_S, \mathcal{T}'_S\}\{\,\})$$

Since there are no more tokens to shift, the algorithm finishes, having found two valid parses for the given input. The two parse trees are presented below, with the left tree representing $\mathcal{T}_S$, and the right tree representing $\mathcal{T}'_S$.

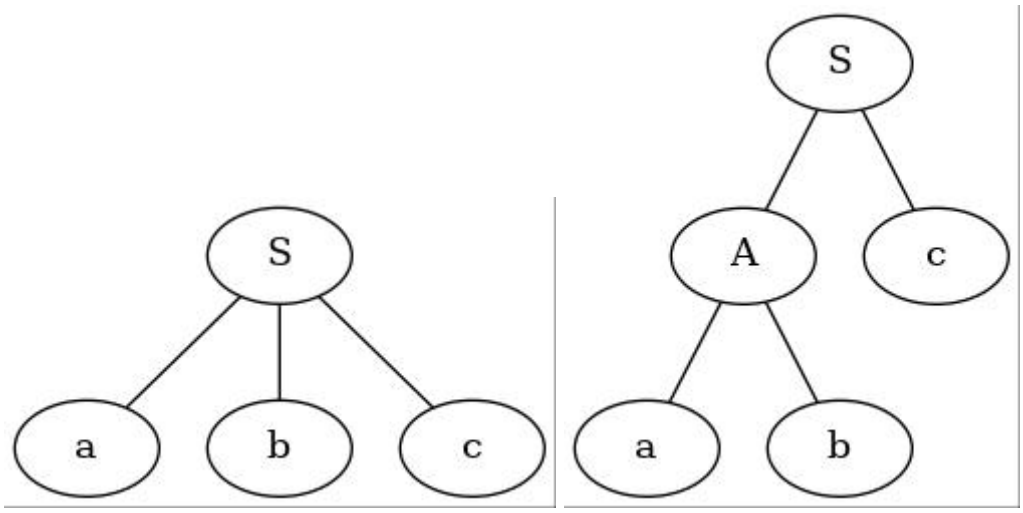


Figure 4.2: The two parse trees found, produced by my visualisation function (see 4.3)

4.3 Test suite

DynGenPar is implemented in C++, with bindings provided for Java as well. I decided to focus my attention on the C++ original. There is roughly 5000 lines of code and documentation comments, spread across two files, `dyngenpar.cpp`, and `dyngenpar.h`. There is also an additional file, `priorityqueue.h`, which contains an implementation of a priority queue, used in the algorithm. I extracted the documentation using Doxygen [9]. I wanted to use Doxygen in

tandem with dot[4] to try and get some insight into the code structure of the software. Since all I managed to generate with them were some dependency diagrams which were not particularly informative, I set about building my own visualisation tools. Although they are very basic, they allowed for better insight and understanding of DynGenPar and evolved into a small test suite for future applications. There is a test suite for the original version of DynGenPar, and a separate, smaller test suite for the trimmed down version.

Parse tree visualisation

Kofler already provides a method for printing out parse trees, which makes use of indentation to represent the tree's hierarchy (see example below)

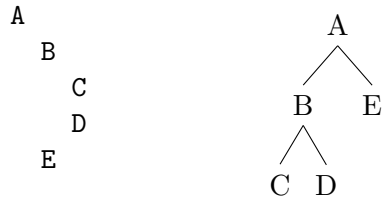


Figure 4.3: An example of Kofler's printing function output (left), and the parse tree it represents (right)

I wanted to use dot, which combines simple syntax with a visualisation engine that can convert text-based descriptions of graphs into images. These text-based descriptions can easily be automatically generated. The naive approach would be to iterate through the parse tree produced by DynGenPar and just create an edge from each node to all of its children. However, if two nodes have the same label, dot would have no means of distinguishing between the two without being provided some form of node ID. This proved to be less straightforward so tried to find an existing tool to solve this problem. I found a simple tool, called ttdot [40] (short for tree-to-dot). It accepts notation very similar to what Kofler's method for printing already outputs. The only difference is that node names in ttdot must be surrounded by a dash on each side, i.e. `-example-`. This meant that only a very slight tweak to Kofler's printing function was needed to output trees in ttdot format. To obtain a dot file (and from that, an image), one must simply run `ttdot <input-file-name.tree> output-file-name.dot`. For ttdot to suit my needs, it required removing a few lines of code which would impose specific formatting onto the dot file (small, circular nodes which could not fit the node labels in). Then, I could

[4]dot is a part of Graphviz [13], which is an open-source visualisation tool. It can produce graphs automatically and outputs in a number of different formats.

produce images of parse trees automatically (since ttdot and dot are run from the command line, writing a bash script to run them in sequence is very straightforward).

```
S
  A
    a
    b
  c


S
  a
  b
  c
```

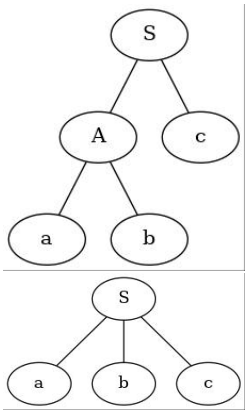


Figure 4.4: The two parse trees printed out in the terminal and their rendered counterparts produced by my visualisation function, for the example worked out in 4.2.4

Initial graph visualisation

The internal representation of the initial graph in the `C++` implementation of DynGenPar is a multi-hash (a hash that can contain multiple values for the same key) between symbols and `FullRules`[5]. Visualising it in dot was just a matter of iterating through the multi-hash and drawing a directed edge from the key to the `cat` argument of the `FullRule` corresponding to said key. For clarity, the edges get labelled with the rule that the `FullRule` represents.

Below is the example grammar given by Kofler in Section 2.1.2, and the corresponding initial graph, created from the dot file using my visualisation function. The code to generate the graph is included in `example-minimal`.

$S \rightarrow Task\ Expr$
$Task \rightarrow min \mid max$
$Expr \rightarrow Expr + Term \mid Term$
$Term \rightarrow Term * Factor \mid Factor$
$Factor \rightarrow x \mid NUMBER$

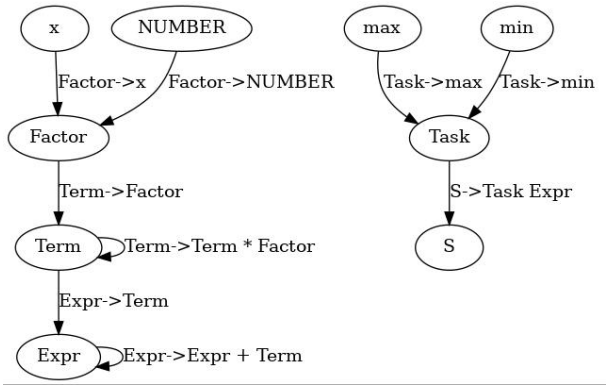


Figure 4.5: The example grammar from Kofler's thesis (left) and the corresponding initial graph (right)

[5]A FullRule is how a rule $(lhs, rhs) \in P$ is represented in the implementation.

Printing to file

Some of the functions in the test suite have nearly identical counterparts, whose job it is to output to a file (usually a `.dot` or `.tree` file) instead of the terminal. Some of these functions are counterparts to functions I wrote myself (e.g., for the initial graph visualisation), and some were just slightly adapted versions of Kofler's functions (e.g., the parse tree visualisation).

4.4 Trimming down

The test suite described above provided very useful information on the behaviour of the program, but in order to determine whether it is suitable for disambiguating mathematical texts, I needed a formal description of what the algorithm does. The source code contained a lot of additional features, which, while useful in practice, had to be removed in order to make understanding the core algorithm easier. All of these additional features are described in Section 2.2.3 of Kofler's thesis [21].

Incrementally, by deleting, recompiling, and running through a test example (to see if the parser still works), I removed, in no particular order:

- **Prediction support**

  The program can supposedly predict what token can follow in order for the sentence to still be valid according to the grammar. Removing it was relatively straightforward, since the algorithm itself never used prediction functionality. It was added so that the user could run prediction in an application which made use of the DynGenPar library. Altogether, prediction support accounted for roughly 1000 lines of code and comments that were removed.

- **Rule actions**

  A rule can have an action attached to it. Actions are fully customizable and must implement the `Action` interface provided by Kofler. A function called `execute` needs to be implemented and gets called once a rule is fully matched (i.e., right after step 5 in *reduce*). It was quite straightforward to remove rule actions, since the check that a rule is matched and execution only happened in one spot in the algorithm.

- **Next token constraints**

  This is the name Kofler uses for the special conditions which allow a grammar to specify

what the token following a given token must/must not be. While useful in practice, they were not needed to understand of the core algorithm. Leaving them in made the code hard to read, since they were checked or modified in many places throughout the algorithm. They also had long variable names, which led to lines of code such as "newNextTokenConstraints.expect.append(pConstraint.first.second.expect);". Removing them was also useful, since it required one less parameter to be carried by the remaining functions and stack items that were used as part of the core algorithm.

- **Support for PMCFGs**

  PMCFGs (parallel multiple context-free grammars) require more work to be implemented than just plain CFGs. Their definition is more complex, making use of an 8-tuple instead of the traditional quadruple used to define a CFG [2]. Supporting PMCFGs in DynGenPar thus involved several dedicated sections of code to be fully implemented. Those were relatively straightforward to identify and remove. There were no separate operations in the algorithm, when the same operation (e.g., *match*) is applied to a PMCFG, though (for example, two separate functions *match_CFG* and *match_PMCFG*). That could have made things more difficult, but the implementation made use of if-else statements to handle either a PMCFG or a CFG, which made it easier to identify and remove parts of the code dedicated to handling PMCFGs. Removing PMCFG support from the implementation accounted for another quite large chunk of code and comments that were removed, and also eliminated some parameters carried by the remaining functions and stack items that were used as part of the core algorithm.

- **Parse tree unification**

  The implementation united parse trees into a shared representation for efficiency (and readability once the parsing is complete) at several places throughout the algorithm. From my understanding, it worked by considering a given set of trees, and if two trees contained a different subtree, but were otherwise identical, it would create a tree which contained both subtrees in that spot. This was done using a special struct, called `Alternative`, which was essentially a sequence of trees. Each parse tree would have a sequence of `Alternatives` as its children. In the case of an unambiguous parse, each node in the parse tree would only contain one `Alternative`. Removing alternatives did not remove significant chunks of code when it comes to the number of lines, but it did remove a

rather complex operation from the overall algorithm. It does come at a cost of efficiency and readability, but it made understanding the algorithm a lot easier, and the increase in runtime for the small grammars and examples I considered so far is negligible. It required slight tweaking when appending trees as children to other trees, since trees now have to be appended to sequences of trees instead of sequences of sequences of trees.

- **Two of the seven stack items**

  The original implementation makes use of seven stack items (continuations). After removing PMCFG support, I realised that types 5 and 6 are not being used any more. It was thus safe to remove them, simplifying the processing of stack items greatly.

- **Stack unification**

  Like parse trees, stacks would also get unified for efficiency. Stacks that were the same up to some parent would get joined together to avoid repeating calculations (unlike our examples, where we computed $match(b)$ multiple times, for example). Removing unification of stacks also did not remove huge portions of code, but it made understanding DynGenPar easier, which is the goal of trimming it down.

- **Support for more token sources**

  Kofler's implementation supports various token sources, such as lists and files, and provides an abstraction for custom implementation. I decided to keep things simple and focus on finite pre-determined sequences of tokens as input. As such, I could remove parts of the abstraction from the implementation. Since the token sources all implement the same interface, removing some of them does not have an effect on the algorithm overall, but it did enable me to fix an input source for reasoning about the algorithm and building example applications.

- **Support for incremental parsing**

  Incremental parsing was also one of the features which did not remove lots of lines of code but simplified things greatly. It enables the parser to finish parsing the input (say, because it reached the end of the input) and then be invoked again to continue processing from the same spot (after additional tokens were added to the input, for example). Because I decided to fix the input to a finite sequence of tokens before to parsing and had no intention of modifying it afterwards, I removed the support for incremental parsing. Before that decision, I did conduct some tests to try and understand incremental parsing and some

of the parameters the parser used, but ultimately I removed all of those, simplifying the parser further to a purely CFG parser.

- **Several arguments carried by the remaining functions**

  The remaining functions and stack items still contained some arguments that were not technically needed for a parse tree to be found. An example of this is the field `len` of the `Match` object (which represents a parse tree). I believe it refers to the number of tokens from the input that the parse tree, represented by the `Match` object parses, but I am not sure. This and other arguments of functions that seemed obsolete were removed to simplify the code and enable me to understand the remaining algorithm better. It only removed tiny fragments of code here and there, but with fewer variables to keep track of while working through examples and reading code it made figuring out how DynGenPar works easier. I am aware that some of these arguments might be useful, for example, in error reporting (as is seen for `len` in the example code Kofler provides). Still, since I was running my tests in a small, controlled environment, I knew the parser should not encounter any errors while parsing.

- **Labels**

  This was a feature which enables the user to add a label to any rule or alternative. Those labels can then be retrieved when printing out parse trees to quickly see what produced them, or when a rule was used. They seem useful in practice, but contribute nothing to the core algorithm and were therefore removed.

- **Priority queue**

  When processing stack items, Kofler first turns a copy of his equivalent of *continuations* from a sequence to a priority queue. I cannot see a good reason to do so, aside from, potentially, efficiency[6]. Therefore, I removed the priority queue in favour of a second list to create a copy of the stacks for processing. This also means that `dyngenpar-minimal.cpp` does not need the additional `priorityqueue.h` file to function properly.

- **QDataStream**

  Some data structures made use of `QDataStream` to implement two functions, `readExternal` and `writeExternal`. I was not sure what their function is, but since the algorithm did

[6] I have not researched the topic as it would require scouring implementation documentation of QList and std::priority_queue to find their respective time complexities, for which I did not have enough time.

not make use of them, I decided to remove them to further simplify the remaining code.

After removing all of this, the trimmed down version contains roughly 1250 lines of code and comments over two files, `dyngenpar-minimal.cpp` and `dyngenpar-minimal.h`, and still supports ambiguous CFGs. Furthermore, instead of all the alternative parsings of an input being represented in one tree (where possible), the program outputs each alternative parsing as a separete tree. All the work I put into trimming the implementation to its current form was crucial to make the code more readable and as such easier to understand. Without it, I would not have been able to produce a formalism of this extent.

4.5 Example applications

To understand what properties and functionality certain parts of the algorithm have, I wrote several small example applications. They are all freely available, alongside the minimal version of DynGenPar. I also provided installation instructions, tested on WSL (Windows Subsystem for Linux) running Ubuntu 20.04. There are also instructions on building, compiling, and running your own applications. To alleviate some of the boilerplate typing, a template for an example application is provided, alongside the project file required by Qt to build it. There is a separate template for the original DynGenPar implementation, and the trimmed down version I created. The template example application contains the example grammar from Kofler's thesis [21], Section 2.1.2, and parses the same input as in Section 2.1.4 of his thesis. An output is produced in the terminal, and also in a `.tree` file. Aside from some minor tweaks and adjustments, the code is taken from `main.cpp` in the DynGenPar source code.

For the examples and test suite to work, some things that were made `private` by Kofler in his implementation of DynGenPar need to be `public`. That is the only edit required to `dyngenpar.h` in order to make everything work. A version of `dyngenpar.h` which contains these changes is provided.

Chapter 5

Conclusions

5.1 Main achievements

1. In Section 2.3.6 I determined that Ganesalingam's algorithm does not exist in any sort of implementation.

2. In Section 2.3.2 I identified DynGenPar, a part of Concise [8], which could be used, as it handles ambiguities and allows for dynamic grammar rule addition.

3. In Section 3.2, I explained why Concise would not be useful as an interface with DynGenPar.

4. In Section 4.1.3, I figured out that the description of DynGenPar found in the literature does not correspond with the C++ implementation and that there is no formal description of either.

5. In Section 4.1.3, I also found a significant difference between the non-deterministic DynGenPar algorithm described in the literature, and the C++ implementation, which concurrently makes all the non-deterministic choices at once. I realised that the details of this concurrent execution and the fact that it even occurs were omitted from the papers written about DynGenPar [21–23].

6. In Section 4.1.3, I also identified the need to produce a formal description of DynGenPar in order to be able to reason about it and make claims about what it does. The closest thing to a formal specification before my work was the description found in the literature [21–23], which, as mentioned, turned out not to be what the implementation does.

7. I realised formalising the entire algorithm would take too much time, so I tried to find anything that could be removed.

8. I hypothesised that there is a core of the C++ implementation that could be isolated, understood, and formalised. In Section 4.4, I describe how I successfully extracted a minimal core of the original DynGenPar implementation that could still parse ambiguous context-free grammars.

9. In Section 4.2.1, I presented a framework within which I could then write a formal specification of the DynGenPar algorithm.

10. In Section 4.2.2, I worked out and presented many of the details of the formal specification of the DynGenPar algorithm.

11. In Section 4.4, I created a list of everything that I removed from the original DynGenPar implementation to extract a minimal core.

12. I created some example applications (see Section 4.5) and a test suite (see Section 4.3) for understanding DynGenPar.

13. I worked out a parsing of an example input with a very simple grammar by hand and presented it in Section 4.2.4, in order to help illustrate how the algorithm works.

5.2 Future work

- The formalism of the minimal version of DynGenPar needs to be completed.
- All the functionality which was removed from the original implementation needs to be formalised and added to the minimal version of DynGenPar.
- The formalism must be checked for correctness.
- After all of this, DynGenPar can be assessed on whether or not it is suitable parse and disambiguate mathematical text.
- A more in-depth review of existing parsing software needs to be conducted to see if any other existing parsers satisfy the criteria for what is needed to parse and disambiguate mathematical text.

Bibliography

[1] *Anaphor*. URL: https://dictionary.cambridge.org/dictionary/english/anaphor (visited on 14/11/2021).

[2] Krasimir Angelov. "Incremental Parsing with Parallel Multiple Context-Free Grammars". In: *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics on - EACL '09*. The 12th Conference of the European Chapter of the Association for Computational Linguistics. Athens, Greece: Association for Computational Linguistics, 2009, pp. 69–76. DOI: 10.3115/1609067.1609074. (Visited on 18/04/2022).

[3] Olga Caprotti and Jordi Saludes. "The GF Mathematical Grammar Library: From OpenMath to Natural Languages". In: CEUR Workshop Proceedings. Vol. 921. July 2012.

[4] Thierry Coquand and Gerard Huet. "The Calculus of Constructions". In: (1988). DOI: 10.1016/0890-5401(88)90005-3.

[5] Marcos Cramer. "Proof-Checking Mathematical Texts in a Controlled Natural Language". PhD thesis. Bonn: Rheinische Friedrich-Wilhelms-Universität, Oct. 2013. URL: https://hdl.handle.net/20.500.11811/5780.

[6] Adrian De Lon et al. "The Isabelle/Naproche Natural Language Proof Assistant". In: *Automated Deduction – CADE 28*. Cham: Springer International Publishing, 2021, pp. 614–624. ISBN: 978-3-030-79876-5.

[7] N.G. de Bruijn. "The Mathematical Vernacular, A Language for Mathematics with Typed Sets". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 133. Elsevier, 1994, pp. 865–935. DOI: 10.1016/S0049-237X(08)70231-3. (Visited on 06/11/2021).

[8] Ferenc Domes. *Concise (Website)*. URL: https://www.mat.univie.ac.at/~dferi/concise.html (visited on 15/11/2021).

[9] *Doxygen*. URL: https://www.doxygen.nl/index.html (visited on 17/04/2022).

[10] *Formalizing 100 Theorems*. Formalizing 100 Theorems. URL: http://www.cs.ru.nl/~freek/100/ (visited on 06/11/2021).

[11] Mohan Ganesalingam. *The Language of Mathematics*. Vol. 7805. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013. DOI: 10.1007/978-3-642-37012-0. (Visited on 28/10/2021).

[12] Adam Grabowski. *Re: State of the Mizar Mathematical Library*. E-mail. 14th Nov. 2021.

[13] *Graphviz*. Graphviz. URL: https://graphviz.org/ (visited on 17/04/2022).

[14] Thomas Hales et al. "A Formal Proof of the Kepler Conjecture". 9th Jan. 2015. arXiv: 1501.02155 [cs, math]. URL: http://arxiv.org/abs/1501.02155 (visited on 27/10/2021).

[15] *HOL Light*. HOL Light. URL: https://www.cl.cam.ac.uk/~jrh13/hol-light/index.html (visited on 06/11/2021).

[16] Muhammad Humayoun and Christophe Raffalli. "MathNat- Mathematical Text in a Controlled Natural Language". In: (July 2010).

[17] Mihnea Iancu. "Towards Flexiformal Mathematics". In: (2017). URL: http://urn-resolving.de/urn:nbn:de:gbv:579-opus-1007213.

[18] Patrick Ion et al. *SemanticWorkshopWhitePaper.Pdf*. May 2016. URL: https://www.wolframfoundation.org/programs/SemanticWorkshopWhitePaper.pdf (visited on 06/11/2021).

[19] Fairouz Kamareddine and J.B. Wells. "Computerizing Mathematical Text with MathLang". In: *Electronic Notes in Theoretical Computer Science* 205 (Apr. 2008), pp. 5–30. DOI: 10.1016/j.entcs.2008.03.063. (Visited on 19/11/2021).

[20] DONALD E KNUTH. "On the Translation of Languages from Left to Right". In: (), p. 33.

[21] Kevin Kofler. "Dynamic Generalized Parsing and Natural Mathematical Language". PhD thesis. Vienna, Austria: University of Vienna, 2017. URL: https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/diss.pdf.

[22] Kevin Kofler and Arnold Neumaier. "A Dynamic Generalized Parser for Common Mathematical Language". In: (), p. 10.

[23] Kevin Kofler and Arnold Neumaier. “DynGenPar – A Dynamic Generalized Parser for Common Mathematical Language”. In: *Intelligent Computer Mathematics*. Vol. 7362. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 386–401. DOI: 10.1007/978-3-642-31374-5_26. (Visited on 19/04/2022).

[24] Michael Kohlhase. *sTeX: An Infrastructure for Semantic Preloading of LaTeX Documents*. sLaTeX: An Ecosystem for Semantically Enhanced LaTeX, 29th Oct. 2021. URL: https://github.com/slatex/sTeX (visited on 29/10/2021).

[25] Michael Kohlhase. “The Flexiformalist Manifesto”. In: *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. Sept. 2012, pp. 30–35. DOI: 10.1109/SYNASC.2012.78.

[26] Michael Kohlhase. “Using LATEX as a Semantic Markup Format”. In: *Mathematics in Computer Science* 2 (Jan. 2008), pp. 279–304. URL: https://kwarc.info/people/mkohlhase/papers/mcs08-stex.pdf.

[27] Alexandre Martin. *Linear Algebra*. 2019.

[28] *MGF GitLab Page*. URL: https://gl.kwarc.info/smglom/GF/-/tree/master (visited on 09/11/2021).

[29] *Mizar*. URL: http://mizar.org/ (visited on 06/11/2021).

[30] *MOLTO (Website)*. URL: http://www.molto-project.eu/ (visited on 09/11/2021).

[31] *Naproche*. URL: http://naproche.net/index.php (visited on 06/11/2021).

[32] Arnold Neumaier. *FMathL - Formal Mathematical Language*. 2011. URL: https://www.mat.univie.ac.at/~neum/FMathL.html (visited on 28/10/2021).

[33] Robert Pagael and Moritz Schubotz. “Mathematical Language Processing Project”. 1st July 2014. arXiv: 1407.0167 [cs].

[34] Andrei Paskevich. “The Syntax and Semantics of the ForTheL Language”. Draft excerpt. Paris, Dec. 2007. URL: http://nevidal.org/download/forthel.pdf.

[35] Aarne Ranta. “Translating between Language and Logic: What Is Easy and What Is Difficult”. In: *Automated Deduction – CADE-23*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Vol. 6803. Lecture Notes in Computer Science. Berlin, Heidelberg:

Springer Berlin Heidelberg, 2011, pp. 5–25. DOI: 10.1007/978-3-642-22438-6_3. (Visited on 20/10/2021).

[36] Seppo Sippu and Eljas Soisalon-Soininen. "On LL(k) Parsing". In: *Information and Control* 53.3 (1982), pp. 141–164. DOI: 10.1016/S0019-9958(82)91016-6.

[37] *SJR - International Science Ranking*. URL: https://www.scimagojr.com/countryrank.php?area=2600&year=2020&order=it&ord=desc (visited on 14/11/2021).

[38] *The Coq Proof Assistant (Website)*. URL: https://coq.inria.fr/ (visited on 06/11/2021).

[39] Masaru Tomita. "The Generalized LR Parsing Algorithm". In: *Generalized LR Parsing*. Springer Science & Business Media, 31st Aug. 1991, pp. 1–16. ISBN: 978-0-7923-9201-9. Google Books: PvZiZiVqwHcC.

[40] *Ttdot: Drawing Trees in Dot*. ttdot: Drawing Trees in Dot. URL: http://www.math.bas.bg/bantchev/ttdot/ttdot.html (visited on 17/04/2022).

[41] Konstantin Verchinine, Alexander Lyaletski and Andrei Paskevich. *System for Automated Deduction (SAD): A Tool for Proof Verification*. 4th Sept. 2007, p. 403. 398 pp. DOI: 10.1007/978-3-540-73595-3_29.

[42] Konstantin Vershinin and Andrey Paskevich. "ForTheL — the Language of Formal Theories". In: *International Journal of Information Theories and Applications* 7.3 (2000), pp. 120–126.