DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

Dynamic Generalized Parsing and Natural Mathematical Language

verfasst von / submitted by

Mag. Kevin Kofler, Bakk.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktor der Naturwissenschaften (Dr.rer.nat)

Wien, 2017 / Vienna, 2017

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 091 405
Dissertationsgebiet lt. Studienblatt / field of study as it appears on the student record sheet:	Mathematik
Betreut von / Supervisor:	Univ.-Prof. Dr. Arnold Neumaier

Abstract

This thesis introduces the dynamic generalized parser DynGenPar and its applications. The parser is aimed primarily at natural mathematical language. It was also successfully used for several formal languages. DynGenPar is available at `https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/`.

The thesis presents the algorithmic ideas behind DynGenPar, gives a short overview of the implementation, documents the applications the parser is currently used for, and presents some efficiency results. Parts of this thesis, mainly in Chapter 2, are based on the refereed conference paper Kofler & Neumaier [56].

The DynGenPar algorithm combines the efficiency of Generalized LR (GLR) parsing, the dynamic extensibility of tableless approaches, and the expressiveness of extended context-free grammars such as parallel multiple context-free grammars (PMCFGs). In particular, it supports efficient dynamic rule additions to the grammar at any moment. The algorithm is designed in a fully incremental way, allowing to resume parsing with additional tokens without restarting the parse process, and can predict possible next tokens. Additionally, it handles constraints on the token following a rule. These allow for grammatically correct English indefinite articles when working with word tokens. They can also represent typical operations for scannerless parsing such as maximal matches when working with character tokens.

Several successful applications of DynGenPar are documented in this thesis. DynGenPar is a core component of the Concise project, a framework for manipulating semantic information both graphically and programmatically, developed at the University of Vienna. DynGenPar is used to parse the formal languages defined by Concise, specifying type systems, programs, and record transformations from one type system to another. Other formal languages with a DynGenPar grammar are a modeling language for chemical processes, a proof-of-concept grammar for optimization problems using dynamic rule additions, and a subset of the AMPL modeling language for optimization problems, extended to also allow intervals wherever AMPL expects a number. DynGenPar can import compiled grammars from the Grammatical Framework (GF) and parse text using them. A DynGenPar grammar also exists for the controlled natural mathematical language Naproche. The use of dynamic rule additions to support mathematical definitions was implemented in a grammar as a proof of concept. There is work in progress on a grammar for the controlled natural mathematical language MathNat. Finally, there is also a well-working DynGenPar grammar for LaTeX formulas from two university-level mathematics textbooks. The long-term goal is to computerize a large library of existing mathematical knowledge using DynGenPar.

Keywords: dynamic generalized parser, dynamic parser, tableless parser, scannerless parser, parser, parallel multiple context-free grammars, common mathematical language, natural mathematical language, controlled natural language, mathematical knowledge management, formalized mathematics, digital mathematical library

Zusammenfassung

Diese Dissertation stellt den dynamischen verallgemeinerten Parser DynGenPar und seine Anwendungen vor. Der Parser zielt hauptsächlich auf natürliche mathematische Sprache ab. Er wurde auch erfolgreich für mehrere formale Sprachen verwendet. DynGenPar ist auf `https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/` verfügbar.

Die Dissertation präsentiert die algorithmischen Ideen hinter DynGenPar, gibt eine kurze Übersicht über die Implementierung, dokumentiert die Applikationen, für die der Parser derzeit verwendet wird, und präsentiert einige Effizienzergebnisse. Teile dieser Dissertation, vor allem in Kapitel 2, basieren auf dem referierten Konferenzartikel Kofler & Neumaier [56].

Der DynGenPar-Algorithmus kombiniert die Effizienz verallgemeinerten LR-Parsens (GLR), die dynamische Erweiterbarkeit tabellenloser Ansätze, sowie die Ausdrucksstärke erweiteter kontextfreier Grammatiken wie paralleler mehrfacher kontextfreier Grammatiken (PMCFG). Insbesondere unterstützt er das effiziente Hinzufügen von Regeln zur Grammatik zu jedem Zeitpunkt. Der Algorithmus ist komplett inkrementell konzipiert, erlaubt also, einen unterbrochenen Parsingvorgang mit zusätzlichen Tokens fortzusetzen, ohne den Parsingvorgang neu zu starten, und kann mögliche nächste Tokens vorhersagen. Zusätzlich behandelt er vorgegebene Einschränkungen (constraints) für den auf eine Regel folgenden Token. Diese erlauben grammatikalisch korrekte englische indefinite Artikel, wenn mit Wörtern als Tokens gearbeitet wird. Sie können auch typische Operationen für scannerloses Parsen wie etwa das Finden einer Übereinstimmung maximaler Länge (maximales Matchen) darstellen, wenn mit Zeichen als Tokens gearbeitet wird.

Mehrere erfolgreiche Anwendungen von DynGenPar sind in dieser Dissertation dokumentiert. DynGenPar ist eine Kernkomponente des Concise-Projekts, eines Frameworks zum Manipulieren semantischer Information sowohl in graphischer als auch in programmatischer Form, entwickelt an der Universität Wien. DynGenPar wird verwendet, um die von Concise definierten formalen Sprachen zu parsen, die Typsysteme, Programme, sowie Recordumwandlungen von einem Typsystem in ein anderes spezifizieren. Andere formale Sprachen mit einer DynGenPar-Grammatik sind eine Modellierungssprache für chemische Prozesse, eine als Machbarkeitsbeweis dienende, dynamisches Hinzufügen von Regeln verwendende Grammatik für Optimierungsprobleme, sowie eine Teilmenge der Modellierungssprache für Optimierungsprobleme AMPL, erweitert, um auch Intervalle zu erlauben, wo immer AMPL eine Zahl erwartet. DynGenPar kann kompilierte Grammatiken des Grammatical Framework (GF) importieren und Text mit ihnen parsen. Eine DynGenPar-Grammatik existiert auch für die kontrollierte natürliche mathematische Sprache Naproche. Die Benutzung dynamischem Hinzufügens von Regeln, um mathematische Definitionen zu unterstützen, wurde in einer Grammatik als Machbarkeitsbeweis implementiert. Eine Grammatik für die kontrollierte natürliche mathematische Sprache MathNat ist in Arbeit. Schließlich gibt es auch eine gut funktionierende DynGenPar-Grammatik für LaTeX-Formeln aus zwei Mathematik-Textbüchern auf Universitätsniveau. Das langfristige Ziel ist, eine große Bibliothek bestehenden mathematischen Wissens mittels DynGenPar zu computerisieren.

Schlagwörter: dynamischer verallgemeinerter Parser, dynamischer Parser, tabellenloser Parser, scannerloser Parser, Parser, parallele mehrfache kontextfreie Grammatiken, gewöhnliche mathematische Sprache, natürliche mathematische Sprache, kontrollierte natürliche Sprache, mathematische Wissensverwaltung, formalisierte Mathematik, digitale mathematische Bibliothek

Contents

Acknowledgements **11**

1 Introduction **13**

1.1 Goals . . . 14
1.2 Related Work . . . 16
1.3 Contents . . . 18

2 The Dynamic Generalized Parser DynGenPar **21**

2.1 The DynGenPar Algorithm . . . 22
2.1.1 Design Considerations . . . 22
2.1.2 The Initial Graph . . . 22
2.1.3 Operations . . . 24
2.1.4 Example . . . 25
2.1.5 Analysis . . . 28
2.2 Implementation . . . 28
2.2.1 Technologies and API . . . 28
2.2.2 Licensing and Availability . . . 29
2.2.3 Implementation Considerations . . . 29

3 DynGenPar and Concise **33**

3.1 Concise . . . 33
3.2 Embedding of DynGenPar into Concise . . . 37
3.3 Concise Features based on DynGenPar . . . 40
3.3.1 Type Sheets . . . 40
3.3.2 Code Sheets . . . 41
3.3.3 Record Transformations . . . 41

3.3.4 Text Views . . . 41
3.3.5 Parsable File Views . . . 44

4 Applications of DynGenPar to Formal Languages 47
4.1 A Grammar for Type Sheets . . . 48
4.2 A Grammar for Code Sheets . . . 53
4.2.1 Elementary Acts . . . 53
4.2.2 Code Sheets . . . 56
4.3 A Grammar for Record Transformation Sheets . . . 59
4.3.1 Basics . . . 60
4.3.2 Document Structure . . . 61
4.3.3 Variables . . . 62
4.3.4 Expressions . . . 63
4.3.5 Path and Path Set Matches . . . 65
4.3.6 Command Blocks . . . 66
4.3.7 Command Lines . . . 70
4.4 A Grammar for Chemical Process Modeling . . . 72
4.5 An Extensible Grammar for Optimization Problems . . . 75
4.6 A Grammar for Robust AMPL . . . 82

5 DynGenPar and the Grammatical Framework (GF) 85
5.1 C Bindings for the GF Haskell Runtime . . . 86
5.2 PGF (Portable Grammar Format) File Import . . . 86
5.3 `PgfTokenSource` – A GF-compatible Lexer . . . 91
5.4 PGF GUI – Graphical Demo Application for Prediction on PGF Grammars 97
5.5 A GF Application Grammar for Mathematical Language . . . 99

6 Applications of DynGenPar to Natural Language 107
6.1 Naproche Parser using DynGenPar . . . 108
6.2 The `TextDocument` Toolchain . . . 110
6.3 `BasicDefinitions` – A Proof of Concept for Dynamic Rule Additions through Mathematical Definitions . . . 116
6.4 `BasicReasoning` – A Concise Grammar based on MathNat . . . 119
6.5 A Grammar for LaTeX Formulas . . . 120

7 Conclusion: Performance Results, Achievements, Extensions **129**

7.1 Performance Benchmark . . . 130

7.1.1 Benchmark of DynGenPar vs. Bison on the Naproche Grammar . . 130

7.1.2 Benchmark of DynGenPar vs. the Grammatical Framework (GF) . 131

7.2 Performance of Dynamic Rule Addition . . . 132

7.3 Test Results on Real-World LaTeX Formulas . . . 134

7.4 Achievements . . . 136

7.5 Future Extensions . . . 138

A The `TypeSheets.cnt` Type Sheet **141**

B Internal Representation (BNF) of the `TypeSheets` Grammar **159**

C Technical Reports **181**

Bibliography **185**

Index **193**

Curriculum Vitae **199**

Acknowledgements

Support by the Austrian Science Fund (FWF) under contract numbers P20631, P23554 and P22239 is gratefully acknowledged. In addition, I would like to thank:

- my dear mother, Hannelore Kofler-Brugnolo, who has supported me throughout my studies in many ways,
- my late father, Gerhard Kofler, who sadly passed away in 2005, but whose love, kindness and moral support will always give me strength,
- my supervisor, Arnold Neumaier, without whom this thesis would not have been possible. He came up with the project idea and provided me with countless insights and input.
- Hermann Schichl, who had supervised my diploma thesis and who also provided a lot of valuable input for this PhD thesis,
- my colleagues Peter Schodl and Ferenc Domes, who developed the applications DynGenPar is now used with, and with whom I had countless productive discussions,
- Tiago de Morais Montanher, who gave valuable feedback as a reader not directly involved with the Concise project,
- Stephan Paukner, who wrote the LaTeX template for the title page meeting the formal requirements for the title page of a PhD thesis at the University of Vienna,
- the participants of the Conferences on Intelligent Computer Mathematics (CICM) 2011 and 2012 for their valuable input,
- the Grammatical Framework (GF) developers for their insights on parsing natural language.

Chapter 1

Introduction

Parsing is the act of transforming some properly structured **input text** (**input sentence**) into a form understandable by a computer. Usually, this is a tree representation called a **parse tree**. A **parser** is a program that parses any given properly structured input text using a set of rules called a **grammar**.

The most commonly used type of grammar is the **context-free grammar** (Chomsky [12, 13]). Computer scientists (e.g., Aho & Ullman [1]) define a context-free grammar as a tuple $G = (N, T, P, S)$ (traditionally given in that order), where

- T is the **alphabet**, a set of symbols called **tokens**. In order to be parsable, the input text must consist exclusively of symbols from the alphabet T.
- N is a set of **nonterminals** (also called **categories**), disjoint from T.
- P is a set of **productions** (**rules**) of the form $n \rightarrow \alpha_1\ \alpha_2\ \ldots\ \alpha_k$, where $n \in N$ and $\alpha_i \in N \cup T$ for all i. Those rules define which input sentences can be parsed and how the resulting parse trees look like.
- $S \in N$ is the **start symbol** (also called **start category**).

The produced parse tree consists of the steps to go from the start symbol S to the input text, where each step replaces the left hand side of a production from P with its right hand side.

The popularity of context-free grammars is due to the efficiency of parsing text with them. More general types of grammars exist (Chomsky [12]) but require exponential time or even infinite time to parse with. Therefore, where context-free grammars are not powerful enough, the common way to extend them is to add additional **context-sensitive constraints** to some or all productions. Those constraints determine whether or not a rule can be applied in a particular context. They can be verified during or after parsing. In this way one obtains a more powerful grammar formalism while retaining the efficiency of context-free grammars.

This thesis presents **DynGenPar** (**Dynamic Generalized Parser**), a parser aimed primarily at natural mathematical language, and its applications. The applications that are presented are grammars for several formal languages and some grammars related to natural mathematical language.

DynGenPar combines

- the efficiency of table-driven parsing algorithms such as **LR** (Knuth [45]) or **Generalized LR** (**GLR**) (Tomita [91], Tomita & Ng [92]),
- the dynamic extensibility of tableless parsing algorithms, and
- the expressiveness of context-free grammars extended with context-sensitive constraints.

In this chapter, first, the goals of the research underlying this thesis are presented. Then, an overview of state-of-the-art related work is given. Finally, the contents of the remaining chapters of this thesis are summarized.

1.1 Goals

I have developed DynGenPar with a very ambitious primary target application in mind: the **FMathL** (**Formal Mathematical Language**) project (Neumaier [70]). FMathL is the working title for a modeling and documentation language for mathematics, suited to the habits of mathematicians, to be developed at the University of Vienna. The project complements efforts for formalizing mathematics from the computer science and automated theorem proving perspective. In the long run, the FMathL system might turn into a user-friendly automatic mathematical assistant for retrieving, editing, and checking mathematics (but also computer science and theoretical physics) in both informal, partially formalized, and completely formalized mathematical form.

Unfortunately, due to limited funding and time, only small parts of FMathL have been realized so far. One of them is **Concise** (Schodl et al. [87], Domes [22]), a framework and GUI (graphical user interface) for viewing and manipulating, both graphically and programmatically, semantic information in graph form. Concise has been developed primarily by Ferenc Domes. Some parts have been added or improved by me. Concise is discussed in more detail in Section 3. The second major output of the FMathL project to have been completed is my parser DynGenPar. Based on Concise and DynGenPar, grammars for several formal languages and subsets of natural mathematical language were developed. DynGenPar and the grammars developed for it are the subject of this thesis.

Many of the design requirements for DynGenPar were studied with the long-term goals of the FMathL project in mind. In particular, the aim is to be able to parse a controlled language that is as close as possible to commonly-used natural mathematical language. That long-term plan means that the current applications make only limited use of the parser's characterizing features. However, one can expect DynGenPar to be very useful for many projects that either have similar goals as FMathL, or just happen to have similar requirements. In particular, the following requirements (Kofler & Neumaier [56]) were imposed as a result of the FMathL project planning: The parser should:

- allow the efficient incremental addition of new rules to the grammar (e.g., when a definition is encountered in a mathematical text) at any time, without recompiling the whole grammar;

- be able to parse more general grammars than just LR(1) (Knuth [45]) or LALR(1) (DeRemer [20]) ones: Natural language is usually not LR(1). In addition, being able to parse so-called **parallel multiple context-free grammars** (**PMCFGs**) (Seki et al. [88]) is a necessity for reusing the natural language processing facilities of the **Grammatical Framework** (**GF**) (Ranta [77, 78], Ranta et al. [79]);
- exhaustively produce all possible parse trees (in a packed representation), in order to allow later semantic analysis to select the correct alternative from an ambiguous parse, at least as long as their number is finite;
- support processing text incrementally, i.e., resuming parsing with additional tokens without restarting the parse process, and predicting the next token (*predictive parsing*);
- be transparent enough to allow formal verification and implementation of error correction in the future;
- support both scanner-driven (for common mathematical language) and scannerless (for some other parsing tasks in the implementation) operation.

These requirements, especially the first one, rule out all efficient parsers currently in use.

I solved these challenges with an algorithm loosely modeled on **Generalized LR** (**GLR**) (Tomita [91], Tomita & Ng [92]), but with an important difference: To decide when to shift a new token and which rule to reduce when, GLR uses complex LR states. In practice, these are mostly opaque entities, which have to be recomputed completely each time the grammar changes and which can grow very large for natural-language grammars. In contrast, my algorithm uses a so-called **initial graph**, which is easy and efficient to update incrementally as new rules are added to the grammar, along with runtime top-down information. (Kofler & Neumaier [56]) The details will be presented in the next section.

This approach allows my algorithm to be both *dynamic*:

- The grammar is not fixed in advance.
- Rules can be added at any moment, even during the parsing process.
- No tables are required. The graph I use instead can be updated very efficiently as rules are added.

and *generalized*:

- The algorithm can parse general PMCFGs.
- For ambiguous grammars, all possible syntax trees are produced.

(Kofler & Neumaier [56])

Additionally, the algorithm handles constraints on the token following a rule. This has several applications in practice. For instance, when working with word tokens, this allows for grammatically correct English indefinite articles, e.g., ***a*** *set*, but ***an*** *operator*. When working with character tokens, it can represent typical operations for scannerless parsing such as maximal matches, e.g., when we are trying to match an integer, if *1234* matches, do not accept just *123*.

Constraints, both those required to parse general PMCFGs and the next token constraints described above, are handled transparently during parsing. No separate post-processing step is needed. Compared to the naive approach of first parsing exhaustively and then validating the constraints, the selected one-step approach ensures maximum efficiency.

I expect this parsing algorithm to allow parsing a large subset of common mathematical language and help building a large database of computerized mathematical knowledge. Additionally, one can envision potential applications outside of mathematics, e.g., for domain-specific languages for special applications (Mernik et al. [66]). These are currently mainly handled by scannerless parsing using GLR (Visser [95]) for context-free grammars (CFGs), but would benefit a lot from my incremental approach. Therefore, DynGenPar is an important achievement independently of whether FMathL will ultimately succeed or not. (Kofler & Neumaier [56])

The new parser is available at `https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/`.

1.2 Related Work

This section is based in part on Kofler & Neumaier [56].

No current parser generator combines all partially conflicting requirements mentioned above.

Ambiguous grammars are usually handled using **Generalized LR** (**GLR**) (Tomita [91], Tomita & Ng [92]), needing the compilation of a GLR table, which can take several seconds or even minutes for large grammars. Such tables can grow extremely large for natural-language grammars. In addition, my parser also supports PMCFGs, whereas GLR only works for context-free grammars (but it may be possible to extend GLR to PMCFGs using my techniques). Costagliola et al. [16] present a predictive parser **XpLR** (**eXtended Positional LR**) for visual languages. However, in both cases, since the tables used are mostly opaque, they have to be recomputed completely each time the grammar changes.

The well-known **CYK** (**Cocke-Younger-Kasami**) algorithm (Kasami [43], Younger [101]) needs no tables, but is very inefficient and handles only CFGs. Hinze & Paterson [36] propose a more efficient tableless parser; their idea has not been followed up by others.

The most serious competitor to my parser is Angelov's PMCFG parser (Angelov [3]) as found in the code of the **Grammatical Framework** (**GF**) (Ranta [77, 78], Ranta et al. [79]), which has some support for natural language and predictive parsing. Alanko and Angelov have recently developed a C version in addition to the existing Haskell implementation. Compared to Angelov's parser, I offer similar features with a radically different approach, which I hope will prove better in the long run. In addition, my implementation already supports features such as incremental addition of PMCFG rules which are essential for my application, which are not implemented in Angelov's current code and which may or may not be easy to add to it. My parser also supports importing compiled PGF (Portable Grammar Format) (Angelov et al. [4]) files from GF, allowing

to reuse the remainder of the Grammatical Framework. When doing so, as evidenced in Section 7.1.2, it reaches a comparable performance. My implementation can also enforce next token constraints, e.g., *an restaurant* is not allowed.

Another important feature of DynGenPar is the support for parsing both with and without a separate lexer (scanner). This is also not a new idea. Scannerless parsing has been originally developed by Salomon & Cormack [81] for programming languages. Visser [95] extended the approach to GLR (Generalized LR), which makes it applicable to general context-free grammars, even ambiguous ones. Additional strategies for disambiguation are discussed in Van Den Brand et al. [93]. This allows the technique to be applied to more flexible domain-specific languages. However, the use of the table-driven LR or GLR algorithm limits these approaches to static languages. In contrast, with DynGenPar, the domain-specific language can be dynamically extended at runtime.

Research on natural language input of mathematics has been driven primarily by the proof assistant community. The formal languages of proof checkers are hard to read and write. Therefore, several ways to make use of natural language to input proofs have been attempted. Hallgren & Ranta [35] make use of GF to allow natural language input in Hallgren's proof editor **Alfa** (Hallgren [34]). Alfa can convert between its supported controlled subset of natural language and formal language in both directions. The limiting factor is the grammar of the controlled natural language. The authors note that in Alfa, "natural-language input is only useful for small expressions, since entering a long expression runs the risk of falling outside the grammar" (Hallgren & Ranta [35]). They work around this issue by allowing interactive input of small expressions at a time. That approach is driven one step further by the **HLM Proof Assistant** (Reichelt [80]), which completely gives up on text as an input format. In HLM, proofs can only be input interactively through menu entries. A natural language representation is used purely as a visualization. Another research direction was taken by Koepke et al. [46], who designed **Naproche** (Cramer et al. [17], Koepke et al. [46]), a text-based controlled natural language for proof checkers. In Naproche, entire proofs can be written in text form. This approach was the most interesting for my research team, and therefore, Naproche was one of the first grammars I implemented in DynGenPar. The DynGenPar implementation of Naproche will be presented in Section 6.1. However, Naproche is also a fairly small subset of natural language. More recently, Humayoun [40] introduced the **MathNat** language (Humayoun & Raffalli [41], Humayoun [39]), based on GF. MathNat is probably the most powerful controlled natural language for proof checkers to date. Therefore, MathNat was chosen as the starting point for the upcoming DynGenPar-based controlled natural language grammar. The state of that grammar's ongoing development will be discussed in Section 6.4.

DynGenPar was not developed in isolation. The research documented in this thesis is part of the **FMathL** (**Formal Mathematical Language**) umbrella project (Neumaier [70]), a modeling and documentation language for mathematics, suited to the habits of mathematicians, to be developed at the University of Vienna. DynGenPar is closely related to and interoperates with **Concise** (Schodl et al. [87], Domes [22]), a framework and GUI (graphical user interface) for viewing and manipulating, both graphically and programmatically, semantic information in graph form. The PhD thesis Schodl [84]

introduces the semantic memory and the type system underlying Concise. The output produced by DynGenPar is stored in the Concise semantic memory and conforms to the Concise type system. The technical report Schodl & Neumaier [85] documents research done by its authors on the grammar of a university-level introductory textbook on analysis and linear algebra (Neumaier [69]). The output from that research was a source of inspiration for the natural language grammars presented in this thesis. The user manual Domes et al. [25] documents the Concise GUI for potential users. The Concise package includes DynGenPar, and the ways to access DynGenPar from the Concise user interface are also documented in the Concise manual. The technical reports that I produced in the context of FMathL, Concise, and DynGenPar are listed in Appendix C.

1.3 Contents

Chapter 2, based on the published conference paper Kofler & Neumaier [56], presents the algorithm and implementation of DynGenPar. This parser is the main product of the thesis, and the central piece on which almost all the research is based. The remaining chapters document mainly applications based on DynGenPar grammars.

Chapter 3 introduces Concise (Schodl et al. [87], Domes [22]), a framework and GUI for viewing and manipulating, both graphically and programmatically, semantic information in graph form. The focus is on the interaction between DynGenPar and Concise and on the features in Concise enabled by DynGenPar.

Chapter 4 describes some formal languages on which DynGenPar is currently used. These applications constitute some successful practical uses of DynGenPar, although they make only limited use of the parser's unique features. The grammars that are featured are:

- Three formal languages used within Concise: type sheets, code sheets, and record transformation sheets
- A language for chemical process modeling
- An extensible grammar for optimization problems demonstrating the dynamic grammar extensibility of DynGenPar
- A grammar for a subset of the AMPL (Fourer et al. [27], AMPL Optimization inc. [2]) modeling language for optimization problems, extended to allow intervals wherever AMPL normally expects a number

Chapter 5 introduces the Grammatical Framework (GF) (Ranta [77, 78], Ranta et al. [79]) for natural language processing and how DynGenPar interoperates with it. In particular, it discusses how DynGenPar can operate on compiled GF grammars in the PGF (Portable Grammar Format) file format. A GUI application demonstrating the prediction functionality of DynGenPar on PGF grammars and a proof of concept for a GF application grammar for a tiny subset of natural mathematical language are also described.

Chapter 6 documents progress towards the research goal that has driven DynGenPar development. It presents applications of DynGenPar to various controlled subsets of natural language:

- Naproche (Cramer et al. [17], Koepke et al. [46])
- A proof of concept for the use of dynamic rule additions to implement mathematical definitions
- MathNat (Humayoun & Raffalli [41], Humayoun [40, 39])
- A subset of natural LaTeX formula notation (without dedicated semantic markup)

Finally, Chapter 7 wraps up the thesis by presenting some practical results obtained with DynGenPar, summarizing what was achieved through DynGenPar, and giving an outlook on possible future extensions. In addition to several performance results, success rates for the grammar for LaTeX formulas on the formulas extracted without modification from two university-level introductory mathematics textbooks (Neumaier [69], Schichl & Steinbauer [83]) are shown.

Chapter 2

The Dynamic Generalized Parser DynGenPar

This chapter, based on the published conference paper Kofler & Neumaier [56], presents the algorithm and implementation of **DynGenPar**, the **Dynamic Generalized Parser**. This parser is the main product of the thesis, and the central piece on which almost all the research is based. The remaining chapters of this thesis document mainly applications based on DynGenPar grammars.

DynGenPar combines

- the efficiency of table-driven parsing algorithms such as **LR** (Knuth [45]) or **Generalized LR** (**GLR**) (Tomita [91], Tomita & Ng [92]),
- the dynamic extensibility of tableless parsing algorithms, and
- the expressiveness of context-free grammars extended with context-sensitive constraints.

This chapter documents how these goals have been achieved.

The first section of this chapter describes the theoretical DynGenPar algorithm. It explains the requirements that had to be considered in the design of the algorithm, precisely describes the algorithm with the help of an example, and gives a short analysis of its efficiency properties. The second (and last) section gives an overview of the practical DynGenPar implementation. It documents the selected technologies, the licensing choices, and the tweaks that were made to the implementation to enhance efficiency and functionality.

The algorithm was first presented in the technical report Kofler & Neumaier [54], an expanded version of which was later published as Kofler & Neumaier [56].

2.1 The DynGenPar Algorithm

In this section, I describe the basics of my algorithm. (Details about the implementation of some features will be presented in Section 2.2.3.) I start by explaining the design considerations which led to my algorithm. Next, I define the fundamental concept of my algorithm: the initial graph. I then describe the algorithm's fundamental operations and give an example of how they work. Finally, I conclude the section by analyzing the algorithm as a whole.

2.1.1 Design Considerations

My design was driven by multiple fundamental considerations.

My first observation was that I wanted to handle left recursion in a most natural way, which has driven me to a bottom-up approach such as LR (Knuth [45]). In fact, my first attempt at solving the problems at hand was using a recursive-descent algorithm. That prototype was quickly discarded because there was no satisfactory way to handle left recursion with such an approach. Doing the transformation automatically would have led to unnecessarily complex code with doubtful efficiency. Requiring it to be done manually by the user would have meant restricting the possible parse trees that can be produced. In particular, left-associative arithmetic is naturally left-recursive. Natural language grammars contain left recursion as well. It is important for the corresponding parse trees to be produced accurately. Thus, a bottom-up approach was chosen.

My next finding was that the desired applications require supporting general context-free grammars (and even extensions such as PMCFGs). Such a need implies that algorithms based purely on deterministic stack automata are not suitable. Such algorithms are restricted to grammar families that can be parsed deterministically (e.g., LALR(k), LR(k), or LL(k)) and cannot handle arbitrary context-free grammars, even unambiguous ones. Instead, even in the unambiguous case, an approach considering multiple alternatives at a time (corresponding to a nondeterministic stack automaton) is needed, such as Generalized LR (GLR) (Tomita [91], Tomita & Ng [92]). Another important advantage of such an algorithm is that it can also enumerate all the possibilities in the case of an ambiguous grammar, which was also a desired property for my algorithm.

However, my main requirement, i.e., allowing to add rules to the grammar at any time, disqualifies table-driven algorithms such as GLR: recomputing the table is prohibitively expensive, and doing so while the parsing is in progress is usually not possible at all. Therefore, I had to restrict myself to information which can be produced dynamically. This also required introducing some top-down steps into the bottom-up algorithm. The dynamic information that is used in the DynGenPar algorithm is described below.

2.1.2 The Initial Graph

To fulfill the above requirements, I designed a data structure called the **initial graph**. Consider a context-free grammar $G = (N, T, P, S)$, where N is the set of nonterminals,

T the set of tokens, P the set of productions (rules), and S the start symbol. Then the initial graph corresponding to G is a directed labeled multigraph on the set of symbols $\Gamma = N \cup T$ of G, defined by the following criteria:

- The tokens T are sources of the graph.
- The graph has an edge from the symbol $s \in \Gamma$ to the nonterminal $n \in N$ if and only if the set of productions P contains a rule p: $n \to n_1\ n_2\ \ldots\ n_k\ s\ \ldots$ with $n_i \in N_0$ for all i, where $N_0 \subseteq N$ is the set of all those nonterminals from which the empty string ε can be derived. The edge is labeled by the pair (p, k), i.e., the rule (production) p generating the edge and the number k of n_i set to ε.
- In the above, if there are multiple valid (p, k) pairs leading from s to n, I define the edge as a multi-edge with one edge for each pair (p, k), labeled with that pair (p, k).

This graph serves as the replacement for precompiled tables and can easily be updated as new rules are added to the grammar.

For example, consider the following toy grammar for unconstrained polynomial optimization problems $G = (N, T, P, S)$, with

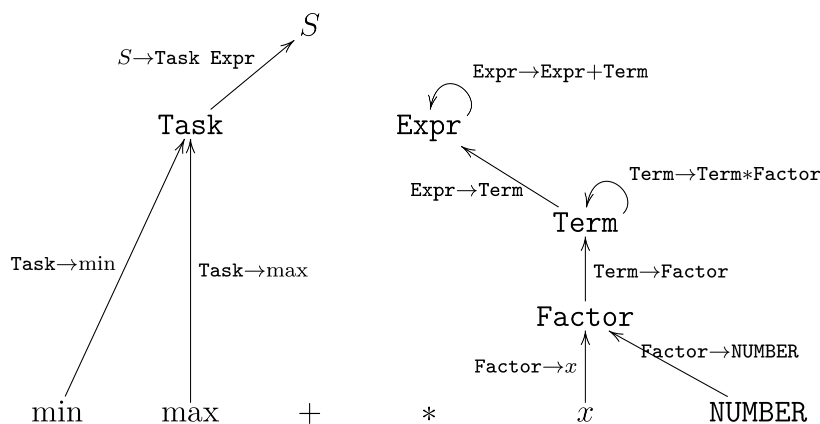
$$N = \{S, \mathtt{Task}, \mathtt{Expr}, \mathtt{Term}, \mathtt{Factor}\}, T = \{\min, \max, +, *, x, \mathtt{NUMBER}\},$$

and where P contains the following rules:

- $S \to \mathtt{Task}\ \mathtt{Expr}$,
- $\mathtt{Task} \to \min \mid \max$,
- $\mathtt{Expr} \to \mathtt{Expr} + \mathtt{Term} \mid \mathtt{Term}$,
- $\mathtt{Term} \to \mathtt{Term} * \mathtt{Factor} \mid \mathtt{Factor}$,
- $\mathtt{Factor} \to x \mid \mathtt{NUMBER}$.

The token NUMBER stands for a constant number, and would in practice have a value, e.g., of `double` type, attached.

The initial graph looks as follows:



It shall be noted that there is no edge from, e.g, Expr to S, because Expr appears not at the beginning, but only in the middle of the rule $S \to \mathtt{Task}\ \mathtt{Expr}$ (and the Task that precedes it cannot produce the empty string ε).

Let $s \in \Gamma = N \cup T$ be a symbol and $z \in N$ be a nonterminal (called the **target**). The **neighborhood** $\mathcal{N}(s, z)$ is defined as the set of edges from s to a nonterminal $n \in N$ such that the target z is reachable (in a directed sense) from n in the initial graph. Those neigborhoods can be computed relatively efficiently by a graph walk and can be cached as long as the grammar does not change.

In the example, we would have, e.g., $\mathcal{N}(\text{min}, S) = \{\texttt{Task} \to \text{min}\}$, $\mathcal{N}(x, S) = \emptyset$ (because there is no path from x to S), $\mathcal{N}(x, \texttt{Expr}) = \{\texttt{Factor} \to x\}$, and $\mathcal{N}(\texttt{Term}, \texttt{Expr}) = \{\texttt{Expr} \to \texttt{Term}, \texttt{Term} \to \texttt{Term} * \texttt{Factor}\}$. (In the last example, we also have to consider the loop, i.e., the left recursion.)

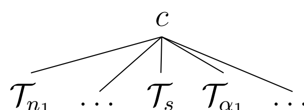
2.1.3 Operations

Given these concepts, we define four **elementary operations**:

- $match_\varepsilon(n), n \in N_0$: This operation derives n to ε. It works by top-down recursive expansion, simply ignoring left recursion. This is possible because left-recursive rules which can produce ε necessarily produce infinitely many syntax trees, and I decided to require exhaustive parsing only for a finite number of alternatives.
- *shift*: This operation simply reads in the next token, just as in the LR algorithm.
- $reduce(s, z), s \in \Gamma, z \in N$: This operation reduces the symbol s to the target nonterminal z. It is based on and named after the LR reduce operation, however it operates differently: Whereas LR only reduces a fully matched rule, my algorithm already reduces after the first symbol. This implies that my *reduce* operation must complete the match. It does this using the next operation:
- $match(s), s \in \Gamma = N \cup T$: This operation is the main operation of the algorithm. It matches the symbol s against the input, using the following algorithm:
  1. If $s \in N_0$, try ε-matches first: $m_\varepsilon := match_\varepsilon(s)$. Now the algorithm only needs to look for nonempty matches.
  2. Start by shifting a token: $t := shift$.
  3. If $s \in T$, the algorithm just needs to compare s with t. If they match, it returns a leaf as its parse tree, otherwise it returns no matches at all.
  4. Otherwise (i.e., if $s \in N$), it returns $m_\varepsilon \cup reduce(t, s)$.

Given the above operations, the algorithm for $reduce(s, z)$ can be summarized as follows:

1. Pick a rule $c \to n_1\ n_2\ \ldots\ n_k\ s\ \alpha_1\ \alpha_2\ \ldots\ \alpha_\ell$ in the neighborhood $\mathcal{N}(s, z)$.
2. For each n_i ($\in N_0$ by definition of the neighborhood): $\mathcal{T}_{n_i} := match_\varepsilon(n_i)$.
3. s was already recognized, let $\mathcal{T}_s$ be its syntax tree.
4. For each $\alpha_j \in \Gamma = N \cup T$: $\mathcal{T}_{\alpha_j} := match(\alpha_j)$. Note that this is a top-down step, but that the *match* operation will again do a bottom-up shift-reduce step.
5. The resulting syntax tree is:



6. If $c \neq z$, continue reducing recursively (*reduce*(c, z)) until the target z is reached. The algorithm also needs to consider *reduce*(z, z) to support left recursion; this is the only place in my algorithm where I need to accomodate specifically for left recursion.

In the case of a conflict between multiple possible *reduce* operations, the algorithm needs to consider all the possibilities. It then unifies its matched parse trees into **directed acyclic graphs** (**DAGs**) wherever possible to both reduce storage requirements and prevent duplicating work in the recursive *reduce* steps. This is described in more detail in Section 2.2.3.2.

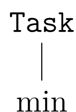
The algorithm is initialized by calling *match*(S) on the start symbol S of the grammar. Conceptually, the remainder happens recursively. The exact sequence of events in my practical implementation, which allows for predictive parsing, is described in Section 2.2.3.1.

2.1.4 Example

As an example, we show how my algorithm works on the basic grammar for unconstrained polynomial optimization problems from Section 2.1.2. The example was chosen to be didactically useful rather than realistic: In later chapters, we shall use the algorithm for grammars significantly more complex than this example. It shall be noted that in this example, the set N_0 of nonterminal which can be derived to ε is empty. Handling ε-productions requires some technical tricks (skipped initial nonterminals with empty derivation in rules, $match_\varepsilon$ steps), but does not impact the fundamental algorithm.

Consider the input `min x * x`, a valid sentence in the example grammar. We will denote the cursor position by a dot, so the initial input is `.min x * x`. The algorithm always starts by matching the start category, thus the initial step is *match*(S). The *match* step starts by shifting a token, then tries to reduce it to the symbol being matched. In this case, the *shift* step produces the token `min`, the input is now `min.x * x`, and the next step is *reduce*(min, S), after which the parsing is complete.

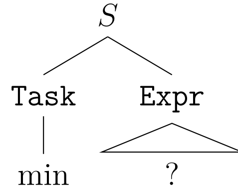
It is now the job of the *reduce* task to get from `min` to S, and to complete the required rules by shifting and matching additional tokens. To do this, it starts by looking for a way to get closer towards S, by looking at the neighborhood $\mathcal{N}(\text{min}, S) = \{\texttt{Task} \rightarrow \text{min}\}$. In this case, there is only one rule in the neighborhood, so the algorithm reduces that rule. The right hand side of the rule is just `min`, so the rule is already completely matched, there are no symbols left to match. The algorithm remembers the parse tree



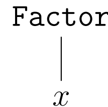
and proceeds recursively with $\mathit{reduce}(\texttt{Task}, S)$.

Now we have $\mathcal{N}(\texttt{Task}, S) = \{S \to \texttt{Task Expr}\}$. There is again only a single rule that matches, but this time there is an α_j left to match: $\alpha_1 = \texttt{Expr}$. Thus, the algorithm needs to $\mathit{match}(\texttt{Expr})$; then, the reduce will be complete, because the left hand side S of the matching rule is already the goal.

So far, the algorithm has recognized the parse tree

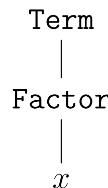


and needs to $\mathit{match}(\texttt{Expr})$. The shift step produces x, leaving it with the input `min x.* x`. As before, the algorithm proceeds with a $\mathit{reduce}(x, \texttt{Expr})$ and looks at the neighborhood $\mathcal{N}(x, \texttt{Expr}) = \{\texttt{Factor} \to x\}$. It reduces again the only matching rule, and since the right hand side of the rule is just x, the rule is already completely matched. The algorithm remembers the parse tree



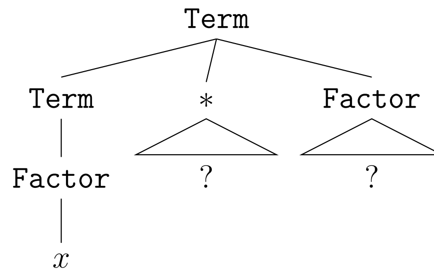
and proceeds recursively with $\mathit{reduce}(\texttt{Factor}, \texttt{Expr})$.

Now we have $\mathcal{N}(\texttt{Factor}, \texttt{Expr}) = \{\texttt{Term} \to \texttt{Factor}\}$. Again, there is only a single rule that matches and it is fully matched, so the algorithm reduces it, remembers the parse tree

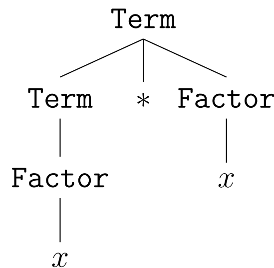


and continues the recursion with $\mathit{reduce}(\texttt{Term}, \texttt{Expr})$.

This time, the neighborhood $\mathcal{N}(\texttt{Term}, \texttt{Expr}) = \{\texttt{Expr} \to \texttt{Term}, \texttt{Term} \to \texttt{Term} * \texttt{Factor}\}$ contains more than one matching rule, we have a **reduce-reduce conflict**. Therefore, the algorithm has to consider both possibilities, as in GLR. If it attempts to reduce $\texttt{Expr} \to \texttt{Term}$, the parsing terminates here (or it tries reducing the left-recursive $\texttt{Expr} \to \texttt{Expr}+\texttt{Term}$ rule and hits an error on the unmatched $+$ token, highlighted in red in the rule), but the input is not consumed yet, thus it hits an error. Therefore, it retains only the option of reducing the left-recursive $\texttt{Term} \to \texttt{Term} * \texttt{Factor}$ rule. This time, there are two remaining symbols: $\alpha_1 = *$ and $\alpha_2 = \texttt{Factor}$, thus the algorithm proceeds with $\mathit{match}(*)$ and $\mathit{match}(\texttt{Factor})$. The parse tree matched so far is

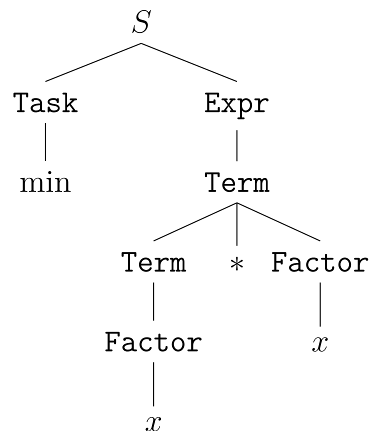


The *match*($*$) operation is trivial: $*$ is a token, so the algorithm only needs to *shift* the next token and compare it to $*$. The input is now `min x *.x`, and the *match*(`Factor`) step proceeds by a last *shift* consuming the last token and yielding the final input `min x * x.`, and a *reduce*(x, `Factor`) which is also trivial because $\mathcal{N}(x, \texttt{Factor}) = \{\texttt{Factor} \rightarrow x\}$. This completes the subtree



Thus the reduction of the left-recursive rule $\texttt{Term} \rightarrow \texttt{Term} * \texttt{Factor}$ is complete and the algorithm recursively proceeds with another *reduce*(`Term`, `Expr`). This time, attempting to reduce the left-recursive rule again yields an error (there is no input left to match the red $*$ in $\texttt{Term} \rightarrow \texttt{Term}*\texttt{Factor}$ against) and the algorithm reduces $\texttt{Expr} \rightarrow \texttt{Term}$. Once again, attempting to reduce the left-recursive rule $\texttt{Expr} \rightarrow \texttt{Expr}+\texttt{Term}$ fails because there is no input to match the red $+$, thus the algorithm is done.

The final parse tree



is obtained, which is identical to what other parsing algorithms such as LR would have produced.

2.1.5 Analysis

The above algorithm keeps efficiency by combining enough bottom-up techniques to avoid trouble with left recursion with sufficient top-down operation to avoid the need for tables. The initial graph ensures that the bottom-up steps never try to reduce unreachable rules, which is the main inefficiency in existing tableless bottom-up algorithms such as **CYK** (**Cocke-Younger-Kasami**) (Kasami [43], Younger [101]).

One disadvantage of the algorithm is that it produces more conflicts than LR or GLR, for two reasons: It is not able to make use of any lookahead tokens, unlike common LR implementations, which are LR(1) rather than LR(0). It also already has to reduce after the first symbol, whereas LR only needs to make this decision at the end of the rule. However, this drawback is more than compensated by the fact that it needs neither states nor tables, only the initial graph which can be dynamically updated. This allows dynamic rule changes. In addition, conflicts are not fatal because – like GLR – my algorithm is exhaustive. I designed my implementation to keep its efficiency even in the presence of conflicts. In particular, it never executes the same *match* step at the same text position more than once.

2.2 Implementation

In this section, I first give an overview of the technologies chosen for my implementation. Then, I describe its licensing and availability. Finally, I document the tweaks to my basic algorithm made in my implementation to enhance its efficiency and functionality.

2.2.1 Technologies and API

My implementation is written in C++ using the **Qt** 4 (Qt Company [74]) toolkit. I used the g++ (Free Software Foundation [29]) compiler. The **application programming interface** (**API**) is documented in Kofler [50]. It consists of C++ classes using Qt implicitly-shared data structures, i.e., object references with automatic reference counting that combine the convenient semantics of copies with the efficiency of shared pointers.

I also implemented Java bindings using the Qt Jambi (Voutilainen et al. [96]) binding generator to allow its usage in Java programs, such as Concise (Schodl et al. [87], Domes [22]), the main client application of DynGenPar, a GUI for semantic graph manipulation which will be described in Chapter 3. The Java API is very similar to the C++ API, but Qt data structures are converted to native Java data structures. In particular, the references are not implicitly shared as in C++ with Qt, but follow the standard Java reference model. Copies have to be requested explicitly using the `clone()` method, as for any other Java object. Other minor differences between the C++ and Java APIs are due to technical limitations of Qt Jambi. They are detailed in Kofler [50].

I picked version 4 of Qt because it was the latest available when development started. In addition, as of 2015, Qt Jambi still does not support Qt 5.

2.2.2 Licensing and Availability

The DynGenPar implementation is licensed under the GNU General Public License (Free Software Foundation [30, 31]), version 2 or later. In addition, a bundled version of DynGenPar is distributed as part of the Concise (Schodl et al. [87], Domes [22]) framework under the same terms as Concise.

The source code is available for free download at Kofler [47].

2.2.3 Implementation Considerations

This section documents some tweaks I made to the basic algorithm from Section 2.1 to improve efficiency and provide additional desired features. I describe the modifications required to support predictive parsing, efficient exhaustive parsing, peculiarities of natural language, arbitrary rule labels, custom parse actions and next token constraints. Next, I briefly introduce my flexible approach to lexing.

2.2.3.1 Predictive Parsing

The most intuitive approach to implement the above algorithm would be to use straight recursion with implicit parse stacks and backtracking. However, that approach does not allow incremental operation, and it does not allow discarding short matches (i.e., prefixes of the input which already match the start symbol) until the very end. Therefore, I replaced the backtracking by explicit parse stacks, with token shift operations driving the parse process: Each time a token has to be shifted, the current stack is saved and processing stops there. Once the token is actually shifted, all the pending stacks are processed, with the shift executed. If there is no valid match, the parse stack is discarded, otherwise it is updated. The implementation also remembers complete matches (where the entire start symbol S was matched) and returns them if the end of input was reached, otherwise it discards them when the next token is shifted. This method allows for incremental processing of input and easy pinpointing of error locations. It also allows changing the grammar rules for a specific range of text only.

The possible options for the next token and the nonterminal generating it can be predicted. This is implemented in a straightforward way by inspecting the parse stacks for the next pending match, which yields the next highest-level symbol, and if that symbol is a nonterminal, performing a top-down expansion (ignoring left recursion) on that symbol to obtain the possible initial tokens for that symbol, along with the nonterminal directly producing them. Once a token is selected, parsing can be continued directly from where it was left off using the incremental parsing technique described in the previous paragraph.

2.2.3.2 Efficient Exhaustive Parsing

In order to achieve efficiency in the presence of ambiguities, the parse stacks are organized in a **directed acyclic graph** (**DAG**) structure similar to the GLR algorithm's graph-structured stacks. (Tomita [91], Tomita & Ng [92]) In particular, a *match* operation can have multiple parents, and my algorithm produces a unified stack entry for identical match operations at the same position, with all the parents grouped together. This prevents having to repeat the match more than once. Only once the match is completed, the stacks are separated again.

Parse trees are represented as packed forests. Top-down sharing is explicit: Any node in a parse tree can have multiple alternative subtrees, allowing to duplicate only the local areas where there are ambiguities and share the rest. This representation is created by explicit unification steps. This sharing also ensures that the subsequent *reduce* operations will be executed only once on the shared parse DAG, not once per alternative. Bottom-up sharing, i.e., multiple alternatives having common subtrees, is handled implicitly through the use of reference-counted implicitly shared data structures, and through the graph-structured stacks ensuring that the structures are parsed only once and that the same structures are referenced everywhere.

2.2.3.3 Rule Labels

My implementation allows labeling rules with arbitrary data. The labels are reproduced in the parse trees. This feature is essential in many applications to efficiently identify the rule which was used to derive the relevant portion of the parse tree.

2.2.3.4 Custom Parse Actions

The algorithm as described in Section 2.1 generates only a plain parse tree and cannot execute any other actions according to the grammar rules. But in order to efficiently support things such as mathematical definitions, I need to be able to automatically trigger the addition of a new grammar rule (which can be done very efficiently by updating the initial graph) by the encountering of the definition. Therefore, the implementation makes it possible to attach an action to a rule, which will be executed when the rule is matched. This is implemented by calling the action at the end of a *matchRemaining* step, when the full rule has been matched.

2.2.3.5 Token Sources

The implementation can be interfaced with several different types of token sources, e.g., a Flex (Flex Project [26]) lexer, a custom lexer, a buffer of pre-lexed tokens, a dummy lexer returning each character individually etc. The token source may or may not attach data to the tokens, e.g., a lexer will want to attach the value of the integer to `INTEGER` tokens.

The token source can also return a whole parse tree instead of the usual leaf node. That parse tree will be attached in place of the leaf. This feature makes hierarchical parsing possible: Using this approach, the token source can run another instance of the parser (DynGenPar is fully reentrant) or a different parser (e.g., a formula parser) on a token and return the resulting parse tree.

2.2.3.6 Natural Language

Natural language, even the subset used for mathematics, poses some additional challenges to my implementation. There are two ways in which natural language is not context free: *attributes* (which have to agree, e.g., for declination or conjugation) and other context sensitivities best represented by **parallel multiple context-free grammars** (**PMCFGs**) (Seki et al. [88]).

Agreement issues are the most obvious context sensitivity in natural languages. However, they are easily addressed: One can allow each nonterminal to have *attributes* (e.g., the grammatical number, i.e., singular or plural), which can be *inherent* to the grammatical category (e.g., the number of a noun phrase) or variable *parameters* (e.g., the number for a verb). Those attributes must *agree*, which in practice means that each attribute must be inherent for exactly one category and that the parameters inherit the value of the inherent attribute. While this does not look context-free at first, it can be transformed to a CFG (as long as the attribute sets are finite) by making a copy of a given nonterminal for each value of each parameter and by making a copy of a given production for each value of each inherent attribute used in the rule. This transformation can be done automatically, e.g., the GF (Grammatical Framework) compiler does this for grammars written in the GF programming language.

A less obvious, but more difficult problem is given by split categories, e.g., verb forms with an auxiliary and a participle, which grammatically belong together, but are separated in the text. The best solution in that case is to generalize the concept of CFGs to PMCFGs (Seki et al. [88]), which allow nonterminals to have multiple dimensions. Rules in a PMCFG are described by functions which can use the same argument more than once, in particular also multiple elements of a multi-dimensional category. PMCFGs are more expressive than CFGs, which implies that they cannot be transformed to CFGs. They can, however, be parsed by context-free approximation with additional constraints. My approach to handling PMCFGs is based on this idea. However, it does not use the naive and inefficient approach of first parsing the context-free approximation and then filtering the result, but it enforces the constraints directly during parsing, leading to maximum efficiency and avoiding the need for subsequent filtering. This is achieved by keeping track of the constraints that apply, and immediately expanding rules in a top-down fashion (during the *match* step) if a constraint forces the application of a specific rule. The produced parse trees are CFG parse trees which are transformed to PMCFG syntax trees by a subsequent unification algorithm, but the parsing algorithm ensures that only CFG parse trees which can be successfully unified are produced, saving time both during parsing and during unification. This unification process uses DynGenPar's feature to attach, to CFG rules, arbitrary rule labels which will be reproduced in the parse tree:

The automatically generated label of the CFG rule is an object containing a pointer to the PMCFG rule and all other information needed for the unification.

2.2.3.7 Next Token Constraints

My implementation also makes it possible to attach constraints on the token following a rule, i.e., that said token must or must not match a given context-free symbol, to that rule. I call such constraints *next token constraints*. This feature can be used to implement scannerless parsing patterns, in particular, maximally-matched character sequences, but also to restrict the words following e.g., “a” or “an” in word-oriented grammars. I implement this by collecting the next token constraints as rules are reduced or expanded and attaching them to the parse stacks used for predictive parsing. Each time a token is shifted, before processing the pending stacks, the implementation checks whether the shifted token fulfills the pending constraints and rejects the stacks whose constraints aren’t satisfied.

Chapter 3

DynGenPar and Concise

Concise (Schodl et al. [87], Domes [22]) is a framework and GUI for viewing and manipulating, both graphically and programmatically, semantic graphs. A **semantic graph** is a labeled directed multigraph representing semantic information.

Concise is the main application in the FMathL project (Neumaier [70]). DynGenPar, as the parser of choice in the FMathL project, plays a central role in Concise. DynGenPar enables Concise to parse input using grammars specified in external text files and loaded at runtime. Concise uses this feature internally for its own programming languages. This feature is also the way to teach Concise a new formal language or a new controlled subset of natural language. Such applications are the subject of the next chapters. The same grammars that allow parsing input using DynGenPar also enable producing output as text files or graphical text views. The prediction functionality of DynGenPar enables editable file views with autocompletion.

Concise is written in Java, whereas DynGenPar is written in C++. Therefore, DynGenPar is used through its Java bindings (see Section 2.2.1) in Concise. Those bindings enable using the C++ code of DynGenPar as if it were implemented in Java.

After giving a short introduction to Concise, this chapter discusses the interaction between DynGenPar and Concise and the functionality it enables in Concise.

3.1 Concise

Concise is a framework for the manipulation of **semantic graphs**, i.e., labeled directed multigraphs with a semantic interpretation. It is written in Java. The core of Concise is called the **semantic memory**, a large semantic graph of which all the semantic graphs processed by Concise are subgraphs. It consists of **objects**, the nodes in the graph, and **sems**, the edges.

Concepts are stored in the semantic memory as sems. Every sem relates three objects: a **handle**, a **field** and an **entry**. It is denoted `handle.field = entry`. In the graph interpretation, the sem is an edge from the handle to the entry, labeled with the field. The mapping from handle and field to the entry is unique, i.e., the dot operator . is

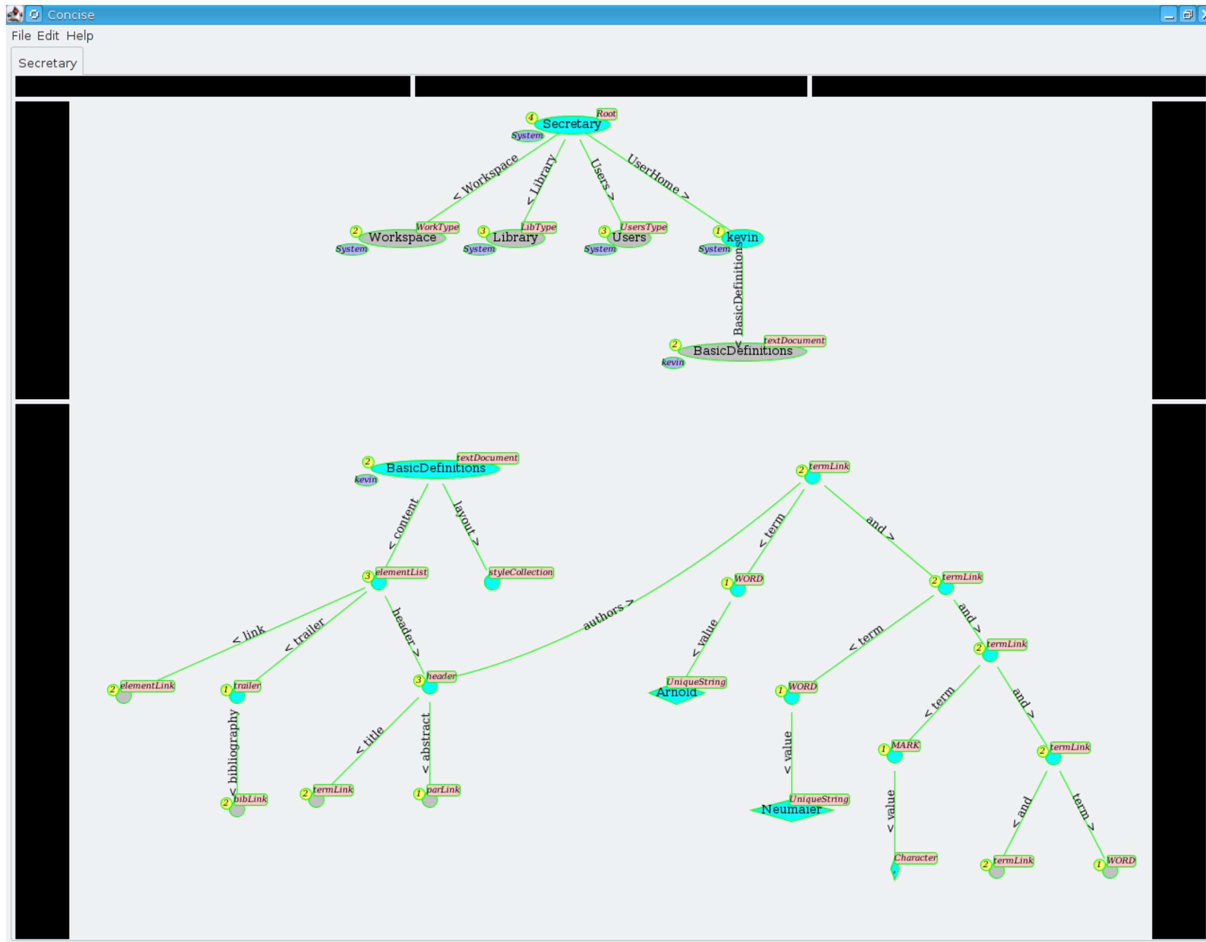


Figure 3.1: Screenshot of Concise showing two graph views

a partial function. On the other hand, there may be multiple fields relating the same handle to the same entry, which makes the semantic graph a multigraph. The sems are also naturally directed, because $x.f = y$ is a totally different sem from $y.f = x$.

An alternative interpretation of sems relates them to triplets (`handle`, `field`, `entry`), as found, e.g., in the **RDF** (**Resource Description Framework**) (W3C [98]) standard. However, unlike RDF, Concise does not allow multiple entries for the same (`handle`, `field`) pair, i.e., its . operation is a true partial function, not a multi-valued map. Therefore, Concise frequently relies on linked lists, which are used to represent concepts that would otherwise require one (`handle`, `field`) pair to carry multiple entries. Unlike multi-valued entries, linked lists are naturally ordered. They are also friendly to parsing, because they are exactly the trees produced for lists by a context-free parser.

In addition, Concise also supports **external objects**. Those are objects representing a value that is stored outside of the semantic memory. Concise currently supports the following types for external objects (**external types**): `Abstract2DShape`, `Array`, `Boolean`, `Character`, `Color`, `Dimension`, `Double`, `DoubleInterval`, `EscapedCharacter`, `EscapedString`, `EscapedUniqueString`, `File`, `Font`, `Integer`, `IntegerInterval`, `IntegerName`, `Matrix`, `Name`, `Picture`, `Point`, `Rectangle`, `String`, `TextLine`, `Timer`,

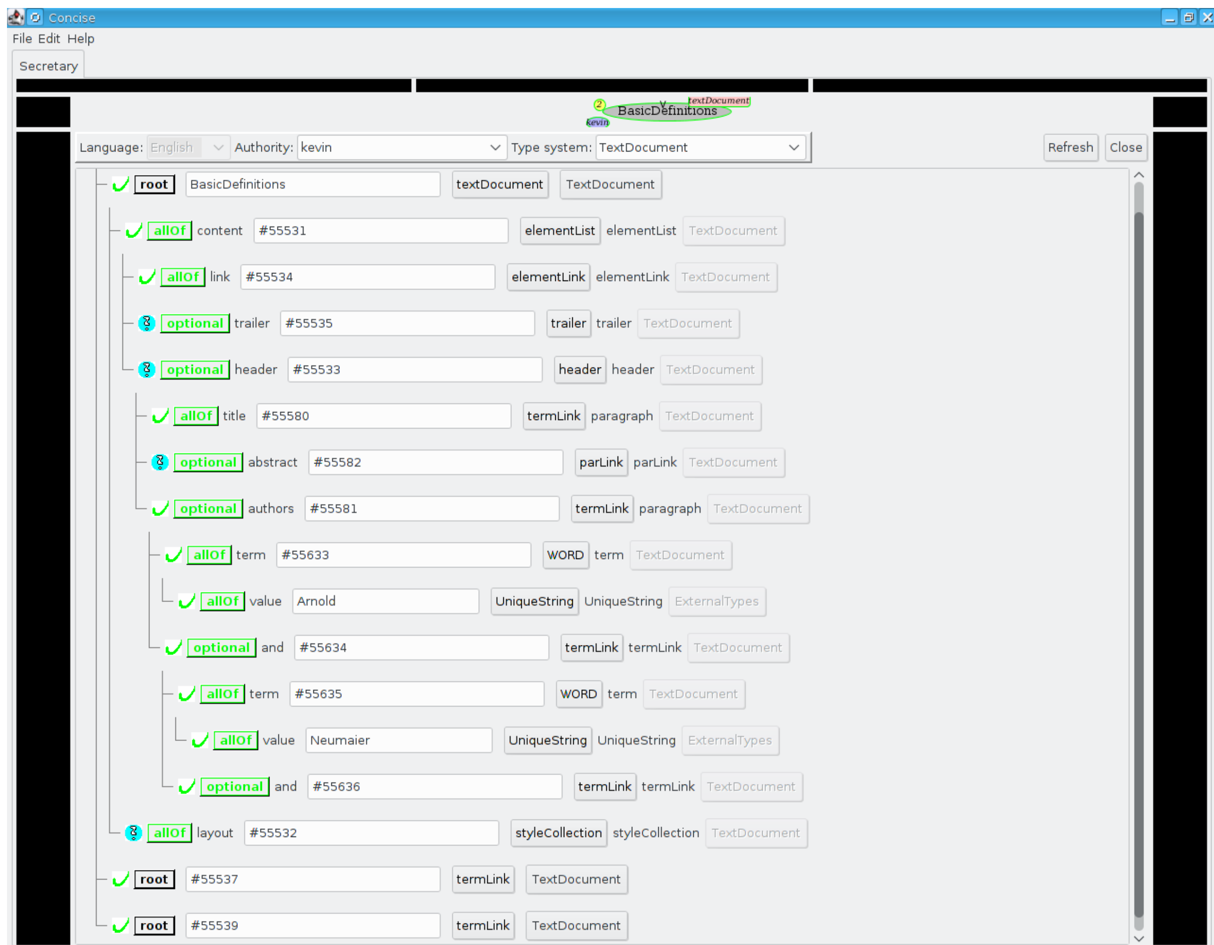


Figure 3.2: Screenshot of Concise showing a record view

`UniqueString`, and `Vector`. Each external type is implemented as a Java class inside Concise. To add support for an additional external type, one would simply implement one more such Java class.

To users, Concise presents itself as an interactive GUI for graph editing, which operates on the semantic memory as described above. The Concise GUI offers several editable views of semantic content. The primary view is the **graph view** (Figure 3.1), in which the semantic memory is drawn as a graph, with the nodes (objects) represented as ellipses and the edges (sems) as lines. As the entire semantic memory is a huge graph which is impractical to draw as a whole, the graph view allows interactively expanding or collapsing subgraphs, and it is also possible to bring up a graph view restricted to a subgraph. An additional view is the **record view** (Figure 3.2), which represents subtrees of the graph as records. It displays fields and their corresponding entries as an expandable tree: Each entry can be expanded to show, in turn, its own fields and corresponding entries, and this can be repeated recursively. (In GUI programming, this is called the **tree view** pattern.) Finally, there is the **text view** (Figure 3.3), a textual representation of records which will be described in detail in Section 3.3.4. Non-tree structures (where multiple edges point to the same entry) are displayed in the record view as references to a common subrecord;

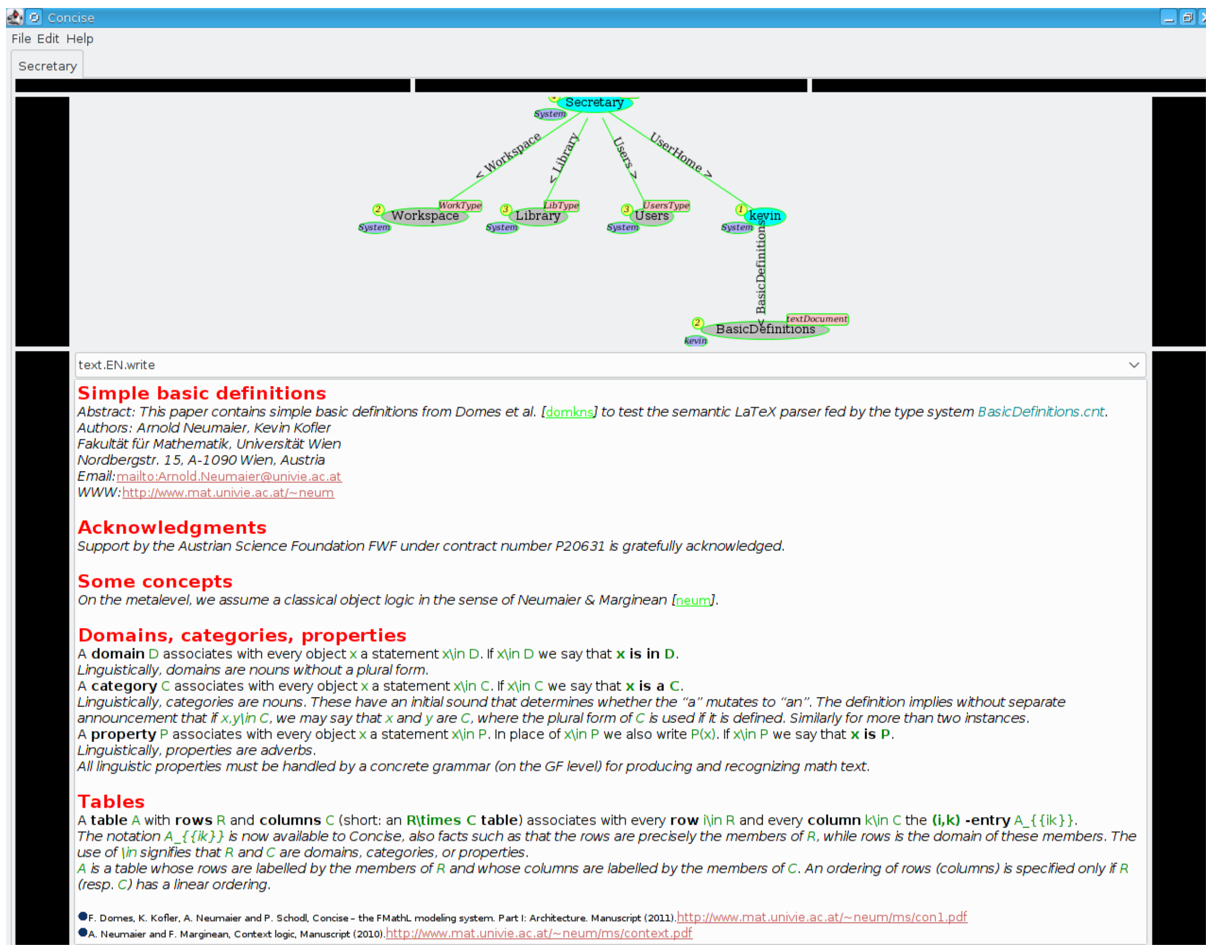


Figure 3.3: Screenshot of Concise showing a text view

in the text view, the text for the common subgraph is simply copied.

Concise views can also operate on external objects. Most notably, there are **parsable file views**, operating on files. Parsable file views are embedded text editor views for text files that can be parsed with DynGenPar, i.e., that are written in a formal language for which a DynGenPar grammar is available. They make use of the prediction functionality of DynGenPar to offer autocompletion. They will be described in detail in Section 3.3.5.

Concise can also execute programs operating on the content of the semantic memory, and supports importing information from and exporting it to various types of textual files: **view sheets**, **record sheets**, **type sheets**, and **code sheets**. Those operations can be done both interactively through the GUI and automatically, using the Concise core directly.

The first type of sheets that Concise can import and export represent arbitrary semantic memory contents. There are two formats for such sheets: **View sheets** (`.cnv`) are the internal serialization format of Concise. While in a text format, they are not easily readable by humans, because they are a raw dump of the graph edges, representing the objects by their numeric IDs. **Record sheets** (`.cnr`) should instead be preferred for

readability. They describe records in a human-readable, tree-structured text form. As in the interactive record view, non-tree structures such as shared subgraphs and cycles are represented through the use of labels. The default `.cnr` representation is a compact custom text format; an alternative **XML** (**eXtensible Markup Language**) representation exists, but is significantly (about 2 to 4 times) larger.

Another important kind of Concise sheets is the **type sheet** (`.cnt`). It describes a type system, i.e., the specification of several types, in a human-readable text form. The specification of a type consists of required fields, optional fields, etc., e.g.:

```
header:
allOf> title=paragraph
optional> abstract=parLink
          keywords=paragraph
          authors=paragraph
          date=paragraph
          copyright=paragraph
          comment=paragraph
```

This specification says that a header always contains a field `title` of type `paragraph`, and that it may or may not contain fields `abstract` of type `parLink`, `keywords` of type `paragraph`, etc. (but if the fields are present, they must have the required type). The details of the FMathL type system implemented in Concise can be found in Schodl & Neumaier [86]. Type sheets can also carry grammatical information, specifying how to parse or linearize the types, turning them into context-free grammars. This will be described in more detail in the next section and in Section 4.1.

The remaining types of Concise sheets are the **code sheet** (`.cnc`), which represents programs in the semantic memory in a language documented in Section 4.2.2, and the automatically generated **package sheet** (`.cnp`), which documents a package of external functions which can be used in programs or for text formatting and is automatically exported from the help data included in the Java classes implementing those functions. Code sheets cannot be exported, package sheets cannot be imported.

3.2 Embedding of DynGenPar into Concise

In order to integrate DynGenPar into the Java-based Concise framework, as already mentioned in Section 2.2.1, the **Qt Jambi** (Voutilainen et al. [96]) binding generator was used to produce Java bindings for DynGenPar. A short **XML** (**eXtensible Markup Language**) file was written containing rules for the binding generator, Qt Jambi automatically takes care of generating the complex **JNI** (**Java Native Interface**) code for the bindings. That generated code wraps C++ classes as Java classes, even allowing classes written in Java to extend the wrapped C++ classes and implement their virtual methods. This allows C++ code to call back into a method implemented in Java without even having to know that it is in fact calling Java code. It also automatically handles conversion between Qt's template data structures (e.g., `QList`, `QHash`, etc.) and the Java generic equivalents. However, some code generation bugs were encountered with nested

containers which do not appear in the code of Qt itself (the code the binding generator was tested with); fixes for those bugs were submitted to the Qt Jambi project. Another feature of Qt Jambi DynGenPar makes use of is the mapping of untyped data between Qt's `QVariant` and Java's `java.lang.Object`.

As described in the previous section, the FMathL type system (Schodl & Neumaier [86]) is represented in the form of text files called **type sheets**. Those type sheets not only represent a pure type hierarchy, but may also carry grammatical annotations called **usages**, which allow the type system to double as a grammar. A usage is attached to a type definition and can serve one of four purposes:

- **input usage**: a grammar rule describing how to parse the type,
- **output usage**: a grammar rule describing how to linearize the type for text output. A special case of output usage is the **text view usage** for use in the text views (see the previous section), which can contain calls to text formatting functions.
- **token source usage**: a rule listing the tokens that should be produced when linearizing the record (of a text-oriented type) to a stream of tokens for the parser, in order to obtain a record of a semantic-oriented type,
- **help usage**: a rule giving documentation for the type, in the form of text which can contain calls to text formatting functions.

E.g., the example type definition from the previous section is actually given in the `TextDocument.cnt` type sheet with the following token source and text view usages:

```
header:
allOf> title=paragraph
optional> abstract=parLink
          keywords=paragraph
          authors=paragraph
          date=paragraph
          copyright=paragraph
          comment=paragraph
#ts> #title#abstract#keywords#authors#date#copyright#comment
#w> #!setFontSize&(19&) #!setColor&(255&, 0&, 0&) #!setBold&(&) &&
    &[#title&n&] #!resetAttribs&(&) &&
    #!toggleItalic&(&) &&
    &[Abstract:#abstract&n&] &&
    &[Keywords:#keywords&n&] &&
    &[Authors:#authors&n&] &&
    &[at #date&n&] &&
    &[Copyright:#copyright&n&] &&
    &[Comment:#comment&n&] &&
    #!toggleItalic&(&)
```

The token source usage `#ts>` specifies that, when using this record as input for parsing, the parser should simply be passed the token stream for each of the fields in sequence (but in practice, a `header` record will not actually be sent to the parser, see Section 6.2 for how the token source usages in `TextDocument.cnt` are used). The text view usage `#w>`

instructs the text view to print the title in large, red, bold letters and the optional fields, if present, in italics, preceded by the name of the field (or by "at" for the date). The complete usage syntax can be found in Section 4.1. The example shows the usage syntax for field references (e.g., `#title`), function calls (e.g., `#!setColor&(255&, 0&, 0&)`), optional parts (`&[...&]`) and line continuation (`&&`).

But the most important type of usage for parsing is the input usage, because a type sheet with input usages is equivalent to a grammar in **Backus-Naur form** (**BNF**), augmented with a specification of the record representation of the syntax tree. I implemented a Java class `concise.parser.Grammar` inside Concise around the DynGenPar Java bindings. It takes as input the internal Concise representation (`concise.types.system.TypeSystem`) of a type sheet annotated with input usages and produces a grammar in DynGenPar's internal representation (`concise.DynGenPar.Cfg`), which parses a token stream and outputs valid records for the given type system. In addition to standard context-free grammars, the context-sensitive constraints supported by DynGenPar can also be represented this way: next token constraints (Section 2.2.3.7) and parallel multiple context-free grammar (PMCFG) (Seki et al. [88]) constraints (Section 2.2.3.6). My code currently supports two possible token sources:

- DynGenPar's built-in `TextByteTokenSource`. This simply produces one token for every character in the input (except that it strips the carriage return CR out of CR LF – carriage return + line feed – line endings), allowing for scannerless parsing of arbitrary text, and
- a special `ConciseTokenSource`. This linearizes a Concise type into a stream of tokens which are sent to the parser, allowing to use DynGenPar to convert a record containing unparsed text into a record representing the same text semantically. This token source is part of the `TextDocument` toolchain described in Section 6.2.

However, most of the conversion process is independent of the underlying token source.

In order to produce Concise records as output, the `concise.parser.Grammar` class attaches to every rule labels that are actually Java objects building a Concise record in its Java representation. This is possible because DynGenPar accepts a `QVariant` as a label, and Qt Jambi allows storing an arbitrary Java object reference inside a `QVariant`. After parsing, Concise calls the `concise.parser.Grammar.parseTreeToRecord` method that traverses the parse tree produced by DynGenPar and calls all the labels to build the resulting Concise record.

The rule labels can be of one of the following Java classes:

- `NewRecordBuilder`: Creates a new record of a given type. Every item index in the rule can be either ignored or mapped to one of the following:
  - a field in the new record, or
  - a subrule, processed using the `IncrementalRecordBuilder` below (used where the original rule contains **extended Backus-Naur form** (**EBNF**) and must be decomposed to multiple plain BNF rules), or
  - a "leaf" field, where the matched item in the parse tree is assumed to be a token and the field in the new record is set to its attached value.

- `IncrementalRecordBuilder`: Like `NewRecordBuilder`, but instead of creating a new record, the fields are added to the record created by the parent `NewRecordBuilder`.
- `RecordCaster`: Converts a record to another type. This is used in `union` rules, to upcast a record to its supertype.
- `TokenCollector`: Recursively collects all the matched tokens in a list. If a nonterminal is matched, the token collector for that nonterminal is invoked to collect its own tokens and the whole list is inserted in the place of the nonterminal. This label is used for lexical rules, which do not produce a record.
- `TokenEmitter`: A special case of token collector. It ignores the matched tokens and "collects" a set of replacement tokens instead. It is used to implement substitution rules, which convert an escaped input to unescaped output.
- `ExternalCaster`: Recursively collects the matched tokens like a `TokenCollector`, builds a string from them, converts it to the given external type (see Section 3.1), and finally casts it to a given final type like a `RecordCaster`. This is the top-level token collector and the interface between token collection and record building. The result is an external object and thus a leaf record.
- `ExternalNameBuilder`: Creates an external object of type `Name` with the given name. It is used for rules dynamically added through definitions, which will be described in Section 6.3.

Concise can import type sheets at runtime and, using the above process, automatically convert them to grammar rules suitable for DynGenPar and then parse documents using the converted grammar. Therefore, user-written rules can be fully read into the parser at runtime, rather than hardcoding them as C++ or Java code or compiling them to some other precompiled format (such as the PGF format of the Grammatical Framework (GF), which DynGenPar also supports, see Section 5.2). Concise type sheets represent a user-friendly mechanism for specifying rules that can be easily converted to my internal representation. This feature is thus an ideal showcase for the dynamic properties of my algorithm.

3.3 Concise Features based on DynGenPar

The Concise GUI fully integrates my DynGenPar parser into its application workflow. Several features of Concise make use of DynGenPar. This section describes where and how my parser is used in Concise.

3.3.1 Type Sheets

Type sheets (as described in the previous section) not only serve as the representation of grammars in Concise, they are also themselves parsed using DynGenPar, using a grammar which will be described in Section 4.1.

Concise originally had a rudimentary parser for type sheets hardcoded in Java. That basic parser is still included for compatibility reasons and as a fallback. However, it supports only a subset of the type sheet syntax, and thus cannot parse most of the current type sheets.

Therefore, Concise now parses type sheets using a grammar that is itself a type sheet, `TypeSheets.cnt`. Because of the bootstrapping issues this would cause, I manually converted the grammar to a C++ program, `cnttoxml`, using DynGenPar's native C++ API. That program converts the type sheets to record sheets (in an XML representation, which is then converted with another C++ program to a `.cnr` record sheet).

Thus, Concise loads type sheets by:

1. converting them to record sheets using these external programs,
2. importing the resulting record sheets, and
3. converting those records of `TypeSheets` type to native type systems.

3.3.2 Code Sheets

Code sheets are the language in which programs for Concise are represented. These are also read using a type sheet, `CodeSheets.cnt` (see Section 4.2.2), and DynGenPar. In this case, no external bootstrap parser is needed. The parsing is based directly on the type sheet and on the `concise.parser.Grammar` class described in Section 3.2. It is then further translated into an internal representation and executed. See Section 4.2 for details.

3.3.3 Record Transformations

It is often necessary in Concise to convert records from one representation (in one type system) to another (in a different type system). Typically, the source type system is used for parsing, the destination type system is the internal, more semantic representation. But there are also other transformations, e.g., in the opposite direction, or simplifications within the same type system.

For this purpose, Concise supports **record transformations**, which convert a record from one type system to another. The two given type systems can also be the same, as in the case of a simplifier. A record transformation can be run from the GUI menus or internally by Concise. **Record transformation sheets** are the source language in which record transformations are represented. These are also read using a type sheet, `RecordTransformation.cnt` (see Section 4.3), and DynGenPar.

3.3.4 Text Views

Text views are Concise views displaying a textual representation of records. Figure 3.4 shows two text views. They work using the **usages**, i.e., the grammatical annotations in

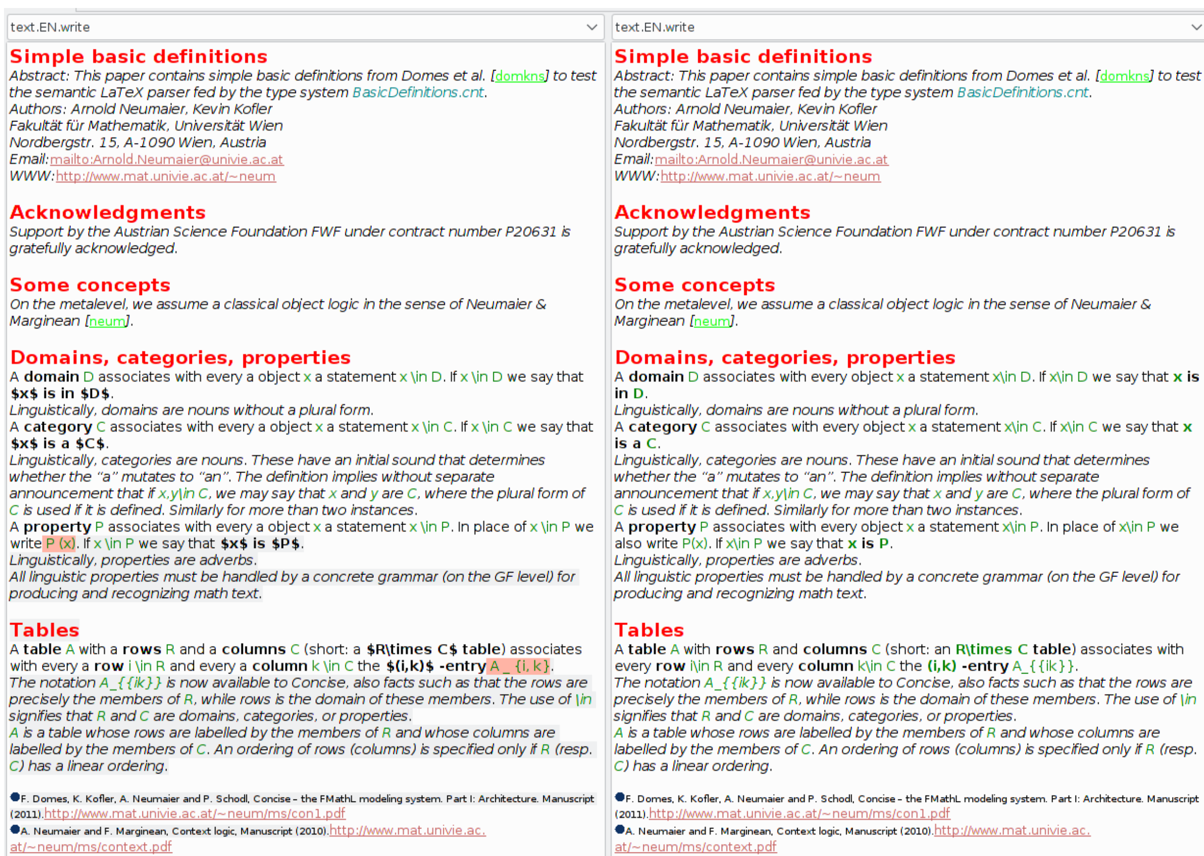


Figure 3.4: Two text views, displaying the parsed (using the `BasicDefinitions` type sheet from Section 6.3 and the `LaTeXFormulas` type sheet from Section 6.5) and unparsed representation of a LaTeX document side by side

type sheets which are also used for parsing, described in Section 3.2. However, unlike the parser, which uses input usages, the text views use output usages. They produce text output for the records by doing **linearization**, i.e., the opposite of parsing.

The text view operates by recursing on the record. For every object it encounters, it checks the type of that object for a matching output usage. It displays as output the text of the usage, expanding any placeholders it contains. The most common placeholder is `#`*`field`*, which tells the text view to insert the text output for the given *field* at the position of the placeholder. This leads to the aforementioned recursive operation.

An additional feature that text views support is formatting. This is implemented through functions that the usages can call. The following functions are currently available (quoting the file `ExternalFunctions.cnp` automatically generated by Concise):

- `ifNotEmpty(testarg, text)` – If the first argument is not empty, the second will be written to the output.
- `resetAttribs` – Reset all style attributes to their default values.
- `resetColor` – Reset the color attribute to its default value.
- `resetFontSize` – Reset the font size attribute to its default value.

- `setBold` – Set the bold attribute.
- `setColor(red, green, blue)` – Set the color attribute to the given rgb value.
- `setFontSize(size)` – Set the font size attribute to the given value.
- `setItalic` – Set the italic attribute.
- `setNonBold` – Clear the bold attribute.
- `setNonItalic` – Clear the italic attribute.
- `toggleBold` – Toggle the bold attribute.
- `toggleItalic` – Toggle the italic attribute.

While in some cases, different usages are required for linearization than for parsing, in many cases the same usages can be shared for both input and output. This implies that the grammar written for DynGenPar can also, with minor changes, be used to visualize the record produced by the parser in a text view. While the text views themselves do not directly use DynGenPar, this property makes them closely related to DynGenPar grammars.

Concise text views were primarily implemented by Ferenc Domes, the main author of Concise. However, I made some improvements to the text views which make it easier to write usages for text views and allow reusing parser rules more often. In particular:

- I added support for optional parts, i.e., `&[...&]` items. They are processed as follows: If the object has all the fields referenced inside the optional part, it is expanded, otherwise, it is skipped. This avoids the need for many `ifNotEmpty` calls and often allows using the same rule as for parsing. In addition, unlike with `ifNotEmpty`, if the optional part is skipped, so are any calls to formatting functions it contains.
- I implemented the `toggleBold` and `toggleItalic` functions. Especially `toggleItalic` is very useful because it is customary to display italics within italics as straight letters. These functions also avoid calling `setNonItalic`, which would reset the italic flag even within a section supposed to be entirely in italics, and likewise for `setNonBold`.
- I implemented a basic **unlexer**, which takes care of inserting the correct whitespace between words and punctuation characters, and which automatically capitalizes the first word of a sentence.
- I fixed several bugs I encountered while testing the text views on my grammars.
- I added a feature that allows disambiguating ambiguous records produced by the parser. See Section 6.5 for details.

A planned future extension is to allow editable text views. DynGenPar can, through its incremental processing of input, validate user input in real time, and produce a valid record as soon as the input is complete. In addition, its prediction functionalities can be used to provide intelligent autocompletion. Such autocompletion is already implemented in Concise in the parsable file views, described in the next section. It is planned to also adapt that functionality to the text views.

3.3.5 Parsable File Views

```
474 #t> #CRange&[,##blanks#next&]
475
476
477 ! Literal characters
478 !
479 Char:
480 union> String
481
482 nothingElse   UnionSpec:
483 oneOf>        allOf>names=NameLink
484 only>         #t> union > ##blanks #names            nk
485 optional>
486 someOf>
487 someOfTyp     :Char
488 template>     :xt=CharLink
489 union>        :next&]
490
491
492 ! Output characters
493 !
494 OutChar:
495 union> String
496 #t> ##outchar
497
498
499 ! Each link of a linked list of output characters is an OutCharLink
500 !
501 OutCharLink:
502 allOf> OutChar=OutChar
```

Figure 3.5: A type sheet view (a parsable file view operating on the grammar for type sheets, see Section 4.1) showing the autocompletion popup

Parsable file views are embedded text editor views for text files that can be parsed with DynGenPar, i.e., that are written in a formal language for which a DynGenPar grammar is available. They make use of the prediction functionality of DynGenPar to offer autocompletion. The user interface was implemented by Ferenc Domes, the interface to DynGenPar by me. Figure 3.5 shows a parsable file view for a type sheet.

The parsable file views allow validating the input and producing a valid record at the press of a button. The user experience is similar to how modern integrated development environments (IDEs) for programming languages work. However, in Concise, everything is done within the Concise application using DynGenPar.

The parsable file views also support intelligent autocompletion. The autocompletion popups list valid continuations for the input. Since the text grammars in Concise are scannerless, instead of tokens, the prediction suggests **literals**, i.e., sequences of characters that appear together in a rule. E.g., if the rule contains a keyword, the whole keyword is returned, not just one character (token) from it. (Note that to make longer completion suggestions than such a literal, one would in general have to introduce placeholders as in DE SOUZA AMORIM et al. [19].) The type sheet defining the grammar can also define **completion filters**, which replace verbose output by a given summary. E.g., instead of offering every single letter in the alphabet, a filter can make the popup display the string "*any letter*".

When the completion popup is open, the incremental processing of input of DynGenPar is used to update the popup in real time. The new suggestions after entering a character are produced virtually instantly. E.g., if the grammar contains the keywords `when` and `while`, and if both are applicable in the current context, the completion popup after `wh` will initially offer both options. After entering the additional letter `i`, the predictions are updated using the incremental parsing of DynGenPar, and only the keyword `while` is offered. However, the predictions are really recomputed, not just filtered. This means that new suggestions can come up, which is the case if a complete literal has been typed in.

A prior proof of concept of such predictive input is included with DynGenPar as a standalone application, operating on PGF grammars compiled by the Grammatical Framework (GF), see Section 5.4. The parsable file views improve on that concept by allowing freeform text entry, whereas the proof of concept allowed only selection of choices from the completion list.

It is planned for the future to also integrate this functionality into the text views, so that records can also be edited directly in the semantic memory, with the same convenient features.

Chapter 4

Applications of DynGenPar to Formal Languages

As we have seen in Chapter 2, DynGenPar is a context-free parser with several unique additional features. Though some of those features are targeted primarily at natural language processing, DynGenPar can of course also parse fully formalized languages. This chapter details the formal languages for which DynGenPar has been successfully applied. Applications of DynGenPar closer to natural language will be the subject of Chapter 6.

As described in the previous chapter, DynGenPar is the parser of choice for Concise, the engine behind the FMathL project. For this project, a suitable parser for natural language processing research was needed. I wrote DynGenPar to fit those requirements and integrated it into Concise. It was subsequently decided to use DynGenPar for all tasks requiring a context-free parser in Concise. This had several advantages. For one, the infrastructure around DynGenPar in Concise could be reused. In particular, the grammars can be described as Concise type sheets (see Sections 3.2 and 4.1). Secondly, this avoided having to require in Concise an additional parser beyond our control. And finally, some features of DynGenPar, e.g., the scannerless parsing support, are also very useful for formal grammars. One of the grammars in this chapter, a grammar for optimization problems, also makes use of the DynGenPar feature that allows to dynamically add rules to the grammar. The feature is used to support a `\newcommand` command as in LaTeX.

This chapter enumerates and documents the formal languages to which DynGenPar was applied. First, grammars for type sheets, code sheets, and record transformation sheets, which are core parts of Concise, are described. Then, a grammar for chemical process modeling using the Concise framework is detailed. Next, an extensible grammar for optimization problems demonstrating the dynamic grammar extensibility of DynGenPar is documented. Finally, a grammar for a subset of the AMPL (Fourer et al. [27], AMPL Optimization inc. [2]) modeling language for optimization problems, extended to allow intervals wherever AMPL normally expects a number, is detailed.

4.1 A Grammar for Type Sheets

The most important application of DynGenPar in the core of Concise is the grammar for type sheets. As explained in Section 3.2, a type sheet is the textual representation of a Concise type system. The type sheet may or may not include grammar information. The most interesting type sheets for DynGenPar are of course those which do encode a grammar.

The grammar for type sheets is itself given in the form of a type sheet, TypeSheets.cnt, reproduced in Appendix A. The following paragraphs are based on the introduction of that type sheet (Neumaier & Kofler, Appendix A).

The information required for parsing and views is specified by productions. There are two kinds of such productions: **Literal productions** are global to the entire type sheet and follow a line consisting only of a colon : (and optional comments). They are used for recurring, purely lexical patterns which do not appear in the final syntax tree. **Categorical productions** are local to a category and follow the type definition of that category. They describe a textual representation for that particular category. (Neumaier & Kofler, Appendix A)

Within productions (only), the exclamation mark !, the hash #, the ampersand &, the newline characters and the blanks are escaped, since they have syntactic meaning in the grammar for the type sheet.

- &! encodes the exclamation mark, and &# encodes the hash #.
- &n encodes a newline. &b encodes a blank, &t encodes a tab, and &c encodes a backspace.
- &0, ..., &9 encode strings consisting of 0, ..., 9 ampersands, each of which is fed to the parser as an individual token.
- Both &0 and && followed by a newline denote the empty string.

Regular expression syntax is encoded as follows:

- &[...&] is optional (appears at most once).
- &(...&) appears once.
- &{...&} appears n times, $n \geq 0$.
- &<...&> appears n times, $n \geq 1$.
- &| separates alternatives in a pattern.

Note that the ampersands before the brackets or | must be present, since otherwise the symbol is treated as a character! (Neumaier & Kofler, Appendix A)

Lexical matching conditions use the following syntax:

- &^ accepts the preceding single-character pattern match if and only if the following pattern does not match the same character (set difference). This consumes both patterns.

- `&E` accepts the preceding pattern match if and only if the following single-character pattern matches the next character ("expect" constraint), otherwise the pattern fails to match entirely (i.e., this fails the entire rule unless there is another `&|` alternative which matches). This consumes the preceding pattern only.
- `&T` accepts the preceding pattern match if and only if the following single-character pattern does not match the next character ("taboo" constraint), otherwise the pattern fails to match entirely (i.e., this fails the entire rule unless there's another `&|` alternative which matches). This consumes the preceding pattern only.
- `&+` maximally extends the preceding pattern of the form `&{`*#char*`&}` or `&<`*#char*`&>`, where *#char* is a single-character pattern.

Single-character patterns may contain any extended Backus-Naur form (EBNF, i.e., regular expression syntax and variable references), but no further `&^`, `&E`, `&T` or `&+`. (Neumaier & Kofler, Appendix A)

Variable references are encoded as follows:

- `#`*fieldName* – recurse to the entry referenced by the field of name *fieldName* and process the productions for the new object. The special word `&this` is not allowed as a *fieldName*.
- `#>`*fieldName* – the name of the entry referenced by the field of name *fieldName*, or the value in case the entry is an external object (see Section 3.1). The *fieldName* `&this` is used for referencing the current object itself rather than one of its fields.
- `#=`*fieldName* – the ID of the entry referenced by the field of name *fieldName* in the current semantic memory. The ID is a negative integer for external objects and a nonnegative integer otherwise. The *fieldName* `&this` is used for referencing the current object itself rather than one of its fields.
- `##`*name* – reference to a literal variable (i.e., to a variable defined by a literal production) of name *name*

Function calls are denoted by `#!`*name*`&(`, followed by a list of patterns separated by the `&,` separator, followed by `&)`. This calls a function of name *name* with arguments separated by `&,`. Which functions can be used depends on the context. For productions targeting text views, the text formatting functions listed in Section 3.3.4 are available. For productions used with the `ConciseTokenSource`, which linearizes a category into a stream of tokens that are sent to DynGenPar, a function named `out` is provided. The `out` function takes one required parameter, an integer representing the ID of the token to be sent to DynGenPar. An optional second integer parameter can be used to attach a value to the token. The `out` function may also be used in productions used for parsing, where it is processed the inverse way. This is done by matching the token given as the first parameter and *inputting* the second parameter from the token's value. In that context, the first parameter (the token to match) must be constant, the second parameter, if given, must be of the `#=`*fieldName* form, and the field *fieldName* of the output record is set to the object whose ID is contained in the token's value. (Neumaier & Kofler, Appendix A)

The special syntax `&=&(`*pattern*`&)` is most useful to define the first parameter of the `out`

function described above. It converts a value *pattern* of one of the types for external data supported by Concise (string, integer, float, etc., see Section 3.1) into the ID representing that value in the semantic memory, a negative integer. E.g., `&=&(NEWCONCEPT&)` is the ID of the string `"NEWCONCEPT"` in the current semantic memory. (Neumaier & Kofler, Appendix A)

The type sheet `TypeSheets.cnt` can of course be automatically imported into Concise and DynGenPar through the process from Section 3.2. The resulting internal representation of the grammar is reproduced in Appendix B. The grammar thus obtained can parse `TypeSheets.cnt` itself. This process can be reiterated and passes bootstrap comparison, i.e., the same result is obtained after each step.

The internal representation is very informative. It not only illustrates the conversion process, but also gives a non-self-referential description of the type sheets grammar. It is a grammar in the standard Backus-Naur form (BNF), with only a few extensions:

- The rules have labels (see Section 2.2.3.3), which are references to Java objects. They are used during the conversion process from the parse tree to a Concise syntax record. Each label object constructs the corresponding portion of the parse tree. The different types of label objects are documented in Section 3.2.

- Some rules also have next token constraints (see Section 2.2.3.7).

An EBNF (extended Backus-Naur form) version of the grammar, produced by manual simplification of the internal representation, is reproduced below. In addition to standard EBNF notation, the grammar below contains a few instances of maximal matches (i.e., greedy matches as used in lexers and scannerless parsers), denoted by a subscript *max*, three instances of character set differences, and one instance of a "taboo" constraint on the next token. These extensions all map to next token constraints in the internal representation.

Tokens: the set of 8-bit characters 0 . . . 255
Start category: *TypeSheet*
Productions:
##hchar → `\x00` . . . `\x09` | `\x0B` . . . `\x0C` | `\x0E` . . . `\xFF`
##char → *##hchar* | `\n`
##letter → `A` . . . `Z` | `a` . . . `z`
##digit → `0` . . . `9`
##blanks → ␣$^{*}_{\max}$
##eol → *##blanks* `\n`$^{+}_{\max}$
##hascii → `\x00` . . . `\x09` | `\x0B` . . . `\x0C` | `\x0E` . . . `\x7F`
##utf8s2 → `\xC0` . . . `\xDF`
##utf8s3 → `\xE0` . . . `\xEF`
##utf8s4 → `\xF0` . . . `\xF7`
##utf8cont → `\x80` . . . `\xBF`
##line → *##hchar*$^{*}_{\max}$
##id → *##letter* (*##letter*|*##digit*)$^{*}_{\max}$
##digits → *##digit*$^{+}_{\max}$
##foreignword → (*##hchar* \ :)+

```
##outchar → ##char \ "
##litchar → ##hascii \ (!|#|&|␣)
          | ##utf8s2 ##utf8cont
          | ##utf8s3 ##utf8cont ##utf8cont
          | ##utf8s4 ##utf8cont ##utf8cont ##utf8cont
          | & ! | & # | & n | & b | & t | & c | & 0 ... & 9
TypeSheet → Header [Targets] [StartCategory] EntryLink [##eol CommentLink] [##eol]
Header → Id ( Id , ##blanks Id ) : : [(##blanks | ##eol ##blanks) ImportLink] ##eol
          CommentLink [TranslationLink]
Id → ##id
PosInt → ##digits
ForeignId → ##foreignword
ImportLink → Id [. Id [- > Id]] [, (##blanks | ##eol ##blanks) ImportLink]
TranslationLink → ##eol [CommentLink] : L A N G : ##blanks Id ##blanks \n
                  IdTranslationLink [TranslationLink]
IdTranslationLink → : : ForeignId : ##blanks Id ##blanks \n [IdTranslationLink]
CommentLink → Comment [CommentLink]
Comment → ##blanks ! ##blanks Line \n
Line → ##line
Targets → ##eol [CommentLink] : T A R G E T > ##blanks TargetLink
TargetLink → Target [##blanks TargetLink]
Target → FieldLink = VarLink (Comment | ##blanks \n)
FieldLink → Id [. FieldLink]
VarLink → # Id [␣ VarLink]
StartCategory → : S T A R T = Id (Comment | ##blanks \n)
EntryLink → (LitDef | CatDef) [EntryLink]
LitDef → ##eol [CommentLink] : (CommentLink | ##blanks \n) LitLink
LitLink → LitProduction [LitLink]
LitProduction → ##blanks # Id > > ␣ # # Id = Substitution (CommentLink | ##blanks \n)
              | ##blanks # Id > > ␣ # # Id = AlternativeLink (CommentLink | \n)
              | ##blanks # Id > > ␣ # # Id = # [ CRangeLink ] (CommentLink | ##blanks \n)
Substitution → CharLink & " [OutCharLink] "
CRange → PosInt [- PosInt]
CRangeLink → CRange [, ##blanks CRangeLink]
Char → ##litchar
CharLink → Char [CharLink]
OutChar → ##outchar
OutCharLink → OutChar [OutCharLink]
Alternative → ElementLink
AlternativeLink → Alternative [& | AlternativeLink]
ElementLink → Element [ElementLink]
Element → CatVar [MatchCase] | LitVar [MatchCase] | Char [MatchCase] | LitId [MatchCase]
        | Function [MatchCase] | Blanks | LineBreak
        | & [ AlternativeLink & ] [MatchCase] | & ( AlternativeLink & ) [MatchCase]
        | & { AlternativeLink & } [MatchCase] | & < AlternativeLink & > [MatchCase]
```

Blanks → ␣ ##blanks, next token ≠ !
LineBreak → & & \n
MatchCase → ##blanks Maximal | ##blanks Expect | ##blanks Taboo | ##blanks Except
Maximal → & +
Expect → & E Element
Taboo → & T Element
Except → & ^ Element
CatVar → # CatVarRec | # CatVarName | # CatVarId
CatVarRec → Id
CatVarName → > (Id | & t h i s)
CatVarId → = (Id | & t h i s)
LitVar → # # Id
LitId → & = & (ElementLink &)
FunArg → ElementLink
FunArgLink → FunArg [& , FunArgLink]
Function → # ! Id ##blanks & ([FunArgLink] &)
CatDef → ##eol [CommentLink] Id : [␣ Id +] \n [CommentLink] SpecLink [CatLink [IrrLink]]
SpecLink → Spec [SpecLink]
Spec → AllOfSpec | OneOfSpec | SomeOfSpec | OptionalSpec | FixedSpec | OnlySpec
    | SomeOfTypeSpec | ItselfSpec | ArraySpec | IndexSpec | TemplateSpec | NothingElseSpec
    | NothingSpec | UnionSpec | AtomicSpec | CompleteSpec
AllOfSpec → a l l O f > ##blanks EqLink
OneOfSpec → o n e O f > ##blanks EqLink
SomeOfSpec → s o m e O f > ##blanks EqLink
OptionalSpec → o p t i o n a l > ##blanks EqLink
FixedSpec → f i x e d > ##blanks EqLink
OnlySpec → o n l y > ##blanks EqLink
SomeOfTypeSpec → s o m e O f T y p e > ##blanks EqLink
ItselfSpec → i t s e l f > ##blanks NameLink
ArraySpec → a r r a y > ##blanks EqLink
IndexSpec → i n d e x > ##blanks EqLink
TemplateSpec → t e m p l a t e > ##blanks Id ##blanks \n
NothingElseSpec → n o t h i n g E l s e > ##blanks \n
NothingSpec → n o t h i n g > ##blanks \n
UnionSpec → u n i o n > ##blanks NameLink
AtomicSpec → a t o m i c > ##blanks NameLink
CompleteSpec → c o m p l e t e > ##blanks \n
EqLink → [# Id :] Id = Id (Comment | ##blanks \n) [##blanks EqLink]
NameLink → [# Id :] Id (, (␣ | ##eol ##blanks) NameLink | \n)
CatLink → CatProduction [CatLink]
CatProduction → ##blanks # Id > ␣
    (AlternativeLink (CommentLink | \n) | # (␣ CommentLink | ##blanks \n))
IrrLink → IrrProduction [IrrLink]
IrrProduction → ##blanks # Id > > ␣ # Id = AlternativeLink (CommentLink | \n)

For practical reasons, Concise does not use `TypeSheets.cnt` directly. Instead, I converted

the grammar by hand to a C++ program, `cnttoxml`, using DynGenPar's native C++ API. That `cnttoxml` program serves as a bootstrap parser for type sheets. It converts the type sheets to the XML representation of a record sheet of the corresponding `TypeSheets` type. The resulting XML record sheet is then converted to a Concise record sheet (`.cnr` file) with another C++ program, `xmltocnr`. `xmltocnr` walks the XML using the `QtXml` **DOM** (**Document Object Model**) API and produces the output in the Concise record sheet format, including correct indentation. Next, the record sheet is imported. Finally, a builtin converter inside Concise (written by Ferenc Domes) transforms the record of type `TypeSheets` to an actual type system of type `TypeSystem`.

I initially wrote the `xmltocnr` converter for the `TextDocument` toolchain, a toolchain for importing LaTeX document structure, which will be the subject of Section 6.2. I found XML to be an easier format to generate than the default record sheet format (`.cnr`) because XML is not sensitive to things such as indentation. However, I only use it as an intermediate format. The generated XML is entirely unreadable for humans due to the lack of line breaks and indentation. A tool called **HTML Tidy** (Raggett [76, 75], Desitter et al. [21], HTACG [37]) can make it human-readable, but still harder to read than the `.cnr` format, and significantly larger than the `.cnr` (about 2 to 4 times). Therefore, `xmltocnr` is used to obtain a `.cnr` file, which optimizes both readability and compactness.

The `cnttoxml` bootstrap parser produces the same records as the parser automatically generated from `TypeSheets.cnt` by the process from Section 3.2. This was verified using `TypeSheets.cnt` itself as the test input.

4.2 A Grammar for Code Sheets

Another use of DynGenPar in Concise is for **code sheets**, the fundamental programming language in Concise. Code sheets are the textual representation of **elementary acts**, the basic operations on the semantic memory. This section first documents the elementary acts, and then describes the grammar for code sheets and how it maps to the elementary acts.

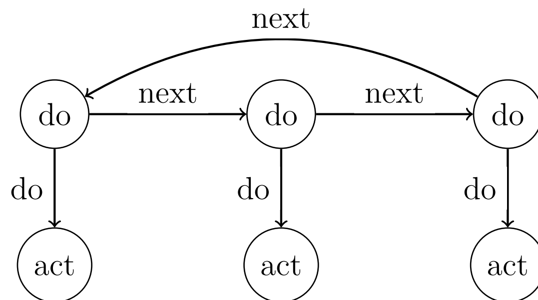
4.2.1 Elementary Acts

This section gives an overview of the internal representation for programs in Concise: **elementary acts**. The execution environment for those elementary acts was written by Ferenc Domes, the main author of Concise, and is therefore out of the scope of this thesis. (Though, some improvements and bug fixes to that implementation were done by me as part of the development of the grammar for code sheets.) However, the concepts of the representation have to be explained to understand the concept of code sheets. That is because code sheets directly map to that internal representation. I will therefore introduce them briefly.

The programming model on Concise is based on programs stored inside the same seman-

tic memory they operate on. For random access machines (i.e., the typical modern-day computer), one would speak about a Von Neumann architecture, as opposed to a Harvard architecture with separate program and data memory. This architectural decision was taken to allow straightforward reflection: Representing programs inside the semantic memory ensures they can be statically analyzed by other programs.

The program representation in the semantic memory replicates the control flow graph. The record representing an instruction is called an **act**. An act by itself just terminates when it is done. Therefore, there exists a special `Do` act, which can be used to wrap every act. The `Do` act carries an optional `next` entry which can point to another act, typically another `Do`. This allows forming linked lists of acts to represent instruction sequences. Thus, basic blocks (i.e., sequences of instructions one cannot jump out of nor into) become linked lists in the semantic graph. The simplest type of loop is simply a cycle in the semantic graph, chained through the entries for the `next` field of `Do`.



However, in practice, this loop representation cannot currently be generated from a code sheet. Instead, the cycle is completed in practice by a `Goto` act, which has a `next` pointer inside the act itself.

The `Do` act can carry, in addition to the chain pointer `next`, a condition under which the act should be executed, and an optional `else` act which is executed instead when the condition does *not* hold. This can be used to implement conditional instructions. Using another `Do` as the conditionally executed act, it can also represent conditional blocks. Conditionalizing the `Goto` act this way provides a termination condition for loops. Alternatively, a conditional `Goto` can also be used to jump out of a loop.

Finally, there are some constructs that influence the control flow beyond what is immediately visible from the structure in the semantic graph: There is a special kind of loop called `ForAllFields`, which loops over all fields a given object has at runtime, and repeats the same subordinate act for each. Moreover, there exist, of course, function calls and returns.

The set of acts currently implemented in Concise corresponds to the most basic operations one needs to do in the semantic memory. They are therefore called the **elementary acts**. In the future, it is planned to support a richer set of acts also containing some more complex acts. Those **complex acts** would then be compiled into elementary acts before execution.

There are 16 different command types defining the individual elementary acts. They are

defined in `ElementaryActs.cnt` (Kofler & Neumaier [57]), from where the documentation that follows is excerpted. They are:

1. `Do`: the standard control flow sequence, described above.
2. `Return`: return from a function call
   Use: `return`
   completes the current function act immediately, discarding the local frame and popping the previous frame.
3. `Goto`: control flow
   Use: `goto` x
   does not continue with the next act but executes x
4. `Assign`: assigning a variable
   Use: l=r
   assigns to l`.content` the entry of r`.content`. l must be a variable.
5. `Set`: assigning an entry
   Use: h.f=e
   replaces the current entry in position (h, f) by e.
6. `Get`: sem manipulation
   Use: e=h.f
   assigns to e`.content` the entry in h.f.
7. `GetType`: get type
   Use: x=`getType(`y`)`
   gets the type of y as a string (more precisely, a value of type `UniqueString`) into x.
8. `IsSubtypeOf`: check of subtype relationship
   Use: z=`isSubtypeOf(`x,y`)`
   assigns `true` to z if x is a subtype of y, and `false` otherwise.
9. `Identical`: object and value comparison
   Use: z=(x=y)
   assigns `true` to z if x and y contain the same object, and `false` otherwise.
10. `Convert`: standard conversion
    Use: y=`convert(`x,t`)`
    converts x to type t and assigns the result to y.
11. `Vcopy`: copy external value
    Use: x=`Vcopy(`y`)`
    copies the external type and value of y to x
12. `ForAllFields`: loop over fields
    Use: `forAllFields` f `of` x `{`
    a
    `}`
    loops over all fields of x, repeating a for each of them. In every iteration, f contains the object identifying the current field (not its entry).

13. `Call`: function call
    Use: `call` *f* `with`
    `in>` *listOfVariableSubstitutions*
    `inout>` *listOfVariableAssignments*
    `out>` *listOfVariableAssignments*
    `end`
    calls the function *f* with the specified input (`in`), bidirectional (`inout`) and output (`out`) arguments. Calling a function pushes the previous frame onto a function stack, and creates a new local frame for the control of its execution.

14. `Ask`: ask the supervisor
    Use: `ask` [*q*[`,` *r*]]
    reports an error the Concise way. The typical way to create errors is by testing for some error condition in a conditional `Do` wrapped around this `Ask`. If the `Ask` is executed, the supervisor is called and presented with the question object *q* and possibly the answer object *r*.

15. `Supervise`: supervise the underlying act
    Use: `supervise` [`with timeout` *t*] `{`
    a
    `}` [`dialog` [`receive` *q* [`,` *r*]] `{`
    d
    `}`]
    tries running *a*, and executes the dialog *d* if an error is caught or if the timeout *t* is reached. The timeout is measured in a pseudotime counting the sum of weights of individual elementary acts. The parameters *q* and *r* are passed from the `Ask`.

16. `Resume`: resume from supervisor
    Use: `resume`
    closes the current supervisor dialog and jumps back to the next task after the `Ask`.

(Kofler & Neumaier [57])

4.2.2 Code Sheets

For the Concise runtime environment for elementary acts, I provide a traditional text-based programming language: **Code sheets** are the textual representation of the elementary acts. DynGenPar is used to parse that text form into records that can be executed by the Concise runtime environment.

The code sheets are parsed using DynGenPar into a record of type `CodeSheet`. The grammar used for that purpose is given by a Concise type sheet `CodeSheets.cnt`, reproduced in Kofler & Neumaier [57]. That type sheet is automatically converted to a DynGenPar grammar using the grammar from Section 4.1 and the process from Section 3.2.

A code sheet consists of three optional parts:

1. an introductory comment block, consisting of comment lines. Each comment line must start with the `!` character.
2. a `Do` block, and
3. one or more function definitions.

"Optional" means that some or all of the three sections can be missing. However, due to technical limitations of the conversion process, all code sheets must currently have a root block. If the `Do` block is present, it is the root block. Otherwise, the first defined function is treated as the root. Any other function definitions are treated as subroutines of the root block and expected to be called directly or indirectly by it.

A `Do` block is enclosed between curly braces. In front of the opening brace, there may be an optional label followed by a colon. The label gives a name to the block, which is especially useful for the root block. It is also used for the targets of `goto` statements. Between the opening and the closing brace, there can be an arbitrary number of code lines. Those code lines which produce elementary acts are chained together in a linked list of `Do` records in the resulting record of elementary acts.

A function definition looks similar to a `Do` block, but the opening brace is preceded by a function prototype like the following:

```
function ok of ExternalTypes::Boolean = ...
    writeToFile(data of ExternalTypes::External, ...
                fileName of ExternalTypes::String):
  description> write a given external in a text file
  static> lastFileWritten of ExternalTypes::UniqueString = str:""
```

(The MATLAB-style `...` line continuations are part of the supported syntax.) The prototype may itself be preceded by an optional comment block. It shall be noted that the name of the function is the identifier *after* the `=` sign, in this case, ***writeToFile***. Before the `=` sign comes the list of output arguments with their name (e.g., ***ok***) and type. The list of input arguments with their name and type comes between the parentheses.

A code line can be any of the following:

- a comment line (starting with `!`). Those are entirely ignored during processing. The conversion process currently makes no attempt at filling in any `comment` fields in the elementary acts.
- a blank line, also ignored entirely.
- a variable declaration. Such a declaration does not directly produce any output, but it declares the meaning of a given identifier. Whenever that identifier is used in the correct scope, it is recognized as a variable. Accordingly, in the resulting elementary acts, a suitable variable record is created, and any subsequent use points to the same record.
- an object reference. Like variable declarations, object references declare the meaning of an identifier. In this case, the identifier is simply a shortcut for any object path

in Concise format. In addition to being shorter, it allows referencing paths such as `Users(English,System).CurrentUser(English,System)`, which are not valid identifiers.

- a type reference, declaring an identifier to be a shortcut for a full type reference. This allows using a short name instead of the full *typesheet*`::`*typename* syntax.
- an act line, corresponding to any of the elementary acts listed in the previous section. For `Do`, the syntax is the same as for the root `Do` block. For all other acts, the syntax is specified in the previous section. These are the lines that actually produce entries in the linked list of `Do` records. They can be followed by `if` and/or `ifNot` conditions, followed by an optional `else` block (a `Do` block executed if the condition is not satisfied). Those conditions are recorded in the resulting `Do` record for the line.

In order to test the code sheet parser, I converted several existing test cases for elementary acts from handwritten records to code sheets. Those test cases were originally created by Ferenc Domes to test the runtime environment. I rewrote them into code sheet syntax and verified that they produce a record equivalent to the original handwritten one. The records produced are either entirely identical to the handwritten versions or have only minor differences, such as a redundant `Do` wrapped around an act for technical reasons. I also tested them in the runtime environment, confirming that they execute correctly.

As an example for the code sheet syntax, the first of the above test cases is reproduced below:

```
! ActTest.cnc - act test
! code sheet converted from ActTest.cnr by Kevin Kofler
!
! ActTest.cnr is only a Do and not a Function.
actTest: {
  ! We need the string type more than once.
  typeref> String = ExternalTypes::String
  ! These could also be comma-separated in one line, both versions are allowed.
  global> cond of ExternalTypes::Boolean = bol:true
  global> a of String = str:"1975"
  global> b of ExternalTypes::MutableExternal = int:0
  b = convert(a, ExternalTypes::Integer) if cond
  system->alert(out=b)
  out=time system->getTime()
  ! We can declare variables where needed.
  global> c of ExternalTypes::MutableExternal = int:0
  global> time of String = str:""
  c = int:22
  c = time
  objref> testUser = testUser(English,System)
  objref> Library = Library(English,System)
  testUser.setTest = Library
}
```

DynGenPar uses the above grammar to produce a record of type `CodeSheet`. This record is then converted into elementary acts using a record transformation (see Section 4.3). The resulting executable record can then be processed by the runtime environment in Concise.

I considered several different options for the implementation language of that conversion process. The original idea was to always perform this type of conversions in the elementary acts language. However, it was not possible to use code sheets for this purpose before the code sheet conversion was implemented, a chicken-and-egg problem. I also found it highly impractical to attempt to write the converter directly in elementary act records. (After all, this is why I was implementing code sheets in the first place.) Another possibility would have been to hardcode the conversion inside the Java code of Concise as Ferenc Domes did for the type sheets, but I did not find that to be a satisfactory approach either. Therefore, I decided to try a third approach – using a language explicitly designed for record conversions: **XSLT** (**eXtensible Stylesheet Language Transformations**) (W3C [100]), on the XML representation of the record. The stylesheet reproduced in Kofler & Neumaier [57] handles the conversion from the parser-level syntax record to a record of executable elementary acts.

My experience with using XSLT for this purpose was mixed. Most of the conversion process is just a matter of mapping input records to similarly structured output records. XSLT excels at such straightforward remapping, making it convenient to implement in XSLT. But unfortunately, converting from a syntax record to executable records also requires some semantic analysis, and XSLT very quickly showed its limitations there. In particular, I needed to resolve variable references. In the code sheet record, a variable reference is encoded as a string giving the name of the variable. In the executable record, the reference must point to the actual variable in the semantic memory. Therefore, the converting style sheet needs to look up the correct variable to point to. Implementing this resolution in XSLT required some very long and ugly XPath expressions.

Therefore, the conclusion was that the ideal implementation language for such a converter would be very similar to XSLT, but would additionally provide some builtin support for name resolution. This is what triggered the design of the record transformation language (see Section 4.3). The record transformation sheet converting code sheets to elementary acts is reproduced in Kofler & Neumaier [57].

4.3 A Grammar for Record Transformation Sheets

This section is based on the technical report Kofler [49].

The **record transformation language** transforms one record (labeled directed graph with specified root node) into another. As explained in Section 3.3.3, this record transformation functionality is fully integrated in the Concise GUI. A document in the record transformation language is called a **record transformation sheet**.

The design concepts for the record transformation language are:

- record transformation sheets are essentially structured like XSLT (W3C [100]) style

sheets, but

- with a more Concise (Schodl et al. [87], Domes [22]) like syntax, and
- most importantly, without the hacks! In particular:
  - whereas the Concise XML record representation uses `<FIELD name="`*field name*`"><`*type name*`>` nesting, in the record transformation language, a field is really only one layer deep, and types are checked separately,
  - `setlabel` (Section 4.3.7.3) and `pointtolabel` (Section 4.3.7.4), which avoid the ugly XPath (W3C [99]) expressions for name resolution, along with `newscope` (Section 4.3.6.4.3) to create a new label scope,
  - a way to create new roots that can be attached through `setlabel` (Section 4.3.7.3) and `pointtolabel` (Section 4.3.7.4), so we do not have to figure out where to emit objects pointed to from more than one place (see the `new` block in Section 4.3.6.4.1).

4.3.1 Basics

Basic principle: The record transformation language transforms one record (labeled directed graph with specified root node) into another. At any point in the process, there are 2 current nodes: the current input node and the current output node. Initially, the current input mode is the root node of the input record, the current output node is a newly-created node that will become the root node of the output record. Some commands of the form:
command`>`
  nested context
`<`
affect the current input and/or output node. In that case, a different node is current within the nested context, including anything called from it. After the `<`, we are back to the input and output node that were current before the command.

Conceptually, the record transformation language is strongly inspired by XSLT (W3C [100]), a language to transform between different XML representations. However, the concepts were adapted to operate (directly) on Concise (Schodl et al. [87], Domes [22]) records instead, and the syntax matches the syntax of Concise record sheets.

Some general rules:

- By default, if multiple rules match, they are **all** applied, in the order in which they appear in the record transformation sheet file. If only the first matching rule out of a set of rules shall be applied, we use the `> otherwise` *COND* `<` construction. The block following that is applied only for input nodes that did `not` match the preceding blocks in the `otherwise` chain, but do match the condition *COND*. Multiple `otherwise` constructs form a chain in which the first block that matches is used (like an else-if chain).
- In XSLT (W3C [100]), anything unmatched is just copied. For us, copying is probably not a good default. Currently, anything unmatched is simply ignored.

4.3.2 Document Structure

A **record transformation sheet**, i.e., a document in the record transformation language, is structured as follows:

- It may start with an optional **comment header**, the only thing that can come before the header line. The comment header can contain only comment lines (Section 4.3.2.1) and blank lines.
- The sheet must be introduced by a **header line** (Section 4.3.2.2).
- What follows is a **body** composed of one or more **contexts** (Section 4.3.2.3). Comment lines (Section 4.3.2.1) and blank lines are also allowed.

4.3.2.1 Comments

Comments start with a `!` sign and optional blanks and end with a newline. In the semantic memory, only the string in between is stored, without the leading blanks. Comment lines can come at almost any place in the record transformation sheet.

4.3.2.2 Header Line

The header line in its simplest form is of the form:
source type system[`(`*language*`)`*]* `->` *destination type system[*`(`*language*`)`*]*
e.g.:
`CodeSheets -> ElementaryActs`
which is the same as:
`CodeSheets(English) -> ElementaryActs(English)`
(English is the default.)

A record transformation sheet can also have parameters:
source type system[`(`*language*`)`*]* `->` *destination type system[*`(`*language*`)`*]*`>`
  variable name[`(`*type name*`)`*] [*`=` *default value]*
  ...
`<`
e.g.
```
RobustAmpl -> RobustOptProb>
  rigorous(Boolean) = true
<
```
Each parameter line defines a parameter global to this instance of this record transformation sheet. It is treated as a global variable (Section 4.3.3). It should be set when invoking the record transformation. A default value can optionally be specified.

4.3.2.3 Contexts

A context has the syntax:
[context name][`(`*type name*`)`*]*`>`

block contents
<

An empty context name symbolizes the root context: The root context is what is automatically applied on the root node. Named contexts, on the other hand, must be explicitly applied using the `apply` (Section 4.3.7.1) command.

An empty type name means no type checking is performed, i.e., the context accepts any type (including untyped objects).

The block contents can be command blocks (Section 4.3.6), command lines (Section 4.3.7), comment lines (Section 4.3.2.1), and/or blank lines.

As in Concise (Schodl et al. [87], Domes [22]) record sheets, multiple closing < characters can be optionally collapsed into a single line, i.e., instead of, e.g.:
<
<
one can also write:
< <
(as in record sheets).

4.3.3 Variables

An important feature of the record transformation language is imperative variables. This is unlike XSLT (W3C [100]), which intentionally omits this feature to allow implementations more flexibility in the evaluation order. With one exception (the `otherwise` block to the `pointtolabel` command, Section 4.3.7.4), the record transformation language has a very straightforward evaluation order, which allows using variables intuitively.

In the record transformation language, imperative variables are also the way to make decisions which depend on the content of sibling nodes. In XSLT (W3C [100]), you would use an XPath (W3C [99]) expression matching the sibling node (matching from the root or using the `..` path operator). This is not possible in the record transformation language because parents and siblings are not a well-defined concept in the semantic memory. So the way it is done is that one first has to match the sibling node on which the decision should depend, and store the information that matters in one or more variable(s), and only then match the node in which the decision should happen, and decide based on the value of the variable(s). The variables are guaranteed to be set and read in that order.

Variables in the record transformation language are scoped:

- The outermost (**global**) scope contains the record transformation parameters (Section 4.3.2.2).
- Each instance of a context (Section 4.3.2.3) opens its own variable scope.

All variables are per-instance. They are **not** remembered between separate runs of the record transformation. Context variables are also **not** remembered from one invocation of the context to the next, and each recursive invocation has its own independent copy of the variable.

Within any context, a variable (which will be local to this instance of that context) can be declared using the `var` command (Section 4.3.7.5). Invoked contexts can refer to the variable (scope inheritance), but if they define their own variable of the same name, it takes precedence (shadowing).

Variables can be referenced in any expression, in some runtime identifiers (field names, scope names, label names), and on the left hand side of an assignment (Section 4.3.7.6). They are referenced using the `#` expression operator (Section 4.3.4). The variable that is used is always the innermost variable in scope, i.e., the copy in the innermost scope that contains the variable. Only if the scope has no variable of that name, the next outer scope is looked up, etc., until the global scope.

A value can be assigned to a variable using the special `:=` operator, which forms a command on its own (the assignment command, Section 4.3.7.6). The value is always assigned to the innermost copy of the variable, as above.

To remember information from one sibling node to another (a common usage of variables, as explained near the beginning of the section), the variable should be:

- declared in the **parent** node (using the `var` command, Section 4.3.7.5),
- assigned in the sibling that has the information (using the assignment command, i.e., the `:=` operator, Section 4.3.7.6), and
- read in the sibling that needs the information (using a variable reference, i.e., the `#` operator, Section 4.3.4). The `if` (Section 4.3.6.2.1) or `switch` (Section 4.3.6.3) conditionals may be of use to make the actual decision. The variable can also be used, e.g., to fill in the value of a `setexternal` command (Section 4.3.7.8).

This is because the parent is the common scope between the two siblings.

4.3.4 Expressions

Several commands in record transformation sheets accept expressions that evaluate to Concise (Schodl et al. [87], Domes [22]) externals. For those, a common `ExternalExpressions` grammar is offered, which the record transformation language extends with 3 application-specific atomic expression types.

The following types of atomic expressions are accepted by the `ExternalExpressions` grammar:

- **bracketed expressions**: Any valid expression can be turned into an atomic expression by surrounding it with parentheses '`(`' and '`)`', bypassing the usual operator priorities.
- **constants**:
  - the **boolean constants** `true` and `false`,
  - non-negative **integer constants**,
  - non-negative **(double-precision) floating-point constants** (e.g., `1.`, `.1`, `1.1`, `1e-1`, `1.1e-1`, etc.),

  - quoted **string constants**, which must be enclosed in double quotes '`"`'. The escapes `\\` (backslash), `\"` (quote) and `\n` (newline) are recognized.

  Note that negative numeric constants are **not** treated as constants, but as the unary minus operator applied to a non-negative constant. This is required to get operator priorities right.

- **type cast expressions**: `(`*type name*`)` *atomic expression*

- **function calls** of a Concise (Schodl et al. [87], Domes [22]) external function or a Concise function implemented in elementary acts inside the semantic memory

The record transformation language adds the following 3 atomic expression types:

- **variable references**: `#`*variable name* (maximally matched) or `#{`*variable name*`}` (see Section 4.3.3).

- **dollar references**: `$` or `${`*path match (Section 4.3.5.1)*`}`
  The dollar operator stands for a reference to the value of the current input node, or its descendant with the given path, which should be of an external type (otherwise, an error is triggered). If type matches are given in the path, the given types are validated and an error thrown on mismatch. The type of the current node can also be validated. `$` with no path match and `${}` with the empty path match are syntactically different, but semantically the same.

- **label existence checks ("have label")**: The `?{`*scope*`::`*label*`}` operator returns `true` if and only if the label *scope*`::`*label* exists, i.e., was defined by a prior `setlabel` command (Section 4.3.7.3), in the current scope for the scope name *scope* at the time of evaluation. See `pointtolabel` (Section 4.3.7.4) for how label scopes are resolved.
  Warning: Unlike `pointtolabel` (Section 4.3.7.4), order of processing matters in this case. If the label is encountered only **after** this operator is executed, it is too late, and the operator will thus already have returned `false`. It can only return `true` if the label is seen **before** this operator is executed.

The following operators are accepted by the `ExternalExpressions` grammar, in decreasing priority order:

1. **product operators**: `*` (times), `/` (division), and `%` (modulo)

2. **sum operators**: `+` (plus), `-` (minus), unary `-` (unary minus)

3. **comparison operators**: `==` ($=$), `~=` ($\neq$), `>` ($>$), `>=` ($\geq$), `<` ($<$), `<=` ($\leq$)

4. **unary boolean not**: unary `~`

5. **boolean and**: `&`

6. **boolean or**: `|`

If not otherwise specified, operators are binary and left-associative.

4.3.5 Path and Path Set Matches

Several commands in the record transformation language need to match paths. In XSLT (W3C [100]), XPath (W3C [99]) expressions are used for that purpose. The record transformation language uses a similar concept, with the following main differences:

- The Concise (Schodl et al. [87], Domes [22]) type system (Schodl & Neumaier [86]) is taken into account: At every step in the path, a type check can be performed. The operator searching for a descendant node also takes a type and finds only nodes of the requested type.
- Going up in the path, i.e., matching parent or sibling nodes, is not supported because parents and siblings are not a well-defined concept in the semantic memory. Instead, use imperative variables (Section 4.3.3) to work with sibling nodes. If what you are trying to do is name resolution, use the `setlabel` (Section 4.3.7.3) and `pointtolabel` (Section 4.3.7.4) commands. The "have label" operator (Section 4.3.4) may also be of use for that use case.
- The syntax was adapted to match Concise (Schodl et al. [87], Domes [22]) syntax conventions.
- Not all constructs of the XPath (W3C [99]) language have an equivalent.

Unlike XPath (W3C [99]), there are two versions of path matches:

- simple **path matches** (Section 4.3.5.1) with a restricted syntax that necessarily resolves to at most one node (in particular, descendant matches are **not** allowed in the path), and
- **path set matches** (Section 4.3.5.2) with an extended syntax (including descendant matches and match conditions) that can produce sets of paths.

4.3.5.1 Path Matches

A **path match** matches the type of the current node: `(`*type name*`)` and/or a chain of field matches (or neither, then it is the empty match, referring to the current input node with no type checking). A **field match** `.`*field name[*`(`*type name*`)`*]* is a field name (an identifier preceded by the `.` operator) and an optional type name to match. The field name can also contain dollar references or variable references (see Section 4.3.4), which must be of a type convertible to String. Their value will be converted to a String and concatenated into the name. Multiple field matches can be chained to a path, e.g., `.exponent(UnaryMinus).term(BigInteger)`. All paths are relative to the current input node.

4.3.5.2 Path Set Matches

A **path set match** is an extended version of a path match that can match entire sets of paths. In addition to the constructs allowed in path matches (a match on the type of the node and/or field matches), it also allows:

- **descendant matches** `..(`*type name*`)` match any descendant of the given type of a node, where a descendant is defined as either the node itself, or one of its children, or one of its children's children, etc.
- optional **match conditions** on each field or descendant match:
  - a `[?`*path*`]` condition requiring that a given subpath exists, or
  - a `[~?`*path*`]` condition requiring that a given subpath does not exist,

  where *path* can again be any path set match, or
  - an `[`*expression*`]` condition that is satisfied if the expression evaluates to `true`, when evaluating any dollar references (see Section 4.3.4) relatively to the node being matched (rather than the surrounding current input node).

4.3.6 Command Blocks

Blocks are of a form similar to contexts:
introductory block command>
  block contents
<

The block contents can again be command blocks (Section 4.3.6), command lines (Section 4.3.7), comment lines (Section 4.3.2.1), and/or blank lines.

As in Concise (Schodl et al. [87], Domes [22]) record sheets, multiple closing < characters can be optionally collapsed into a single line, i.e., instead of, e.g.:
  <
<
one can also write:
< <
(as in record sheets).

4.3.6.1 Matching and Enumerating Blocks

The following blocks can be chained in an `otherwise` chain:
[match condition]>
  block contents
< `otherwise` *[match condition]*>
  block contents
...
>
Only the contents of the first block that matches are executed.

4.3.6.1.1 Matching block. A **matching block** matches the type of the current input node and/or of one of its descendants (with the specified path). The current output node within the block is the matched node. Syntax:
path match (Section 4.3.5.1)>

The matching block operates on a uniquely determined path. (In particular, `..(`*type name*`)` descendant matches are **not** allowed in the path.) To loop through a set of paths, use `foreach` (Section 4.3.6.1.2). To merely check whether a set of paths is nonempty, use `ifexists` (Section 4.3.6.2.2).

The `otherwise` block, if given, is executed if there was no match.

4.3.6.1.2 `foreach` block.

A **`foreach` block** loops through all the nodes in a given path set. The current output node within the block at each iteration is the matched node. To merely check whether a set of paths is nonempty, without affecting the current output node or looping, use `ifexists` (Section 4.3.6.2.2). Syntax:
`foreach` *path set match (Section 4.3.5.2)*`>`
Whitespace is required after `foreach`.

The `otherwise` block, if given, is executed if there was no match (i.e., if the set was empty).

4.3.6.1.3 `forallfields` block.

A **`forallfields` block** loops through all the fields of the current output node, optionally restricted to fields where the field (**not** the entry) has a given type. The current output node within the block at each iteration is the **entry** corresponding to the matched field.

Syntax:
`forallfields` *[*`(`*type name*`)`*]* *[*`#`*variable name]*`>`

If a *variable name* is given, the field contents (**not** the entry contents) will be stored in the variable at each iteration. This is only supported if the field type (**not** the entry type) is external. Otherwise, an error is triggered. (You may want to restrict the loop to a suitable external type to prevent that.)

The `otherwise` block, if given, is executed if there was no match.

4.3.6.2 Conditional (if-else) Blocks

The following blocks can be chained in an `else` chain:
`if`*[*`exists`*] condition*`>`
  block contents
[`< else`*[ ]*`if`*[*`exists`*] condition*`>`
  block contents]
...
[`< else >`
  block contents]
`>`
Only the contents of the first block that matches are executed.

4.3.6.2.1 `if` block. An **`if` block** checks a condition. Syntax:
`if` *condition (expression, Section 4.3.4)*`>`
Whitespace is required after `if` except for the sequence `if(`.

4.3.6.2.2 `ifexists` block. An **`ifexists` block** checks whether a given path set is nonempty, i.e., whether one or more nodes with the given path exist. Syntax:
`ifexists` *path set match (Section 4.3.5.2)*`>`
Whitespace is required after `ifexists`.

Unlike the matching (Section 4.3.6.1.1) or `foreach` (Section 4.3.6.1.2) block, this does **not** change the current output node within the block. Use `foreach` to operate on the nodes in the set.

4.3.6.3 Switch Blocks

A **`switch` block** makes a case distinction. It is basically syntactic sugar for an if-else chain, though there is also a minor performance advantage because the common expression is evaluated only once.

Syntax:
`switch` *expression*`>`
  [`case` *expression*`>`
    block contents
  `>`*]*
  ...
  [`default>`
    block contents
  `>`*]*
`>`
Whitespace is required after `switch` except for the sequence `switch(` and after `case` except for the sequence `case(`.

4.3.6.4 Constructor Blocks

4.3.6.4.1 `new` block. A **`new` block** creates a new node that is **not** attached to the current output node, and makes that the current output node within the block. Syntax:
`new` *[type name]*`>`

This is normally used together with `setlabel` (Section 4.3.7.3), so it can be attached at a different point of the record using `pointtolabel` (Section 4.3.7.4).

4.3.6.4.2 `newfield` block. A **`newfield` block** creates a field with the given *field name* pointing to a new entry node in the current output node, optionally sets the entry node's type, and makes it the current output node within the block. Syntax:
`newfield` *field name[*`(`*type name*`)`*]*`>`

Alternatively, instead of specifying a *field name*, it is possible to construct a new object (or an external, using `setexternal` (Section 4.3.7.8)) which will become the field using the following **alternate syntax**:
`newfield>`
  [block contents for field contents]
`< =` *[*`(`*type name*`)`*]*`>`

Specifying the type name is syntactic sugar for `settype` (Section 4.3.7.2), i.e.:
`newfield` *fieldName*`(`*typeName*`)>`
is equivalent to:
`newfield` *fieldName*`>`
  `settype` *typeName*

4.3.6.4.3 `newscope` block. A **`newscope` block** creates a new label scope, valid within the block (see also the `pointtolabel` line, Section 4.3.7.4). Syntax:
`newscope` *scope name*`>`

The scope name can also contain dollar references or variable references (see Section 4.3.4), which must be of a type convertible to String. Their value will be converted to a String and concatenated into the name.

4.3.6.4.4 `newlist` block. A **`newlist` block** builds a list of type *type name* at the given *field name* (or if *field name* is omitted, at the current output node; in particular, this allows building lists in the root node of the output record). Syntax:
`newlist` *[field name]*`(`*type name*`)`*[*`.`*next name]*`>`
The block may optionally be followed by an `< otherwise >` block (with no arguments).

The list is initially empty, which means *field name* does not initially point to anything (or if *field name* is omitted, the current output node is initially not modified). Therefore, this command does not by itself change the current output node. Use an `append` block (Section 4.3.6.4.5) to actually create the list (by appending the first element). The *next name* is the name that should be given to the *next* chaining field in the list. If not specified, it defaults to `next`.

The optional `otherwise` block is used if nothing was appended to the linked list using `append`. In that case, the field will not have been created. (If no `otherwise` block is given, nothing happens. If you want an error to be raised for an empty list, use the `error` command (Section 4.3.7.10) in the `otherwise` block.)

4.3.6.4.5 `append` block. An **`append` block** appends an entry to the list in the innermost `newlist` block (Section 4.3.6.4.4). Syntax:
`append` *[field name[*`(`*type name*`)`*]]*`>`

When used without an argument, only the linked list node is created and becomes the current output node within the block. Passing an argument is a shortcut to create a data node (as used in many linked lists), i.e.:
`append` *fieldName*`(`*typeName*`)>`

is equivalent to:
`append>`
  `newfield` *fieldName*`(`*typeName*`)>`

The type name can be omitted, in which case the type will not be set, as for the `newfield` block (Section 4.3.6.4.2). Therefore, the above examples are also equivalent to:
`append fieldName>`
  `settype` *typeName*
and:
`append fieldName>`
  `newfield` *fieldName*`>`
    `settype` *typeName*
(See Section 4.3.7.2 for `settype`.)

4.3.7 Command Lines

4.3.7.1 `apply` Line

The **`apply` line** applies the context with the given *context name*, or the root context if no name is given. Syntax:
`apply` *context name*

4.3.7.2 `settype` Line

The **`settype` line** sets the type of the current output node. Syntax:
`settype` *type name*

4.3.7.3 `setlabel` Line

The **`setlabel` line** gives a label to the current output node, which can be referenced by `pointtolabel` (Section 4.3.7.4). Syntax:
`setlabel` *scope*`::`*label*

Both the scope name and the label name can also contain dollar references or variable references (see Section 4.3.4), which must be of a type convertible to String. Their value will be converted to a String and concatenated into the name.

See `pointtolabel` (Section 4.3.7.4) for how label scopes are resolved.

4.3.7.4 `pointtolabel` Line

The **`pointtolabel` line** points to a `setlabel` (Section 4.3.7.3) label, even if the label is encountered only later. Syntax:
`pointtolabel` *scope*`::`*label*
The line may optionally be followed by an `otherwise >` block (with no arguments).

Both the scope name and the label name can also contain dollar references or variable references (see Section 4.3.4), which must be of a type convertible to String. Their value will be converted to a String and concatenated into the name.

The label resolution works as follows: When looking for the label `foo::bar`, we look first for a `setlabel foo::bar` inside the innermost `newscope foo>`, then in the next outer scope etc., then finally outside of any `newscope foo>`, i.e., in the global `foo` scope. Any labels with another context scope, e.g., `bla::bar`, are ignored entirely.

The optional `otherwise` block is used if no label with the given name can be found in the appropriate scope. (If no `otherwise` block is given, an error is raised in that case.) Please note that label resolution may be done in a later phase, and thus any object created in the `otherwise` block can be created and later discarded by the implementation. Therefore, the `otherwise` block **must not** trigger errors even if the label exists, and the effects of using `setlabel` (Section 4.3.7.3) inside the `otherwise` block of `pointtolabel` are **undefined**.

4.3.7.5 `var` Line

The **`var` line** defines a variable local to this instance of the current context. Invoked contexts can refer to the variable (scope inheritance), but if they define their own variable of the same name, it takes precedence (shadowing). Syntax:
`var` *variable name[*`(`*type name*`)`*]*

See Section 4.3.3 for more information about variables.

4.3.7.6 Assignment Line

The `:=` operator assigns a value to a variable. Syntax:
variable reference (Section 4.3.4) `:=` *value (expression, Section 4.3.4)*

See Section 4.3.3 for more information about variables.

4.3.7.7 Function Call Line

One way to call functions is on the right hand side of an assignment (Section 4.3.7.6). However, some functions do not return a value, or you are not interested in the return value. Therefore, it is also possible as a special case to call functions only for their side effects, by putting the function call (Section 4.3.4) on its own line, as a statement by itself.

4.3.7.8 `setexternal` Line

The **`setexternal` line** replaces the current output node with an external object of the type and value of the given expression. Use a type cast expression (Section 4.3.4) to force a specific type. Syntax:

`setexternal` *value (expression, Section 4.3.4)*
Whitespace is required after `setexternal` except for the sequence `setexternal(`.

4.3.7.9 `setname` Line

The **`setname` line** sets the name of the current output node to the value of the expression *name*. The result will be converted to a string. It can be in any of the Concise (Schodl et al. [87], Domes [22]) name formats, e.g., `Foo`, `Foo(English,System)`, etc. Syntax:
`setname` *name (expression, Section 4.3.4)*
Whitespace is required after `setname` except for the sequence `setname(`.

4.3.7.10 `error` Line

The **`error` line** throws an exception with an implementation-defined message containing the value of the expression *message* (which will be converted to a string). Syntax:
`error` *message (expression, Section 4.3.4)*
Whitespace is required after `error` except for the sequence `error(`.

4.3.7.11 `warning` Line

The **`warning` line** displays an implementation-defined message containing the value of the expression *message* (which will be converted to a string). The way the message is displayed (`stderr`, `stdout`, dialog box, etc.) is also implementation-defined. Syntax:
`warning` *message (expression, Section 4.3.4)*
Whitespace is required after `warning` except for the sequence `warning(`.

4.4 A Grammar for Chemical Process Modeling

Another formal language DynGenPar has been applied to is a modeling language for chemical processes. The idea came from Ali Baharev and Arnold Neumaier, who had been working on optimization techniques for chemical process simulation. (Baharev & Neumaier [7]) They had attempted using Modelica (Mattsson et al. [64], Modelica Association [68]) to model the chemical processes. (Baharev & Neumaier [6]) Unfortunately, neither the syntax nor the available implementations of Modelica proved satisfactory. The syntax did not allow to conveniently express chemical processes. As for the implementations, the primary issues with them were performance and bugs. Therefore, I was tasked with designing a specialized modeling language for chemical processes.

Together with Ali Baharev and Arnold Neumaier, I designed a modeling language in which such chemical processes can be expressed conveniently. I called the language **`ChemProcMod`**, short for "**Chem**ical **Proc**ess **Mod**eling". The main objective was to make the syntax intuitive for a chemical engineer. Ali Baharev's input proved invaluable for that, due to his chemical engineering background. I implemented the grammar

as a Concise type sheet, parsed using DynGenPar. The type sheet `ChemProcMod.cnt` is reproduced in Kofler & Baharev [51].

The syntax was designed to follow the conception of the engineer rather than of a mathematician or computer scientist. As a result, it is declarative rather than imperative, and it uses vocabulary from the chemical application, not from the mathematical model or from object-oriented programming. For example, where Modelica would speak of a *class*, we instead call the class by what it actually is, e.g., an **atomic unit**, a **composite unit**, etc.

The syntax is line- and block-oriented. A file can consist of several elements, some of which are line elements, e.g.:

- comments, introduced by a `%` sign, or
- `import:` statements, e.g.
  `import: UnitLibrary (C = C, VLE := custom VLE, enthalpy := enthalpy)`,

whereas many are block elements, introduced by a line like
`atomic unit: heat exchanger {`
ending with an opening brace, and terminated with a closing brace `}`.

Blocks themselves consist of a list of elements between the opening and the closing brace. Those elements can also be lines or (nested) blocks. Some elements, such as parameter or variable definitions, are allowed everywhere; others can occur only in specific contexts. For example, the following elements can only occur at the top level of the file, not within another block:

- `import:` lines, which import the contents of another file,
- `stream` blocks, which declare the defining variables of every stream flowing in the model,
- `model` blocks, which define reusable sets of equations that are independent of any unit, typically thought of as **model equations** by the engineers,
- `atomic unit` blocks, which define an **atomic unit**, a building block from which more complex units can be composed,
- `composite unit` blocks, which define a **composite unit**, built from atomic units or other composite units,
- `flexible unit` blocks, which specify a list of possible unit types from which one can be picked, like a `union` in the C programming language, and
- `process` blocks, which define a **process**, a self-contained unit that can be simulated on its own. They can be thought of as the main class in a programming language.

On the other hand, many elements are only allowed within specific types of blocks. For example, **inlets** and **outlets** are ports through which matter enters resp. exits a unit. They only make sense within a unit. Therefore, `inlets:` and `outlets:` lines are only allowed within `atomic unit`, `composite unit` and `flexible unit` blocks.

The example below (Kofler & Baharev [51]) specifies a **divider**, an atomic unit that divides a stream into two streams such that the first is ζ (zeta) times as strong as the second, in the ChemProcMod language:

```
atomic unit: divider {
  inlets: i
  outlets: o1, o2

  variable: zeta .. real number

  equations {
    o1.f = zeta * o2.f
    o1.H = zeta * o2.H
  }
}
```

The modeling language comes with a **unit library** (UnitLibrary.cpm), consisting mostly of atomic units. It is reproduced in Kofler & Baharev [51]. The example unit above comes from that UnitLibrary.cpm file. The unit library evolved from Ali Baharev's unit library (Baharev & Neumaier [6]) in Modelica. I converted it to ChemProcMod, and we made some iterative refinements to it together. The concept of ChemProcMod is to hierarchically build more complex units using those atomic units as building blocks. A few such composite units are included in the unit library as well.

Ali Baharev and I also implemented two example problems in the language, reproduced in Kofler & Baharev [51]:

- JacobsenTest.cpm, the first practical test case for both the ChemProcMod language and the parser. This is an example for multiple steady-states in ideal two-product distillation from Jacobsen & Skogestad [42]. The model equations are taken from Baharev et al. [5].

- A parametrized problem consisting of the two files RD.cpm and RD_params.cpm. This example gives the steady state model of a reactive distillation column for ethylene glycol synthesis, taken from Ciric & Miao [15]. The column corresponds to the cost-optimal column of Ciric & Gu [14]. In the column, ethylene glycol is produced from ethylene oxide and water.

In addition, I wrote a ParserTest.cpm test file that has as its only purpose to test language features that do not appear in the unit library nor in the example problems, in order to ensure that the parser processes them correctly. That test file has no pretense of making any sense from a chemical engineering point of view, it only serves to test the parser. It is reproduced in Kofler & Baharev [51].

The above set of test files, namely, the unit library, the example problems, and the parser test, ensure the quality of the ChemProcMod grammar.

4.5 An Extensible Grammar for Optimization Problems

An additional application of DynGenPar based on a formal language came out of the **COCONUT** (Schichl [82], Neumaier & Schichl [71]) project. It showcases the dynamic extensibility of DynGenPar. COCONUT (COntinuous CONstraints – Updating the Technology) is a framework for solving global optimization and continuous constraint satisfaction problems. In COCONUT, there is a need for an input format that is extensible for new language features. For example, a recent research project involves solving optimization problems on Lie groups with COCONUT; for that project, the language would be extended with Lie groups and related operators. Hermann Schichl, the maintainer of COCONUT, realized that the dynamic properties of DynGenPar are perfectly suited for this problem.

Therefore, I implemented a DynGenPar grammar for optimization problems, called **`OptProbl`**. The input format for the optimization problems is a subset of LaTeX. The main feature of the grammar is that, as in LaTeX, the `\newcommand` command can be used to define new commands. E.g., `\newcommand{frac}[2]{}` defines a new `frac` operation with two parameters. (It is also possible to give a replacement expression between the braces, e.g., to automatically replace a fraction with a division. But that substitution is performed only after parsing.) What makes the feature special is that every such new command is actually a new pair of rules, added to the grammar at runtime. This feature is where the dynamic extensibility of DynGenPar is put to use. Future versions can easily add additional forms of extensibility: The current implementation only allows defining `\newcommand`-style commands. In the future, the user could be allowed to define, e.g., binary operators such as `\times` in a similar way. The implementation would follow the same approach.

An important restriction is that the expressions in the LaTeX markup must be given in an unambiguous syntax known from programming languages. For instance, implied multiplication (e.g., $2x$) is not supported, the $*$ operator must be used (e.g., $2 * x$). That is the reason why this grammar is classified as a formal language. A grammar that accepts natural LaTeX with its inherent ambiguities will be presented in Section 6.5.

The implementation is written in C++, using the DynGenPar API (application programming interface) directly. The initial grammar rules are given as C++ code compiled directly into the parser. User-defined rules can be added at runtime through constructs that are part of the recognized `OptProbl` input language. The choice of C++ as the implementation language was made in order to allow easy integration into the COCONUT project, which is also written in C++. Therefore, the `OptProbl` application is not based on the Concise framework.

A BNF (Backus-Naur form) version of the grammar, manually converted from the C++ source file `optprobl.cpp`, is reproduced below. The grammar is almost context-free, but it contains one single instance of a non-context-free extension to the BNF notation. The extension handles the peculiar way LaTeX treats the end of a tag. The tag ends at the first non-alphanumeric character. But if that character is a space, the space is consumed.

This is implemented in the grammar as a "taboo" constraint on the next token in the definition of the *TagEnd* category. In addition, the rules for *NewCommand* trigger a parse action that extends the grammar. This is marked in the grammar with the '¤' symbol. The parse action is described below, after the BNF.

Tokens: the set of 8-bit characters 0 . . . 255
Start category: *OptProbl*
Productions:

Whitespace → ε | *Whitespace Whitespace1*
Whitespace1 → ␣ | \ \ | \n
Letter → A . . . Z | a . . . z
Digit → 0 . . . 9
WordChar → *Letter* | *Digit*
WordChars → *WordChar* | *WordChar WordChars*
TagEndTaboo → *WordChar* | ␣
TagEnd → ε [next token ↛ *TagEndTaboo*] | ␣
BracketedExpr → (*Whitespace Expr Whitespace*)
Var → *Letter* | \ l a m b d a *TagEnd*
NumChar → *Digit* | .
NumChars → *NumChar* | *NumChar NumChars*
PosNum → *NumChars* | *NumChars* e *NumChars* | *NumChars* e - *NumChars*
NegNum → - *Whitespace PosNum*
Num → *PosNum* | *NegNum*
Interval → [*Whitespace Num Whitespace* , *Whitespace Num Whitespace*]
Placeholder → # *Digit*
AtomicExpr → *BracketedExpr* | *Var* | *PosNum* | *Interval* | *Placeholder*
CaretExpr → *PowerExpr Whitespace* ^ { *Whitespace Expr Whitespace* }
PowerExpr → *AtomicExpr* | *CaretExpr*
TimesExpr → *ProductExpr Whitespace* * *Whitespace PowerExpr*
DivExpr → *ProductExpr Whitespace* / *Whitespace PowerExpr*
ProductExpr → *PowerExpr* | *TimesExpr* | *DivExpr*
PlusExpr → *Expr Whitespace* + *Whitespace ProductExpr*
MinusExpr → *Expr Whitespace* - *Whitespace ProductExpr*
UnaryMinusExpr → - *Whitespace ProductExpr*
Expr → *ProductExpr* | *PlusExpr* | *MinusExpr* | *UnaryMinusExpr*
Min → \ m i n *TagEnd*
Max → \ m a x *TagEnd*
Goal → *Min* | *Max*
Objective → *Goal Whitespace Expr*
Set → *Interval*
Eq → =
Gt → >
Lt → <
Geq → \ g e q *TagEnd*
Leq → \ l e q *TagEnd*
Neq → \ n e q *TagEnd*

Relation → *Eq* | *Gt* | *Lt* | *Geq* | *Leq* | *Neq*
NewCommand → `\ n e w c o m m a n d {` *WordChars* `} [` *Digit* `] { }` ¤
| `\ n e w c o m m a n d {` *WordChars* `} [` *Digit* `] {` *Whitespace Expr Whitespace* `}` ¤
NewCommands → ε | *NewCommand Whitespace NewCommands*
Constraint → *Expr Whitespace* `\ i n` *TagEnd Whitespace Set*
| *Expr Whitespace Relation Whitespace Expr*
Constraints → *Constraint* | *Constraints Whitespace* `,` *Whitespace Constraint*
OptProbl → *Whitespace NewCommands Objective Whitespace*
| *Whitespace NewCommands Objective Whitespace*
`\ s t` *TagEnd Whitespace Constraints Whitespace*

The two rules marked with the '¤' symbol trigger a parse action called *NewCommandAction*. The action extends the grammar with two new rules, which are then recognized by DynGenPar in any text that follows. When executing the action, DynGenPar passes it the parse tree that was matched by the rule. Out of the final parse tree, this is the subtree with, as root, the left hand side of the rule. In this case, the root is *NewCommand*. The *NewCommandAction* starts by collecting the leaves (i.e., the tokens) below the children of type *WordChars* and *Digit*. The *WordChars* correspond to the **name** of the new command, the *Digit* to its number of arguments **numArgs**. Next, the *NewCommandAction* builds a rule that recognizes the command. If **numArgs** $\neq 0$, the rule is

$$\textit{Command_}\mathbf{name} \to \texttt{\textbackslash}\ \mathbf{name}\ (\texttt{\{}\ \textit{Whitespace Expr Whitespace}\ \texttt{\}})^{\mathbf{numArgs}},$$

where $(\ldots)^{\mathbf{numArgs}}$ denotes repetition, i.e., the braces and their contents are repeated **numArgs** times. If **numArgs** $= 0$, the rule is

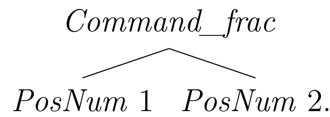
$$\textit{Command_}\mathbf{name} \to \texttt{\textbackslash}\ \mathbf{name}\ \textit{TagEnd}.$$

The second rule built by *NewCommandAction* is of the form

$$\textit{AtomicExpr} \to \textit{Command_}\mathbf{name}.$$

That rule actually adds the new command to the grammar, making it an atomic expression. (Of course, all the other possible expansions for *AtomicExpr* remain valid.)

There are two different rules for *NewCommand*, which both trigger the *NewCommandAction*. The difference is that the second form takes an expression *Expr* within the braces, whereas the first form does not. That expression is not used during the parse process itself. However, after parsing, the `OptProbl` application substitutes the command with the given expression. Any placeholders (*Placeholder* category) in the expression are replaced with the corresponding command argument. For example, given the command definition `\newcommand{frac}[2]{(#1)/(#2)}`, any use of `\frac{2}{3}` is automatically converted to `(2)/(3)`. It shall be noted that, unlike in LaTeX, the type of substitution done is not a text substitution, but a parse tree substitution. This means that, if we define `\frac` as only `#1/#2` (without the parentheses), `OptProbl` will give different results from LaTeX. E.g., for `\frac{1}{1+1}`, `OptProbl` will still respect the priorities and treat it as `1/(1+1)`, whereas LaTeX will expand it to just `1/1+1`, which is 2, not $\frac{1}{2}$. If the expression *Expr* is omitted in the *NewCommand*, e.g., `\newcommand{frac}[2]{}`, the new command is not substituted at all. Instead, it is kept in the final syntax tree as a custom type of node, e.g.,



After completing the parsing, the `OptProbl` application simplifies the parse tree to an abstract syntax tree. The simplification step traverses the parse tree recursively and does the following transformations:

- For the *Var* and *PosNum* categories, the subtrees are removed. Instead, all the leaves (i.e., the character tokens) are collected into a string. The resulting string is attached directly to the *Var* resp. *PosNum* node as its value.
- The same is done for the *Placeholder* category, except that there, the value is stored as an integer rather than a string.
- Nodes of type *Expr*, *PowerExpr*, *ProductExpr*, *AtomicExpr*, *Num*, and *Bracketed-Expr* are removed. Those categories are only needed during parsing and do not carry a semantic meaning. In the syntax tree, those nodes are replaced by their single child, or, in the case of *BracketedExpr*, the second child (i.e., the one that is not a parenthesis token).
- For all categories other than *Letter*, *Digit* and *NumChar*, any of their children that are tokens are completely removed, as the semantics of those tokens are already given by the category. E.g., *NegNum* and *UnaryMinusExpr* already specify that the number or expression is being negated, so the minus sign token is not needed.
- For the same reason, nodes of type *Whitespace*, *Whitespace1*, *TagEnd*, and *TagEndTaboo*, and their entire attached subtrees, are completely removed.
- The same is done to nodes of type *NewCommands* and *NewCommand*, because the command definition does not have any semantic meaning for the final syntax tree. Those categories only serve to trigger the *NewCommandAction*, which adds rules to the grammar during parsing.
- The substitution of user-defined commands described in the previous paragraph is made.
- Any other nodes encountered in the parse tree are copied into the abstract syntax tree.

Finally, the `OptProbl` application exports the abstract syntax tree into the `.dag` file format used by COCONUT to represent optimization problems. The `.dag` format is a textual serialization of COCONUT's internal representation. As indicated by the name, the format represents optimization problems as directed acyclic graphs (DAGs). Basically, every line in the `.dag` file defines a node or an edge of the DAG. The abstract syntax trees produced by the `OptProbl` parser can be converted into such COCONUT DAGs in a straightforward way, which is implemented in the `OptProbl` application.

As an example, consider the following optimization problem:

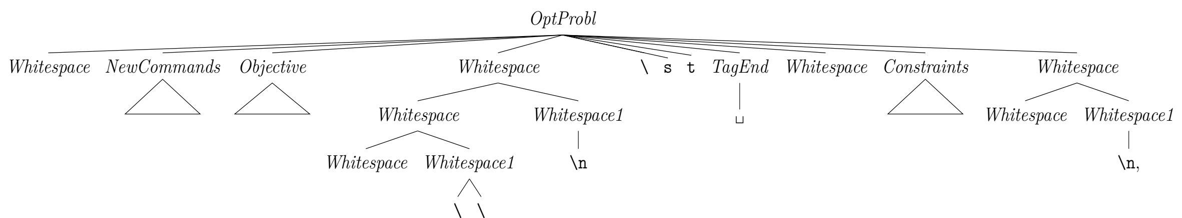
$$\begin{array}{ll} \min & \frac{1}{2}x^2 \\ \text{s.t.} & x \in [-1, 1]. \end{array}$$

(The problem is kept very simple to be able to show the full parse tree.)

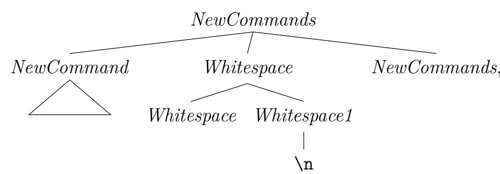
In the OptProbl language, this problem can be specified as follows:

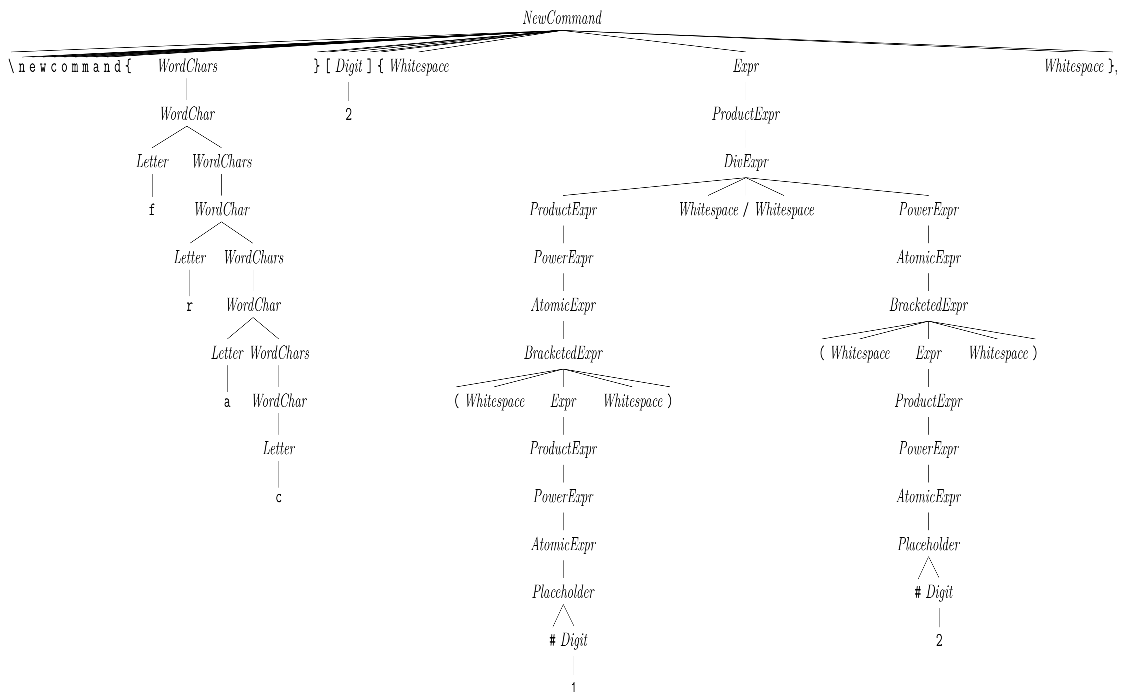
```
\newcommand{frac}[2]{(#1)/(#2)}
\min \frac{1}{2}*x^{2}\\
\st x \in [-1,1]
```

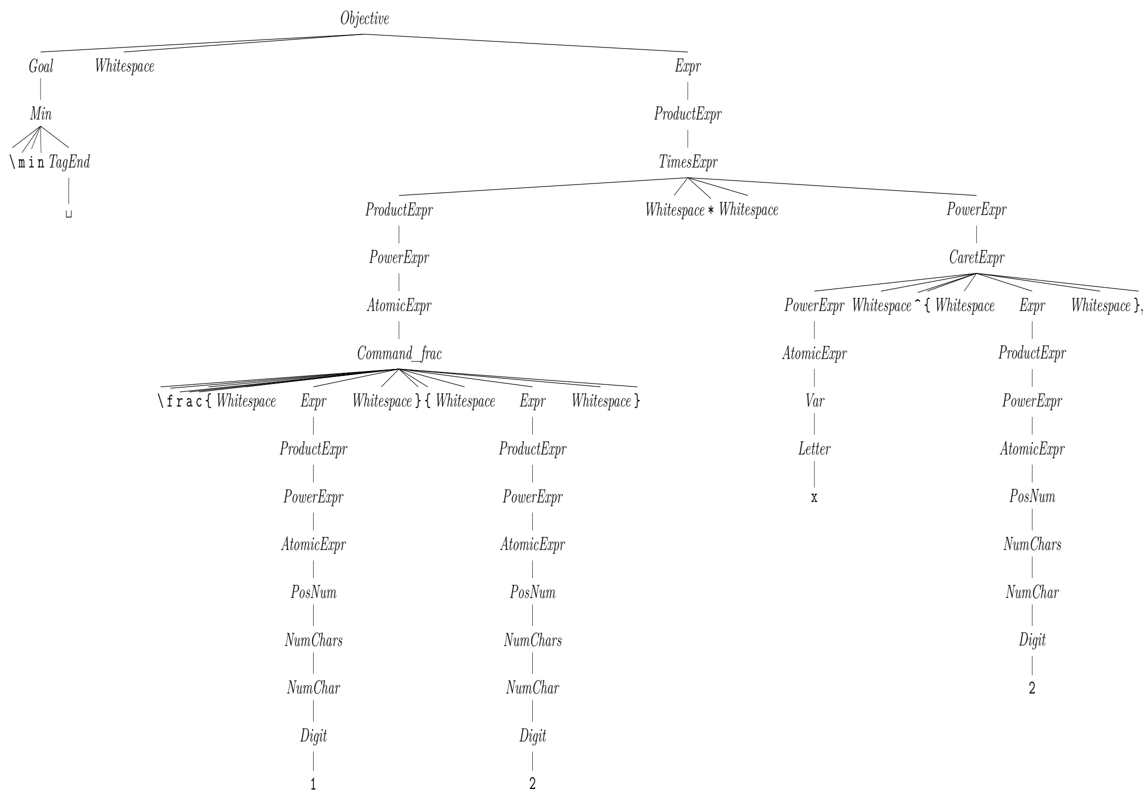
For the above example, the OptProbl grammar produces the following parse tree:



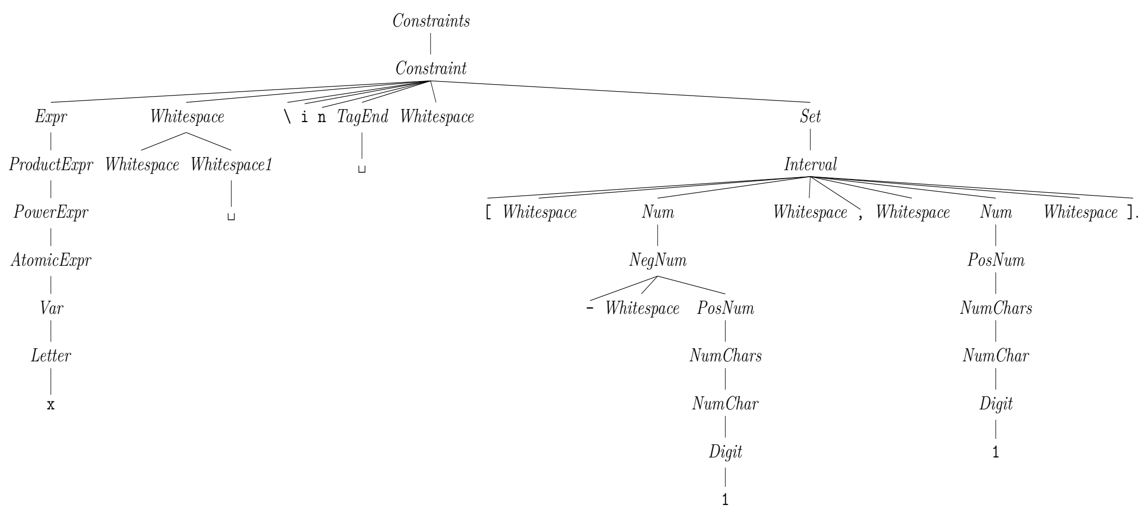
with the subtrees:



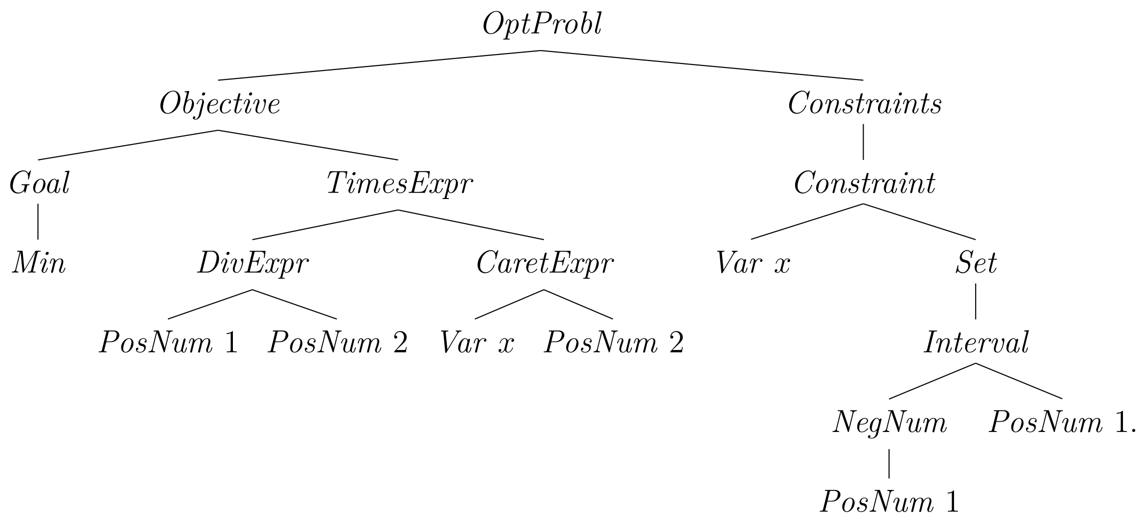




and

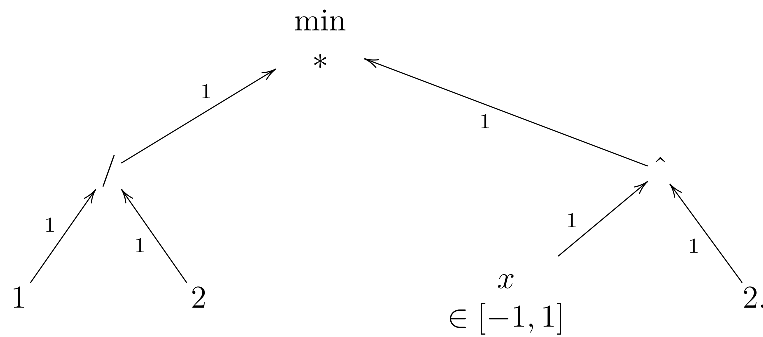


The tree is then simplified into the following syntax tree:

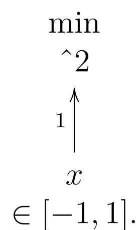


Notice how the *NewCommands* subtree is completely gone, as are the *Whitespace* ones and the syntactic tokens. Also notice how the *Command_frac* was replaced by its definition. Finally, note the simplified expressions.

Finally, the `OptProbl` parser outputs a `.dag` text file representing the following COCONUT DAG:



The edge labels, which are all 1 in this case, are linear multipliers. It is a special feature of COCONUT DAGs that every edge carries such a multiplier. This is used to represent subtractions (as a + node with multipliers 1 and −1) or any other linear combinations. It could also be used to represent the factor $\frac{1}{2}$ in this example, but the `OptProbl` application leaves this kind of representational optimizations to the COCONUT simplifier. The COCONUT simplifier simplifies the above DAG into the equivalent:



4.6 A Grammar for Robust AMPL

This section discusses a DynGenPar grammar for a subset of the **AMPL** (**A Mathematical Programming Language**) (Fourer et al. [27], AMPL Optimization inc. [2]) modeling language for optimization problems. The language was additionally extended to allow intervals wherever AMPL normally expects a number, and the implementation supports reading decimal numbers with outwards rounding, thus we call it **robust AMPL**. The goal is to be as compatible as possible with the official implementation (Fourer et al. [27], AMPL Optimization inc. [2]) of AMPL. Therefore, the accepted subset of the AMPL language is growing over time.

The parser grammar for robust AMPL is implemented as a Concise type sheet `RobustAmpl.cnt`, written by me. The type system described in it is a parser-oriented representation: variables are represented by their names as strings, numbers are represented as arbitrary-precision `BigDecimal` and `BigInteger` types. The semantic analysis resolving the variable references and the rounding of the decimals to either floating-point intervals or simply rounded floating-point numbers (depending on the needs of the application) are done by a later record transformation step. Operations in this representation are just elementary operations corresponding to what is written in the input, e.g., a power is just the operation `x^y` (i.e., x^y), independently of whether x and/or y are constant, integer, etc., and any additional coefficients are represented separately. Calls to well-known functions such as `sin` are represented as a generic `CallExpression`, with the function name as a string. My version of the type sheet can currently parse the AMPL version of the library Lib1 of the COCONUT benchmark (Shcherbina et al. [89], Neumaier & Schichl [72]) that was automatically written by the GAMS Convert (see the section *CONVERT* in McCarl et al. [65]) utility. Handwritten AMPL input, as in the libraries Lib2 and Lib3 of the COCONUT benchmark, poses additional challenges: On one hand, the grammar tries to retain comments in the parsed record, and thus the more liberal use of comments and whitespace in handwritten code can confuse it. On the other hand, handwritten code also makes use of AMPL features such as vectors and matrices that are flattened away in the automatically-generated code in Lib1. The grammar additions for handwritten AMPL are now being worked on by David Langer, but his work is not yet integrated into our system. (In particular, the record transformation described below does not yet support it.)

Internally, the parsed optimization problems are represented in a different type system, given by the type sheet `RobustOptProb.cnt`. That type sheet was originally written by Ferenc Domes. I adapted it to work together with the AMPL grammar. The `RobustOptProb` representation is more semantic and solver-oriented. Variable references are represented by a direct pointer to the semantic memory object for the variable. Numbers are represented as machine types (`Integer`, `Double`, `RealInterval`, `RealIntervalUnion`). Operations in this representation are called **nodes**, because they correspond to nodes in a directed acyclic graph (DAG). The representation of the nodes is optimized for solving, e.g., there are different node types for integer powers, constant noninteger powers, and general powers, and optional coefficients are allowed within a

node, e.g., integer powers can be as general as $(ax + b)^n + c$. Calls to well-known functions such as `sin` are represented as dedicated nodes for each function, e.g., `SinNode`. There is also a distinction between nodes operating on constants (that can be evaluated beforehand) and nodes operating on variables (that have to be passed to the solver). With a few exceptions, the available nodes correspond to the ones used in the Java version of Ferenc Domes's GloptLab (Domes [24, 23]). The type sheet contains output usages that produce AMPL output parsable as `RobustAmpl`, but those usages cannot themselves be used for parsing. This is due to several technical limitations: The parser cannot produce direct pointers to variables, it sees multiple instances of the name and thus produces multiple string objects. In addition, the distinction between named constant coefficients and variables is context-dependent. There are also other ambiguities in the rules. Therefore, I created the `RobustAmpl.cnt` type sheet as a parsable version of `RobustOptProb.cnt`. The conversion from the parser-oriented representation (`RobustAmpl`) to the solver-oriented representation (`RobustOptProb`) is then done by a record transformation.

The record transformation sheet `RobustAmpl.cnrt` (written by me) transforms a record in the type system `RobustAmpl` (as produced by DynGenPar) to a record in the type system `RobustOptProb` (suitable to be passed to a solver). The record transformation does several semantic analysis steps that the parser is unable to do:

- Variable names are resolved. The references are pointed to the actual variable object rather than a copy of the name.
- Constants are converted to the appropriate machine type: `BigInteger` constants that fit into the fixed-size `Integer` machine type are converted to that type. How `BigDecimal` constants (and large `BigInteger` constants) are handled depends on the boolean flag `rigorous`: If rigorous conversion is desired, they are converted to the `RealInterval` type using outwards rounding. Otherwise, they are simply rounded to the `Double` type.
- For arithmetic operations, the record transformation determines whether they operate on at least one variable or only on constants, and uses the corresponding operation nodes in the `RobustOptProb` type sheet accordingly.
- Well-known function names such as `sin` are also recognized and converted to the appropriate operation nodes.

Since the `RobustOptProb` type system uses more general operations than the basic arithmetic used by `RobustAmpl`, the output of the transformation is not necessarily the most efficient representation. (In particular, the round trip from `RobustOptProb` to AMPL text to `RobustAmpl` and back to `RobustOptProb` will typically produce a more complex representation.) This limitation was accepted by design to keep the record transformation simple. To alleviate those issues and perform further simplifications, I implemented a separate simplifier as another record transformation.

The record transformation sheet `RobustOptProbSimplifier.cnrt` (written by me) transforms a record in the type system `RobustOptProb` to another record in the type system `RobustOptProb`. In the output record, expressions are simplified where possible. Anything that cannot be simplified is simply copied. The goals of the simplification are

twofold: On one hand, trivial operations such as multiplication by 1 are eliminated entirely. On the other hand, patterns are matched to group multiple elementary operations to more complex operations such as `UQuadNode` (a node representing a general univariate quadratic, of the form $ax^2 + bx + c$, where x is a variable and a, b, and c are constant coefficients) wherever possible. The following simplifications are currently performed:

- simplifying sums or products containing of only one element to just that element,
- simplifying sums or products containing multiple constants by replacing them with their constant sum resp. product,
- eliminating constant 1 terms from coefficient products and constant 0 terms from coefficient sums,
- converting powers by the integer 2 to a `UQuadNode`,
- simplifying a constant times a `UQuadNode` by pulling the constant factor into the `UQuadNode`, i.e., $k(ax^2 + bx + c) = (ka)x^2 + (kb)x + (kc)$,
- simplifying quadratic sums (i.e., sums of `UQuadNode`s in a given variable x, linear terms in the same variable x, and constants) to a single `UQuadNode` $ax^2 + bx + c$.

Any arithmetic operations that come up during the simplification are performed on the data types that the operands have. In particular, if one or both of the operands is an interval (even a point interval), an interval operation with outward rounding is performed, producing an interval as the result. If the operands are approximate floating-point (`Double`) constants, the operation is performed with simple rounding and the result is again a `Double`. This ensures that models read with rigorous input enabled, where all non-integer coefficients are intervals, are treated rigorously, whereas approximate models remain approximate.

It shall be noted that several of the above simplification steps can uncover new opportunities for simplification. E.g., a `UQuadNode` is built in several steps: first, the second power is detected and a `UQuadNode` is built from it, then the constant coefficient is pulled into the `UQuadNode`, yielding a `UQuadNode` for the quadratic term, and finally, that term and the linear and constant terms are grouped into a single `UQuadNode`. The elimination of trivial sums and products can also uncover new simplification opportunities by rendering further operations trivial. Therefore, it is often beneficial to run multiple iterations of the simplifier. Typical examples take around three iterations to fully simplify.

Chapter 5

DynGenPar and the Grammatical Framework (GF)

This chapter discusses the interoperability between DynGenPar and the **Grammatical Framework** (**GF**) (Ranta [77, 78], Ranta et al. [79]). GF is a state-of-the-art framework for grammar-driven natural language processing. It comes with repositories of linguistic and morphological information, called **resource grammars**. Hence, using the framework, it is easy to parse – with some restrictions – natural language text and to produce grammatically correct natural language output. Therefore, in the FMathL project, it was envisioned to leverage the information contained in a GF resource grammar in one way or another. This chapter documents the outputs of that research.

DynGenPar is interoperable with GF to the point where, in many cases, DynGenPar can be used as a drop-in replacement for either of the GF runtimes. I.e., it is possible to first use the GF compiler to compile the grammar to the runtime representation, then use DynGenPar to parse input using the compiled grammar. Even though DynGenPar was not optimized for this use case, the performance is within an order of magnitude of both available GF runtimes (see Section 7.1.2). A drawback is that using DynGenPar in this way does not allow leveraging the full feature set of DynGenPar, in particular, dynamic rule addition at runtime.

The first section summarizes the first attempt at interoperating with GF in the FMathL project, by developing C bindings to its Haskell code. The second section presents how compiled GF grammars in the PGF (Portable Grammar Format) file format can be imported by DynGenPar. The next section describes a lexer compatible with GF, implemented as a DynGenPar token source. The next section presents a GUI application demonstrating the prediction functionality of DynGenPar on PGF grammars. The last section documents a proof of concept for a GF application grammar for a tiny subset of natural mathematical language.

5.1 C Bindings for the GF Haskell Runtime

This section briefly summarizes the historical C bindings that I had developed for the Haskell runtime of the Grammatical Framework (GF) (Ranta [77, 78], Ranta et al. [79]).

In 2008-2009, our work group had the problem that we wanted to interface with the grammatical knowledge encoded in the GF resource grammars, but, at the time, the only way to use that knowledge was the GF Haskell runtime. GF grammars are compiled into **PGF** (**Portable Grammar Format**) (Angelov et al. [4]) grammar files. The GF runtime is a library that loads such PGF files and uses them to parse or linearize text. At the time, there was only a Haskell implementation.

Therefore, I implemented a C binding for the GF Haskell runtime using the Haskell foreign function interface (FFI). The typical use of the FFI is to use C code from Haskell. However, it can also be used in the opposite direction, to use Haskell code from C, which is what my bindings do. Those bindings were the only convenient way to use PGF files from C or C++ code at the time.

The Haskell functions needed to parse and linearize text using a PGF grammar are wrapped with a C interface. Haskell objects are represented in C using the `HsStablePtr` type, which corresponds to the `StablePtr` Haskell type. Lists are represented as arrays terminated by a null pointer. Convenience functions to free an entire list are implemented in C.

In 2010, the Haskell implementation of the GF lexer was moved from the runtime library to the command-line interface. Therefore, for the C bindings, to avoid requiring the entire GF command-line interface, I reimplemented the lexer in C.

These C bindings were included with several GF releases. They were moved to a separate `gf-contrib` repository in 2013. The C bindings also served as the base for the GF Python bindings.

These C bindings for the GF Haskell runtime have since been obsoleted by the new GF C runtime, which is now implemented directly in C, making it unnecessary to bind the Haskell implementation. On the other hand, in our work group, the idea of using either GF runtime directly has been abandoned in favor of the GF interoperability in DynGenPar described in the next sections.

5.2 PGF (Portable Grammar Format) File Import

The DynGenPar implementation can import **PGF** (**Portable Grammar Format**) (Angelov et al. [4]) grammar files produced by the Grammatical Framework (GF). This section describes how this is achieved.

The PGF file format is a binary format based on parallel multiple context-free grammars (PMCFGs) (Seki et al. [88]). Therefore, the support for PGF files in DynGenPar builds upon the parser's support for PMCFGs (see Section 2.2.3.6). The import is handled

by converting the PGF file to PMCFG standard form. A few extensions (Kofler & Neumaier [56]) to the PMCFG standard form, supported by DynGenPar, are required for some GF features:

- Additional context-free rules can be given, the left-hand sides of which can be used as “tokens” in the expression of PMCFG functions.
- Next token constraints (see Section 2.2.3.7) can be used. This and the previous extension are required to support GF’s rules for selecting, e.g., “a” vs. “an”.
- PMCFG functions can be given a token (or a context-free nonterminal as above) as a parameter, in which case the syntax tree will reproduce the parse tree of that symbol verbatim, including attached data, if any. This extension is required to support GF’s builtin `String`, `Int` and `Float` types.

Given DynGenPar’s existing support for context-free grammars and next token constraints, it was straightforward to implement these extensions.

A PGF file is a binary serialization of the Haskell data structures used by the GF compiler and the GF Haskell runtime, based on the `Data.Binary` Haskell library. Unfortunately, the binary serialization produced by Haskell is very different from the one used by Qt’s `QDataStream` class. Therefore, PGF files cannot be read with `QDataStream`. Instead, I implemented a `HaskellDataStream` class with an interface imitating `QDataStream`. Like `QDataStream`, `HaskellDataStream` overloads the `>>` operator with functions that read bytes from the underlying `QIODevice`, decode them, and store the result into the right-hand operand. Overloads of the `operator>>` are provided not only for the basic `Data.Binary` types, but also for the PGF structures as serialized by the `PGF.Binary` module. The serialization of those structures builds upon the serialization of the primitive types, both in the Haskell implementation and in DynGenPar’s `HaskellDataStream` class. The `HaskellDataStream` class is not part of the public DynGenPar API, i.e., users of DynGenPar are not expected to access this class directly. It is, instead, designed to be used in the implementation of the `Pgf` class, through which DynGenPar users can make use of its functionality.

The core of the PGF file import code in DynGenPar is the `Pgf` class. It is part of the public DynGenPar API, i.e., it is exported directly to programs using DynGenPar. The `Pgf` class is, in fact, the one class which gives DynGenPar users access to the PGF import functionality. The `Pgf` class represents the information contained in PGF files in a format that can be processed by DynGenPar. The constructor takes the name of a PGF file to load, and optionally the name of the concrete grammar to import from the PGF file. One single PGF file can contain multiple concrete grammars for the same abstract grammar. DynGenPar can only work with one concrete grammar at a time. Therefore, if the PGF file contains more than one concrete grammar, the name of the concrete grammar to use is required. (If no concrete grammar name is specified and if the PGF file contains only one concrete grammar, that concrete grammar is loaded. If no concrete grammar name is specified and if the PGF file contains multiple concrete grammars, a fatal error is raised. A fatal error is also raised if the specified concrete grammar name is not present in the PGF file.) The `Pgf` class constructor reads the data from the PGF file using the `HaskellDataStream` class described above and stores it in its class member variables.

The `Pgf` class has the following data members:

- `pmcfg`: the PMCFG in its standard form, with the extensions given at the beginning of this section. The `Pmcfg` structure, the type of this variable, also contains the start category. Grammar symbols (tokens and categories) and functions are identified by integers.
- `catNames`: the names of the grammar symbols, i.e., the tokens and the (nonterminal) categories, as a `QStringList`. Those names are in general not unique, because a PGF file can contain multiple copies of the same category with different attribute values. (See Section 2.2.3.6 for how attributed grammars are handled by the GF compiler.)
- `functionNames`: the names of the PMCFG functions as a `QStringList`. Those names are in general not unique either, because GF supports function overloading.
- `tokenHash`: a hash table mapping the token strings to their integer identifiers, for quick lexing of the tokens.
- `suffixes`: the list of token suffixes with their integer IDs. GF supports a special symbol called `Prelude.BIND`. (Up to GF version 3.5, it was a token with the special value `"&+"`. As of GF version 3.6, it is now a dedicated type of symbol.) When linearizing, i.e., when producing text using the grammar, this token pastes the previous and the succeeding token together. During parsing, the lexer needs to split the token at those points. Therefore, the PGF file import code collects a list of all the tokens that follow `Prelude.BIND` in the grammar to help the lexer do that.
- `componentNames`: a hash table mapping each category name to the list of names of its components. A PMCFG category can consist of multiple components. In GF, those components have names. The mapping is from the name of the category rather than from its integer ID because, when there are multiple variants of the same category for different attributes, those all have the same components. Those multiple category variants have the same name, but different IDs.
- `firstFunction`: the ID (an integer) of the first true function. The PGF import creates several synthetic PMCFG functions called **coercion functions**, which are basically only typecasts. They are given the lowest IDs. This variable stores the ID of the first function that is not such a coercion function.

The constructor of the `Pgf` class performs the following steps:

1. It deserializes the contents of the PGF file using the `HaskellDataStream` class described above. I.e., it reads those file contents into data structures that closely match the on-disk representation.
2. It picks the requested concrete grammar out of the list of concrete grammars contained in the PGF file.
3. It identifies the start category and stores it in the `Pmcfg` structure.
4. It converts the list of category names `catNames` and the lists of their component names `componentNames`.

5. It adds the special token-like symbols of GF to the set of tokens in the `Pmcfg` structure. Those are the GF symbols `Var`, `Float`, `Int`, and `String`, which are treated as tokens carrying a value in DynGenPar.

6. It converts the sequences, i.e., the right hand sides of PMCFG functions, to DynGenPar sequences. This is done separately from converting the functions and rules because PGF files store the sequences separately and reference them by their number in the functions. Any tokens that are encountered are added to the set of tokens in the `Pmcfg` structure and to the `tokenHash` hash table. The `suffixes` list (see above) is also created in this step. Some PGF files contain tokens that end in a dot, which complicates lexing. Therefore, if a token ends with a dot, the dot is split off into a separate token.

   A particularly complex item that can be contained in a PGF sequence is the GF `pre` construct. The `pre` stands for "**pre**fix matching". The construct lists a list of tokens, from which only one is valid, depending on the beginning (prefix) of the token that follows. If several prefixes match, the first that matches is selected. E.g., the English indefinite article is defined (in `ResEng.gf` in the GF source code) as:

```
pre {
  "eu" | "Eu" | "uni" | "up" => "a" ;
  "un" => "an" ;
  "a" | "e" | "i" | "o" | "A" | "E" | "I" | "O" => "an" ;
  "SMS" | "sms" => "an"
  _ => "a"
}
```

   i.e., if the word starts with "eu…", "Eu…", "uni…", or "up…", use "a", otherwise, if it starts with "un…", use "an", otherwise, if it starts with a vowel other than "u…" or with "SMS…", use "an", otherwise, use "a". These constructs are represented in DynGenPar as context-free categories. As an extension to standard PMCFGs, those context-free categories are treated like tokens in the PMCFG. For every unique alternative (e.g., in the above example, "a" and "an"), a context-free rule for the category producing the alternative token is generated. The rule is constrained by a placeholder next token constraint of type "expect" with an empty list of accepted next tokens. In addition, the lists of prefixes on the left and the corresponding unique alternative are remembered so that the next step can add the correct next token constraints to the rules. Note that the order of the prefix lists must be preserved, and thus there can be multiple prefix lists for the same unique alternative, as in the example above.

7. It adds next token constraints enforcing the above prefix matches. This has to be done in a separate step because it requires the full set of tokens, which is only known after importing all the sequences. The import process goes through all the tokens, matches them against the lists of prefixes, picks the first that matches, and adds the token to the corresponding rule's list of expected tokens. As a special case, tokens starting with the `$` or `` ` `` character, which represent formulas, are allowed after any article. Thus, they are added to each prefix match rule's list of expected tokens.

8. It generates the **coercion functions** for each category. Those are synthetic functions that define the category. The coercion function for a category specifies the number of components of the category. It is otherwise simply an identity function. Coercion functions are used in the PMCFG like typecasts, thus the name. The PGF files contain a special function for each category called its “lindef” (**lin**earization **def**ault definition). The intended purpose of those functions is not relevant for DynGenPar. However, the “lindef” functions normally reflect the number of components of the category. This property implies that the coercion functions needed by DynGenPar can be generated from them. (The original “lindef” functions stored in the PMCFG are not needed after generating the coercion functions, and are thus discarded, replaced by the coercion functions.) The coercion functions are saved in the `Pmcfg` structure, and their names (“coerce ” + the name of the category) stored in the `functionNames` list.
9. It sets the `firstFunction` variable indicating the first true function.
10. It converts the true functions, i.e., those that are not coercion functions, to DynGenPar PMCFG functions. This is done in a separate step because those functions do not need the special treatment that coercion functions need. It is also done separately from converting the rules because PGF files store the functions separately and reference them by their number in the rules. Like the coercion functions, the imported functions are also saved in the `Pmcfg` structure, and their names are also stored in the `functionNames` list.
11. Finally, it imports the production rules. PMCFG files contain two different types of productions. One of the types is a standard PMCFG rule, i.e., a call of a PMCFG function. This type of rule is straightforward to convert to a DynGenPar PMCFG rule. The other type of PGF productions is a coercion to a new variant of the same category. That mechanism is used to avoid having to copy rules that accept multiple variants of the same category. Instead of doing those copies, GF groups the accepted variants of the category into a single coerced variant. That type of production is converted to a call of the coercion function for the category. In addition, since GF does not store a name for those coerced categories in the PGF file, a name (category name + “(coerced)”) is automatically generated. In both cases, the converted rule is saved in the `Pmcfg` structure.

The second class in DynGenPar’s public PGF API is the `PgfParser` class. `PgfParser` is a subclass, operating on a `Pgf` object, of the DynGenPar `Parser` class. There is also a convenience constructor that takes a file name and, optionally, a concrete grammar name, and automatically constructs a `Pgf` object from those. Parsing with PGF grammars requires a special lexer, which will be described in the next section. That lexer needs access to the `Pgf` object, because that object contains the sets of tokens and token suffixes. Therefore, the lexer is fully controlled by the `PgfParser` class and hidden from the public API, ensuring that they always both work on the same `Pgf` object. Thus, the `PgfParser` class provides methods to set a different input source (e.g., `setInputFile`), which are forwarded to the token source, i.e., the lexer. There are also `catName` and `functionName` methods, which retrieve the name of, respectively, a grammar symbol (i.e., a token or a category) or a function from their integer ID. The

catName method also supports category components, producing names of the form "*categoryName*[*componentName*]". The last method provided by the PgfParser class is filterCoercionsFromSyntaxTree. That method operates on the PMCFG syntax tree produced by DynGenPar's parseTreeToPmcfgSyntaxTree utility function. In the case of PGF grammars, that syntax tree can contain calls to coercion functions, which are typically not wanted in the final output. Therefore, the filterCoercionsFromSyntaxTree method traverses the syntax tree recursively and removes all calls to coercion functions, replacing them with their (unique) child node. (There is always exactly one child node because a coercion function is simply an identity function.)

In order to test the PGF support, I wrote a simple test program, pgftest. The test program takes the name of a PGF file as a command line argument, and either an input file name as the second command line argument, or input from stdin. It assumes that the PGF file contains exactly one concrete grammar. The pgftest program imports and parses the PGF file using the PgfParser class. If parsing succeeds, it prints the parse tree to stdout, runs parseTreeToPmcfgSyntaxTree on it, prints the resulting PMCFG syntax tree to stdout, runs PgfParser::filterCoercionsFromSyntaxTree on that syntax tree, and prints the resulting filtered syntax tree to stdout. (The intermediate trees are output for demonstration and debugging purposes.) If a parse error is encountered, the unexpected token and its position are printed to stderr. If the parsing completes without hitting an error, but no syntax trees are produced, this means that the input was incomplete. In that case, the pgftest program uses the DynGenPar prediction API to output the valid continuations for the incomplete input to stderr.

I also ported the pgftest test program to Java using the DynGenPar Java bindings (see Section 2.2.1 and Section 3.2). The Java code is exactly equivalent to the C++ code. The ported PgfTestJava serves as the main test for the DynGenPar Java bindings outside of the Concise framework (see Chapter 3).

5.3 PgfTokenSource – A GF-compatible Lexer

This section presents the **PgfTokenSource**, a lexer compatible with PGF grammars. It describes how the lexer works, the peculiarities it needs to take care of, and the limitations it currently has.

GF grammars are token-based, as opposed to scannerless. This is of course reflected in the compiled PGF files. Therefore, parsing from PGF grammars requires a compatible lexer (scanner). Thus, I implemented a GF-compatible lexer as a DynGenPar token source, called **PgfTokenSource**. The token source was implemented more as a necessity than as a research focus, designed for simplicity rather than perfection.

One important challenge is that, in GF, parsing is not completely separated from lexing. Most importantly, the set of tokens that the lexer should be able to match is implicitly given by the PGF file: The tokens appear as strings in the PGF rules. This means that the lexer needs access to the Pgf grammar. Therefore, I decided to let the PgfParser class own the PgfTokenSource, ensuring that they always both work on the same Pgf

object. Thus, users of DynGenPar are expected to use the `PgfTokenSource` class only through the `PgfParser` class that owns it, never directly, i.e., the `PgfTokenSource` class is *not* part of the public DynGenPar API.

The main method of the `PgfTokenSource` is the `readToken` method. This is a virtual method that all DynGenPar token sources must implement and that returns the next token from the input. As in most token sources, the `readToken` method is where most of the work happens.

The `PgfTokenSource` is a **stateful lexer**, meaning that the operation of the lexer depends on the preceding context. In other words, this means that the `readToken` method saves some information from one invocation to the next. That information forms the **lexer state**. It consists of:

- the character position in the input stream,
- the boolean flags `inFormula` and `pastFormula`, which represent lexer states in the traditional sense, and which are used when parsing formulas, and
- a list of `suffixes`, which are tokens that have already been matched and that will be returned before consuming any further input.

The `PgfTokenSource` implements the virtual `saveState` and `rewindTo` methods, which allow rewinding to a previous position in the text with the correct lexer state, i.e., without confusing the lexer.

The first thing the `readToken` method does is to check whether there are any pending tokens in the `suffixes` list. (Where those tokens come from will be explained below.) In that case, it removes the first token from the `suffixes` list and returns it.

Next, it checks whether the input stream is completely consumed. In that case, it returns the ε token. In DynGenPar, that is the token with the ID 0. It is how a token source signals the end of the input. This check is typically the first thing a `readToken` method does. However, in the `PgfTokenSource`, the `suffixes` must be checked first, because they are already matched from previous stream input and thus the input only counts as consumed when the `suffixes` are consumed.

What happens next depends on the parser state. Let us first consider the default state, where both the `inFormula` and `pastFormula` flags are false. The basic principle is very simple: The token source matches a token, which is either a number, or a word, or a symbol. The following characters are considered **alphabetic characters**:

- letters (i.e. ‘A’ to ‘Z’ and ‘a’ to ‘z’),
- the ‘_’, ‘'’ and ‘\’ characters, and
- for simplicity, all non-ASCII UTF-8 characters, i.e., all bytes with value $\geq$ 128. Technically, the Unicode codepoints should be decoded and a Unicode character classification table used, but in practice, it is typically acceptable to just treat them all as letters. Most non-ASCII Unicode codepoints are, in fact, letters. The ones encountered in practice are typically either Latin letters with diacritics or non-Latin letters.

A **lexeme** is a character sequence that is recognized by the lexer as a token. The `PgfTokenSource` applies the following rules:

- Lexemes are maximally matched, i.e., as long as characters matching the criteria are available in the input, they are always (with one exception, given below) added to the lexeme.
- Any lexeme that starts with an alphabetic character and contains only alphabetic characters, dashes ('-'), and digits ('0' to '9') is a word.
- Any lexeme that starts with a digit and contains only digits and the characters '-', '.', and 'e' is a number. (Special care has to be taken for the '.', which can also be a sentence period. Therefore, as an exception to the maximal matching rule, the `PgfTokenSource` accepts a '.' only in the middle of a number, not at the end.)
- Lexemes starting with a '-' character are either words or numbers depending on the first character that is not a '-'. They are then processed with the same rules as other words resp. numbers.
- Whitespace characters (i.e., characters for which `std::isspace` returns true) end lexemes, but are otherwise ignored.
- Any other encountered character is treated as a symbol. Symbol characters are always lexemes by themselves: They end the previous lexeme, and they do not accept any further characters.

The above rules are not perfect. E.g., the '-' character is accepted in the middle of a number in order to accept valid numbers like "1e-3", but that simple rule also incorrectly treats "1-3" as one number. However, they have been good enough in practice so far.

Once the `PgfTokenSource` has identified the lexeme, i.e., the character sequence forming the token, it converts it to the token ID to return. There are three special GF tokens that accept (and in fact require) a value of the corresponding data type: `String`, `Int`, and `Float`. (There is actually a fourth one, `Var`, which is currently not supported by the `PgfTokenSource`.) The other tokens are determined by the PGF file and correspond to a fixed character sequence. Therefore, they do not accept a value. One difficulty is that the same character sequence can be interpreted in more than one way. And in fact, different PGF grammars can expect a different interpretation of the same character sequence. This occurs most often with the `String` value token, because basically any character sequence can be returned as a `String`. But there are also grammars that want to parse integers as words (i.e., as fixed-character-sequence tokens), whereas others expect them to be returned as `Int` value tokens. And of course an integer could also be a `String`. Therefore, it is often tricky to decide what type of token to return. In GF, the lexer and the parser are tightly integrated. In DynGenPar, the `PgfTokenSource` would ideally call into the `PgfParser` and use the prediction information to (help) make the decision. This would be possible without changing the API, because the `PgfTokenSource` is already internal to the `PgfParser`. However, this feature is not currently implemented. Instead, the `PgfTokenSource` relies on the following heuristic rules:

- Any character sequence identified as a number by the rules in the previous paragraph is returned as a number: If the sequence contains the characters '.' or 'e', it is

returned as a `Float` value token. If it does not contain any of those characters, it is returned as an `Int` value token. The actual numeric value is attached as the value of the token. If the character sequence fails to parse as a number (of the expected type, i.e., `double` resp. `int`), an error token is returned instead. The error token will trigger a parse error in the parser because it is intentionally not in the grammar.

- Any other sequence, i.e., one identified as either a word or a symbol, is treated as a word. The `PgfTokenSource` first tries to match those as tokens using the `tokenHash` hash table created in the `Pgf` structure during the PGF file import. It first tries to match the character sequence exactly as in the input, then with the first character converted to lower case. This is also a limitation, in two ways:

  1. If the version with uppercase first character matches, the lowercase variant is not even tried. Thus, if, e.g., both the German verb "liebe" and the German noun "Liebe" are in the grammar, a sentence starting with the verb "liebe" (capitalized as "Liebe"), e.g., "Liebe deinen Nächsten!", will not be interpreted correctly. This problem is not as frequent in English, where only proper nouns are capitalized, but one English-language example would be "Turkey steaks taste delicious." (A workaround is to lowercase all the words in the grammar, but that then produces an overgenerating grammar, i.e., one that accepts incorrect sentences and may have bogus ambiguities. The workaround also relies on the second limitation.)
  2. The lowercasing is currently tried independently of the context, even where capitalization does not make sense grammatically.

  If there is no match, the `PgfTokenSource` tries to match suffixes. As explained in the previous section, GF supports a special symbol called `Prelude.BIND`, which pastes the previous and the succeeding token together. The PGF import creates a list called `suffixes` of all the tokens that follow `Prelude.BIND` in a rule and stores it in the `Pgf` structure. (The implied assumption is that `Prelude.BIND` is actually followed by an explicit token. In practice, this is often, but not always the case. Other uses of `Prelude.BIND` are not supported. The `Prelude.BIND` symbol is then ignored, and the rule will only match if whitespace is added in the input to separate the tokens.) The `PgfTokenSource` tries to match each of those suffixes, i.e., check whether the word ends with the suffix. If the suffix matches, the remaining word is checked against the `tokenHash` table. If there is still no match, the process is repeated until either the remainder is accepted as a token or no further suffix matches. Once there is a match, the `PgfTokenSource` records the matched suffixes in its own `suffixes` list and returns the remaining token. That `suffixes` list is what is checked for pending tokens at the very beginning of `readToken`. It will result in returning the matched suffix(es) in the next invocation(s) before lexing any further input. It shall be noted that the suffix matching is subject to the same limitation as the lowercasing: If a word is valid as a token, it will not be considered for suffix matching at all.

  Another special rule is used in order to allow incrementally parsing suffixes: If the remaining word after matching suffixes is empty, the first suffix is returned as the token. This bears the limitations that a lone suffix is falsely accepted as a valid

word (but only if it is not also a valid word in the grammar, in which case the word is preferred to the suffix), and that incrementally parsing a suffix is only possible if the suffix is not also a valid word in the grammar.

If there is still no match after attempting to match suffixes, the word or symbol is returned as a `String` value token, with the identified character sequence as its value. This is done unconditionally because, if the grammar does not accept a `String` at this location, it will just cause a parse error. Returning an error token would have the exact same effect, except for the unexpected token in the error message being different. There are, however, some practical limitations of this way to handle `String`:

1. Since prediction information is not used at this time, a token is only returned as a `String` if it does not match any other token type, even where the grammar accepts only `String`. Therefore, it is not possible to use any word in the grammar as a `String` (nor even a suffix, because lone suffixes are accepted for incremental parsing support), nor a number. E.g., the GF example grammar `mathtext` expects a `String` as the theorem identifier. The `PgfTokenSource` does not accept a number for that `String`. Thus, theorems can only be numbered with identifiers like "T1".
2. The `PgfTokenSource` has no way to know whether to accept any suffixes (and return the rest of the string separately from the suffix tokens) or whether to return the whole string as is. It currently always does the latter. This is not always what is wanted, especially considering that the "`String` `Prelude.BIND` *token*" sequence is a good reason to use `Prelude.BIND`.
3. A `String` stops at the first symbol character. This prevents recognizing, e.g., a formula as one `String` token. As this functionality was required for the grammars the FMathL team experimented with, I implemented a special workaround for formulas, described below.

- A very special symbol is the '$' symbol. That symbol is treated as a delimiter for formulas. If the '$' symbol is encountered, two things happen: On one hand, it is processed like any other symbol (or word), i.e., as in the previous case. In particular, if the '$' token appears in the grammar, this results in the corresponding token ID being found in the `tokenHash` table and returned. On the other hand, the `inFormula` flag is set, putting the lexer into a special state, described below.

The first non-default lexer state is the state where `inFormula` is true. In that state, the `PgfTokenSource` simply collects all the characters it encounters, until it encounters an unescaped '$' sign. The backslash is treated as an escaping character, i.e., the character immediately following the backslash loses its special meaning. This means that the sequences "\\" and "\$" are treated as normal characters. The escaping backslash is also retained. The `readToken` method then returns a `String` value token. The value is the whole formula, i.e., the entire captured character sequence, not including the terminating '$' symbol. In addition, the `inFormula` flag is cleared, and the `pastFormula` flag is set. This implementation is what was needed for the FMathL project's own grammars. It is, though, also a limitation, because some GF grammars want to parse formulas in a more detailed way, which is not supported by the `PgfTokenSource`. E.g., in the `mathtext`

grammar, only variable names work as formulas, expressions like $x \in S$ are not lexed correctly.

The last lexer state is the state where `pastFormula` is true. (This also implies that `inFormula` is false. The algorithm does not allow both flags to become true at the same time.) That state is almost identical to the default state, but the next '\$' symbol only unsets the `pastFormula` flag, it does *not* set the `inFormula` flag. This is needed because the same symbol '\$' marks both the beginning and the end of a LaTeX formula. The first encountered character should always be '\$', but with incremental parsing, it can also be something else. In that case, the first encountered alphabetic character results in unsetting the `pastFormula` flag.

This algorithm results in a lexer that is sufficient in practice for many applications of PGF grammars, subject to the aforementioned limitations. In particular, the current version of the `PgfTokenSource` is sufficient for what was needed in the FMathL project. Some improvements are envisioned for the future, but given the time constraints of the project and the different focus of the DynGenPar development, I have not been able to pursue them at this time:

- The lexer could be given a true understanding of Unicode, rather than operating on UTF-8 bytes as it does now. The official Unicode character classification tables could then be used.
- The lexing of numbers could be done in a more precise way, rejecting some invalid numbers (e.g., "1-3"), that are currently accepted. This would result in lexing, e.g., the subtraction $1 - 3$ correctly as three separate tokens.
- Prediction information from the parser could be used to determine what tokens make sense to return in any given context. This would allow lifting several of the current ambiguities that currently require workarounds or heuristics, or that currently result in lexing not working at all.
- For the ambiguity between capitalized and all-lowercase versions of a word at the beginning of a sentence, context-free rules could be used to replace the two tokens, e.g., "turkey" and "Turkey", with three tokens (i.e., in the example, "turkey", "Turkey", and "[tT]urkey"). The third token would be the one returned when the case cannot be determined because the context forces capitalization. The context-free rules would then be of the form:

  $$c_turkey \to \text{turkey} \mid \text{[tT]urkey},$$
  $$c_Turkey \to \text{Turkey} \mid \text{[tT]urkey},$$

  and the synthetic categories *c_turkey* and *c_Turkey* would be used instead of the tokens "turkey" resp. "Turkey". This would also require changes in the PGF import code. (Alternatively, one could consider simply adding rules of the form:

  $$\text{turkey} \to \text{[tT]urkey},$$
  $$\text{Turkey} \to \text{[tT]urkey},$$

  at runtime, but that would require changing the DynGenPar implementation to allow symbols that are both tokens and nonterminal categories.)

5.4 PGF GUI – Graphical Demo Application for Prediction on PGF Grammars

This section describes the DynGenPar **PGF GUI**, a GUI application that demonstrates both the PGF (Portable Grammar Format) file import and the prediction abilities of DynGenPar. The proof-of-concept user interface lets the user input text conforming to a GF (Grammatical Framework) grammar by interactively selecting from the list of possible next tokens allowed by the grammar. This shows how autocompletion could work based on the prediction functionality in DynGenPar, though it does not allow entering freeform text. True autocompletion allowing freeform text entry was implemented later in the Concise parsable file views (see Section 3.3.5).

The PGF GUI is a GUI frontend for the GF interoperability in DynGenPar. The user interface is based on the libraries Qt 4 (Qt Company [74]) and kdelibs (KDE libraries) 4 (KDE Webmasters [44]). In addition, the application uses DynGenPar. In particular, it uses the classes operating on the PGF grammar files generated by GF (see Section 5.2) and the prediction API that enumerates the possible continuations for the input (see Section 2.2.3.1).

To use the PGF GUI application, one first needs to select a PGF file containing a compiled GF grammar. The application starts up with an example PGF file (a simple test grammar) loaded by default. It is possible to select another PGF grammar using the *File / Open...* menu entry. The application currently accepts only PGF files containing **exactly one** concrete grammar. A PGF file can contain multiple concrete grammars for the same abstract grammar; this is currently not supported and will result in the PGF GUI terminating with a fatal error. (This limitation exists because the PGF GUI is only a simplified proof of concept. A production application would allow the user to select one of the contained concrete grammars.)

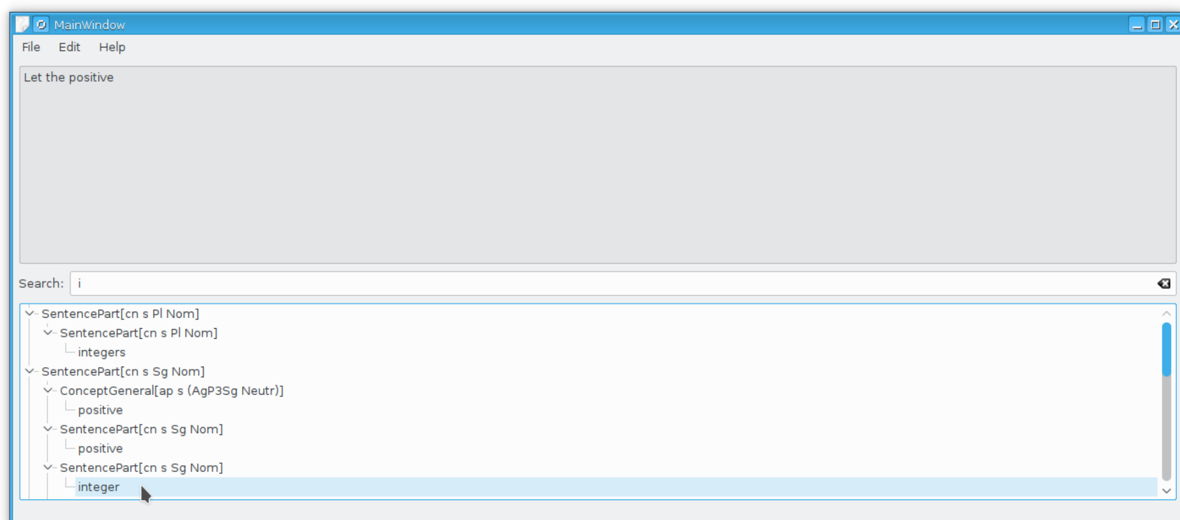


Figure 5.1: Screenshot of the PGF GUI in text entry mode

The main window (Figure 5.1) is divided into two vertical halves. The upper half is a read-only view of the text that has been input so far. Initially, it is empty. The lower half is a tree view with a search box on top. It has two different purposes, depending on whether text entry is active or complete.

During text entry, the tree view is filled with the suggestions computed by the DynGenPar predictive parsing methods. The suggested tokens are not presented as a flat list, but as a tree, giving the user some context. The categorization is done according to what symbol the prediction algorithm encounters:

- If the prediction sees a specific token as the next symbol in the rule currently being parsed, this token is offered. In this case, no categorization is done.
- If, on the other hand, the prediction encounters a nonterminal category C, a tree of suggestions is built. At the highest level, the category C is shown. As explained in Section 2.2.3.1, the prediction algorithm then expands that category to a list of possible symbols it can start with. (In other words, it goes through all the rules for this category, and picks the first symbol of each rule.) This is repeated recursively, in a top-down manner, until a token is reached. (Directly or indirectly left-recursive rules are skipped because they cannot produce any additional leading tokens. This prevents running into an infinite loop.) Once a token is reached, the nonterminal N that expanded to the token is remembered, along with the token T it expanded to. Note that N can also be equal to C, but in most cases, it is not. In the produced prediction tree, the found nonterminal categories N are attached as children of C, and for each N, the tokens T it (directly) expands to are attached as children of N.

The tree is then sorted alphabetically at each level.

The search box, a text box above the tree view, allows filtering the suggested tokens. This is most useful when the (categorized) list of tokens is very long. Entering some text into the search box hides all suggested tokens that do not start with the entered text. If, as a result, some nonterminals no longer correspond to any visible tokens, they are also hidden.

Clicking on one of the tokens in the tree appends that token to the input. The updated input is immediately shown in the upper half of the user interface. Clicking on a nonterminal category has no effect. As a special case, clicking on one of the GF tokens requiring a value (`String`, `Int`, or `Float`) brings up a dialog box in which a value can be entered. (Warning: The value is currently **not** validated. Entering something that is not recognized as being of the appropriate type will confuse the parser and result in an error. For a `String`, what is valid depends on the context: In a formula, everything not containing the formula delimiter `$` is a valid string. Outside of a formula, only a single word, starting with a letter, and not corresponding to a word in the grammar is accepted as a string. See the list of limitations in the previous section.)

If the tree view offers no suggestions, this means that there is no valid way to continue the input. Normally, this means that the input is complete. However, context-sensitive constraints can also trigger this condition. E.g., if the grammar contains no noun starting with a vowel, the article “an” may still be initially offered, but then there will be no valid token to follow “an”. In that latter case, the user’s only option is to go back using the

Edit / Undo menu entry. *Undo* is also supported in other contexts. The PGF GUI does that by remembering the parse state after each token, making it easy to rewind to the previous one.

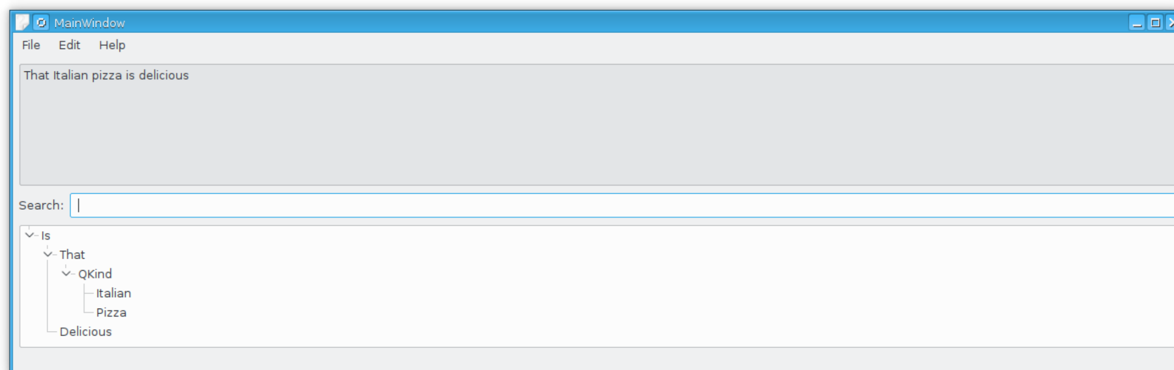


Figure 5.2: Screenshot of the PGF GUI displaying a produced syntax tree

Once the user has entered a complete sentence, the PGF GUI can build the corresponding GF syntax tree (Figure 5.2). In some grammars, as mentioned in the previous paragraph, it is easy to detect that the sentence is complete because the list of possible continuations will be empty. In other grammars, it is always possible to add some more text, and thus the list of suggestions is never empty. In either case, use the *Edit / Generate syntax tree* menu entry to build the GF syntax tree for the sentence. If the input is not complete, this results in an error dialog (with the message "The input is incomplete. Only complete input can be parsed."), and the user can continue inputting. If the input is complete, the syntax tree is built and replaces the suggestion tree in the lower half of the user interface. One can use the *Edit / Undo* menu entry to go back into inputting mode. The GF syntax tree is an abstracted version of the parse tree, generated from the parse tree using the `parseTreeToPmcfgSyntaxTree` and `filterCoercionsFromSyntaxTree` methods described in Section 5.2. It is the same tree that GF would also produce.

The *Edit / Clear* menu entry can be used at any time to restart inputting from scratch. (Warning: This cannot be undone. If one accidentally clears the input, one will have to reenter it completely.)

5.5 A GF Application Grammar for Mathematical Language

The Grammatical Framework (GF) was also used as the framework for some early research on grammars for natural mathematical language in the FMathL project (Neumaier [70]). This section documents two versions of a research grammar I implemented together with Peter Schodl in the GF programming language.

While GF ships with grammars for several natural languages, called the **resource grammars**, those grammars are not normally used directly to parse natural-language texts, for

several reasons:

- The resource grammars only know about purely linguistic grammar concepts, with few to no semantics. E.g., the resource grammar knows about noun phrases while a grammar for mathematics must instead have the concept of a mathematical object (that happens to be linguistically a noun phrase).
- This lack of semantic knowledge makes the resource grammars unsuitable for automatic translation. Every language comes with its own resource grammar, there is no complete abstract grammar common to all languages. (There is an attempt at a common abstract grammar, but it is limited and does not cover language-specific idioms.)
- The resource grammars are huge, making it very inefficient and impractical to parse large documents with them.
- The lexicon typically has to be provided by the application, because the sample lexica included with the resource grammars do not cover the subject-specific terms of any given application.
- The resource grammars also do not know about common application-specific idioms, which are therefore only recognized (if at all) as their linguistic components. E.g., the idiom *let x be an integer* is recognized as a third-person imperative ordering the proper noun x to *be* the noun phrase *an integer*. This also implies that such expressions are not necessarily translated into the correct idiomatic expression in other languages.

Therefore, it is commonplace in GF to define an **application grammar** that builds on top of the resource grammar, extracting only the subset reasonable for the given application, and adds application-specific knowledge. The application grammar normally consists of two parts:

- The **abstract grammar** defines how an abstract syntax tree should look like. It is typically a semantic representation, not a purely linguistic one. In a multilingual grammar, the abstract grammar is common to all supported languages. The abstract syntax trees conforming to the abstract grammar are therefore the common representation used as the *lingua franca* for translation. The abstract grammar roughly corresponds to the type systems (Schodl & Neumaier [86]) in Concise (without any usage annotations).
- The **concrete grammar** defines how the abstract concepts declared in the abstract grammar shall be worded in a specific language. There must be a separate concrete grammar for each natural language. However, it is often possible to use abstract GF resource grammar interfaces that work across several languages, and to define a common base concrete grammar that has to be extended only with the few things that depend on the language. (Whether to use the common base or whether to start from scratch is then a decision left to the implementor of the particular concrete grammar. Thus, some languages can use the common base, whereas languages using very peculiar idioms can opt to remain separate.) The concrete grammars roughly correspond to usages in Concise type sheets (see Section 3.2).

Thus, an early goal in the FMathL project was to write a GF application grammar – both an abstract grammar and a concrete grammar – for a small controlled subset of mathematical language. How to integrate this research with our existing Concise system was still an open question at that time.

The focus of that research was linearization (the opposite of parsing, i.e., generating text from abstract syntax trees), which was performed using the linearization functions of the C bindings for the GF Haskell runtime (see Section 5.1).

The first version of that application grammar was a small proof of concept entirely handwritten by me. Peter Schodl provided me with some examples of how the abstract syntax trees should ideally look like. We agreed on the following example sentences:

- Suppose that $K + f = x \in \mathcal{S}$, where K is an integer and $f \in [0, 1)$.
- S is the set of eigenvalues of the matrix A.

in LaTeX notation, i.e.:

- `Suppose that $K + f = x \in \mathcal{S}$, where $K$ is an integer and $f \in [0,1)$.`
- `$S$ is the set of eigenvalues of the matrix $A$.`

The grammar only covers the text, the formulas are represented as verbatim strings. In my handwritten application grammar, the above example sentences correspond to the following abstract syntax trees, in the LISP-style notation used by GF:

- `AssumptionSentence (Suppose (Stmt (Rel (Dollar "K + f = x \in \mathcal{S}"))) (And (Is (Var (Dollar "K")) (Det IndefSgArticle IntegerType)) (Stmt (Rel (Dollar "f \in [0,1)")))))`
- `StatementSentence (Is (Var (Dollar "S")) (DetQual DefSgArticle SetType (Qual OfPrep (DetQual IndefPlArticle EigenvalueType (Qual OfPrep (DetFormula DefSgArticle MatrixType (Dollar "A")))))))`

The complete GF source code of the proof of concept is reproduced below:
Abstract grammar `Math.gf`:

```
-- Copyright (C) 2010 Kevin Kofler

abstract Math = {
  flags startcat = Sentence ;
  cat
    Sentence ; Assumption ; Statement ; Relation ; Variable ; Formula ;
    Object ; Article ; ObjectType ; Preposition ; Qualification ;
  fun
    IndefSgArticle : Article ;
    IndefPlArticle : Article ;
    DefSgArticle : Article ;
    DefPlArticle : Article ;
    IntegerType : ObjectType ;
    SetType : ObjectType ;
    EigenvalueType : ObjectType ;
```

```
    MatrixType : ObjectType ;
    OfPrep : Preposition ;
    Det : Article -> ObjectType -> Object ;
    DetQual : Article -> ObjectType -> Qualification -> Object ;
    DetFormula : Article -> ObjectType -> Formula -> Object ;
    Qual : Preposition -> Object -> Qualification ;
    Dollar : String -> Formula ;
    Var : Formula -> Variable ;
    Rel : Formula -> Relation ;
    Stmt : Relation -> Statement ;
    Is : Variable -> Object -> Statement ;
    And : Statement -> Statement -> Statement ;
    Suppose : Statement -> Statement -> Assumption ;
    AssumptionSentence : Assumption -> Sentence ;
    StatementSentence : Statement -> Sentence ;
}
```

Concrete grammar `MathEng.gf`:

```
-- Copyright (C) 2010 Kevin Kofler

--# -path=alltenses:../lib/prelude:../lib/alltenses

concrete MathEng of Math = open Prelude, ResEng, TryEng, ExtraEng in {
  flags language = en_US;
  lincat
    Sentence = Text ;
    Assumption = Imp ;
    Statement, Relation = S ;
    Variable = PN ;
    Formula = Str ;
    Object = NP ;
    Article = TryEng.Det ;
    ObjectType = N ;
    Preposition = Prep ;
    Qualification = {p : Prep ; o : NP} ;
  lin
    IndefSgArticle = a_Det ;
    IndefPlArticle = aPl_Det ;
    DefSgArticle = the_Det ;
    DefPlArticle = thePl_Det ;
    IntegerType = mkN "integer" ;
    SetType = mkN "set" ;
    EigenvalueType = mkN "eigenvalue" ;
    MatrixType = mkN "matrix" "matrices" ;
    OfPrep = mkPrep "of" ;
    Det article objecttype = TryEng.mkNP article objecttype ;
    DetQual article objecttype qualification = TryEng.mkNP article (mkCN
      (mkN2 objecttype qualification.p) qualification.o) ;
```

```
    DetFormula article objecttype formula = TryEng.mkNP article (mkCN
      objecttype (TryEng.mkNP (formulaToPN formula))) ;
    Qual prep obj = lin Qualification {p = prep ; o = obj} ;
    Dollar s = "$" ++ s.s ++ "$" ;
    Var f = formulaToPN f ;
    Rel f = lin S {s = f} ;
    Stmt relation = relation ;
    Is var obj = MathEng.mkS (mkCl (TryEng.mkNP var) (mkVP obj)) ;
    And s1 s2 = MathEng.mkS and_Conj s1 s2 ;
    Suppose statement withcl = mkImp (mkVP (mkVS (mkV "suppose"))
      (MathEng.mkS statement (mkAdv (mkSubj "where") withcl))) ;
    AssumptionSentence a = mkText (mkUtt a) ;
    StatementSentence s = mkText s ;
  oper
    formulaToPN : Str -> PN = \f -> lin PN {s = table
      {Gen => f ++ Prelude.BIND ++ "'s" ; _ => f} ; g = nonhuman} ;
    -- add a mkS S Adv
    -- Without this, it "works" for English, but only because S and Adv happen
    -- to have the same linearization!
    -- (It actually uses mkS Adv S with the S and Adv swapped.)
    mkS = overload TryEng {
      mkS : S -> Adv -> S = \s,a -> lin S {s = s.s ++ "," ++ a.s} ;
    } ;
}
```

The issue we had with this approach was that for every addition to the grammar, it was necessary to update two different places in lockstep: the type system in the semantic memory and the GF grammar. If those went out of synchronization, it was no longer possible to convert the abstract syntax trees in either direction.

Therefore, Peter Schodl came up with a new idea: He wrote a MATLAB script that *automatically generated* a GF abstract grammar from his MATLAB implementation of the semantic memory. I then wrote the corresponding concrete grammar. To help with both debugging the abstract grammar and implementing the concrete grammar, I wrote a Perl script `mkdummy.pl` that automatically generated a concrete grammar for the dummy language `Tre` (short for "trees"). That dummy language linearizes parse trees to a LISP-style notation essentially identical to the input notation. This allows a quick verification that the abstract grammar is valid. If it is not, either `mkdummy.pl` will complain, or the GF compiler or runtime will produce an error, or the mistake is easy to spot in the output. It also produces a starting point for the actual English concrete grammar, like a form to fill in. I wrote the concrete grammar for English based on that starting point.

This second version of the application grammar covers a larger controlled subset of mathematical language than the first proof of concept. In addition, the MATLAB implementation of the semantic memory was able to automatically generate the GF input that could then be sent to GF for linearization. The drawback is that the abstract syntax trees are more verbose than in the first version. E.g., the sample input:

```
(_Sentence (_SentencePartLink (_Let2SentencePart ((_Let (_Obj
(_Empty2TextUnit (_Empty)) (_Concept2ConceptGeneral ((_Concept
(_Empty2ConceptGeneral(_Empty)) (_Set2SentencePart ((_Set)))
(_Qualification (_Concept2ObjConc ((_Concept (_Positive2ConceptGeneral
((_Positive))) (_Integers2SentencePart ((_Integers)))
(_Empty2Qualification (_Empty))))))))) (_MathString2Expression
((_MathString ("S")))) (_Empty2Expression (_Empty)) (_Empty2Qualification
(_Empty)) (_Empty2TextUnit (_Empty))) (_Empty2Expression (_Empty))
(_Concept2ConceptGeneral ((_Concept (_Empty2ConceptGeneral (_Empty))
(_Set2SentencePart ((_Set))) (_Qualification (_String2ObjConc ("item
size"))))))))) (_Empty2SentencePartLink (_Empty))))
```

produces the LaTeX output:

```
Let the set of positive integers $S$ be the set of item sizes.
```

which renders to:
Let the set of positive integers S be the set of item sizes.

These proofs of concepts have not been pursued further, as the focus of our research switched to grammars written as Concise type sheets annotated with **usages** (see Section 3.2). The insights gained from the research on GF application grammars were extremely useful for the grammars now written as Concise type sheets, and research is ongoing on how to make use of the information encoded in the GF resource grammars within the Concise framework. While GF is an extremely powerful framework, it assumes a way of working that turned out to map poorly to the current and intended workflows in the FMathL project, making this way of interfacing with GF too inconvenient in the long run:

- Concise has its own representation of type systems (Schodl & Neumaier [86]). Using a GF application grammar for a type system requires mapping that type system to a GF abstract grammar. The mapping Peter Schodl developed produced really complex abstract syntax trees, due to the need for type cast functions. It is unclear to our work group whether we can produce better mappings automatically without reducing the flexibility of our type systems.

- In Concise, the goal is to encode the entire grammar in our type sheets (see Section 4.1), not just the abstract grammar, but also the concrete grammar. Concrete grammar information is encoded through so-called usages (see Section 3.2). Producing a GF application grammar from that information would require automatically generating not only the abstract grammar, but also the concrete grammar, which would be significantly harder. Writing the concrete grammar for the automatically-generated abstract grammar by hand requires writing complex code such as:

  ```
  _Concept2ObjConc x1 = lin ObjConc {e = x1.e ; np = table {Definite
  => TryEng.mkNP (DefArticle x1.n) x1.cn ; Indefinite => TryEng.mkNP
  (IndefArticle x1.n) x1.cn}} ;
  ```

- A goal of the FMathL project is to make writing grammars accessible to every mathematician. Unfortunately, to access the large repository of grammatical knowledge that comes with GF, it is necessary to write idiomatic sentences (e.g., for

`Is var obj`) using APIs such as `MathEng.mkS (mkCl (TryEng.mkNP var) (mkVP obj)) ;`. A mathematician will want to write something like `#var is #obj` instead. That verbatim notation has obvious issues when it comes to things such as subject-verb agreement, but the FMathL project aims at finding a middle ground between the complete grammatical analysis required by the GF resource grammar API and straightforward verbatim text with no grammatical information. Some research in that direction is now ongoing.

Chapter 6

Applications of DynGenPar to Natural Language

The main goal of DynGenPar is natural language processing, though, as described in Chapter 4, DynGenPar has also been successfully used for several formal languages. This chapter enumerates and documents applications of DynGenPar to various controlled subsets of natural language, including

- Naproche (Cramer et al. [17], Koepke et al. [46]),
- MathNat (Humayoun & Raffalli [41], Humayoun [40, 39]), and
- a subset of natural LaTeX formula notation (without dedicated semantic markup),

progressing towards the research goal that has driven DynGenPar development. The long-term aim is to grow the supported controlled subset of natural language until it ultimately covers all commonly-used idioms and to allow the user to add custom rules for anything beyond that.

The first section documents a hierarchical DynGenPar grammar for the Naproche language, a controlled natural language for mathematical logic. The second section describes a toolchain, based on LaTeXML (Miller [67]), to automatically process complete LaTeX documents with Concise and DynGenPar. The third section describes a Concise type sheet for a limited subset of natural mathematical language whose main feature is mathematical definitions. This type sheet serves as a proof of concept for dynamic rule additions, which happen as definitions are encountered. Andreas Pichler has, in his master thesis (Pichler [73]), extended this type sheet to be able to parse a text that introduces matrix calculus. The fourth section describes a work-in-progress Concise type sheet for the MathNat language, a larger subset of natural mathematical language. Finally, the fifth section details a grammar for LaTeX formulas in natural notation implemented as a Concise type sheet.

6.1 Naproche Parser using DynGenPar

This section introduces the DynGenPar parser for the **Naproche** (Cramer et al. [17], Koepke et al. [46]) language, a controlled natural language for mathematical logic. It was the first controlled natural language implemented using DynGenPar. The implementation is hierarchical: a text parser and a subordinate formula parser, both using Flex (Flex Project [26]) and DynGenPar. In other words, there is not one single grammar, but a pair of DynGenPar grammars that together implement the Naproche language. Unlike later grammars in this chapter, which are implemented as Concise type sheets, the pair of Naproche grammars is implemented directly in C++.

An important note is that the Naproche EBNF grammar that was available to me (Kühlwein [62]) appears to be from an early version of Naproche. Newer versions implement additional language constructs that are only roughly documented (without a precise grammar) in the Naproche wiki (Cramer & Kühlwein [18]), but not included in the grammar. Since the source code of the Naproche implementation is not public, the extended grammar used by the current version of the actual implementation was not available to me. Therefore, and because this was essentially a proof of concept, I decided to implement the available EBNF grammar as is, without the extensions added in later versions.

The Naproche EBNF grammar is a static context-free grammar. Therefore, it is also suitable for traditional parser generators. Hence, I wrote a first implementation using Flex (Flex Project [26]) and GNU Bison (Free Software Foundation [28]). Flex was used to generate the lexer, Bison was used to generate the parser. The grammar is not in LALR(1) (DeRemer [20]), so I used Bison's Generalized LR (GLR) mode. (The Bison GLR mode is actually generalized LALR(1), but that is sufficient to parse general context-free grammars.) Subsequently, I ported the grammar from Bison to DynGenPar, still using Flex for the lexer. (Flex lexers are one of several possible token sources for DynGenPar, see Section 2.2.3.5.) This allowed benchmarking the performance of DynGenPar against Bison. Those measurements showed that at raw parsing, DynGenPar comes within an order of magnitude of Bison's raw parsing speed (Bison is only 4 to 5 times faster), but DynGenPar is 11 times faster at converting the grammars to the format that can be used for parsing (and that does not even include the time required to compile the C code Bison outputs). The detailed results can be found in Section 7.1.1. One notable difference is that the Bison version is in C and uses the C mode of Flex, whereas the DynGenPar version is in C++ and uses the Flex lexers as C++ classes.

In both versions, formulas are parsed by a separate grammar. For the main Naproche grammar, a formula is just a token. It is recognized as a token by the main lexer, using the regular expression `\$[^$]*\$` that matches dollar-enclosed strings. The lexer then passes the whole string to the formula parser, which scans it using the dedicated formula lexer and parses it using the dedicated formula grammar. The Naproche formula grammar accepts a small controlled subset of LaTeX corresponding to first-order logic, documented informally in the Naproche wiki (Cramer & Kühlwein [18]). In the DynGenPar implementation, the formula parser returns a parse tree, which is returned by the main token source in lieu of the leaf node that would be usual for a token. This parse tree is then automatically

attached to the main parse tree as a subtree in lieu of a leaf. This DynGenPar feature is called hierarchical parsing (see Section 2.2.3.5). Note that this works differently in the Bison implementation because it does not produce an explicit parse tree, but executes actions on the fly.

The EBNF grammar for Naproche formulas used in my implementation is based on the expression grammar pattern and on the informal documentation from Cramer & Kühlwein [18]. It is reproduced below:

variable_symbol → `m` | `n` | `t` | `u` | `v` | `w` | `x` | `y` | `z`
constant_symbol → `\emptyset` | `c` | `0` | `1` | `2` | `3` | `4` | `5`
function_symbol → `f` | `f2` | `g` | `h` | `*` | `+` | `^` | `’` | `succ` | `\cup` | `\cap`
relation_symbol → `=` | `\neq` | `<` | `>` | `\leq` | `\geq` | `\in` | `\subset` | `\supset` | `R` | `r` | `M` | `ord`
  | `trans` | `contradiction`
term → *variable_symbol* | *constant_symbol* | *function_symbol* `(` *term*$^+$ `)`
  | `(` *term function_symbol term* `)` | *term* `’`
atomic_formula → *relation_symbol* `(` *term** `)` | *term relation_symbol term* | `(` *formula* `)`
quantifier → `\forall` | `\exists`
quant_formula → *atomic_formula* | *quantifier variable_symbol*$^+$ `(` *formula* `)`
neg_formula → *quant_formula* | `\neg` *neg_formula*
logical_and → `\wedge` | `\and`
and_formula → *neg_formula* | *and_formula logical_and neg_formula*
logical_or → `\vee` | `\or`
or_formula → *and_formula* | *or_formula logical_or and_formula*
logical_implies → `\rightarrow` | `\implies`
implies_formula → *or_formula* | *implies_formula logical_implies or_formula*
logical_equivalent → `\leftrightarrow` | `\equivalent`
formula → *implies_formula* | *formula logical_equivalent implies_formula*
dollar_formula → `$` *formula* `$`

The space, tab, newline, dot (`.`), colon (`:`), and comma (`,`) characters are treated as token separators and otherwise ignored. No token is produced for those characters by the lexer. The `succ`, `ord`, `trans`, and `contradiction` tokens are also accepted with the first letter capitalized. The start category is *dollar_formula*.

The DynGenPar implementation of the Naproche grammar also demonstrates the prediction functionality of DynGenPar (see Section 2.2.3.1). If the input is incomplete or truncated, but can be completed to an error-free Naproche document, the valid possibilities for the next token are enumerated to `stderr`. The proof of concept version does not implement prediction for formulas or in the case of errors, but it would be possible to add them if needed.

The output of the Naproche parser is a MATLAB script that produces the parse tree as a record in Peter Schodl's MATLAB implementation of the semantic memory. Nodes in the parse tree are constructed using commands such as

```
node2 = mksentence(node1);
```

The details of the output format can be found in Schodl [84]. In the Bison implementation, parse actions directly write these constructor commands to `stdout`. In the DynGenPar implementation, DynGenPar first builds the parse tree in memory. Subse-

quently, that parse tree is walked to output the constructor commands to `stdout`. This input format is no longer used by Concise, but since there was no immediate need to process Naproche in Concise, the output of the Naproche parser has not yet been updated. Alternatively, the DynGenPar version of the Naproche parser can also output the parse tree in an indented plain-text representation designed to be human-readable.

The main test case for the Naproche parser was a first-order formulation of the Burali-Forti paradox (Burali-Forti [10]), encoded in the Naproche language by Peter Schodl (Schodl [84]). The reason a paradox was chosen as the test case is that paradoxes stress the corner cases in logic, and therefore demonstrate the expressiveness of the language. Peter Schodl's MATLAB implementation of the semantic memory (Schodl [84]) is able to automatically write the Naproche text from the internal semantic memory representation. My Naproche parser can perform the opposite transformation.

In order to allow parsing LaTeX documents containing Naproche text, I wrote a Perl script to extract the text from the XML output of LaTeXML (Miller [67]), bringing it into a plain-text format that the Naproche parser can handle. The original LaTeX document structure is not preserved. In addition, for convenience, a small shell script runs LaTeXML (assumed to be installed system-wide), then my Perl script extracting the text, and finally the Naproche parser. Those scripts were the initial prototype for the `TextDocument` toolchain that allows reading LaTeX documents into Concise, preserving their document structure (see Section 6.2).

The DynGenPar implementation of the hierarchical Naproche grammar is included as an example in the Open Source releases of DynGenPar. The Burali-Forti sample input is also included, in the `examples` directory. In addition to the valid version of the sample input, that directory also contains a deliberately erroneous version to test the error reporting and a deliberately truncated version to showcase the prediction functionality.

6.2 The `TextDocument` Toolchain

This section describes a toolchain to automatically process complete LaTeX documents with Concise and DynGenPar, the **`TextDocument`** toolchain. The toolchain works by

1. converting the LaTeX documents to **XML** (**eXtensible Markup Language**) using **LaTeXML** (Miller [67]),
2. turning the XML into a Concise record (of type `TextDocument`) representing the document structure, and
3. sending every paragraph to DynGenPar to convert the document structure record to a semantic one, using a grammar such as `BasicDefinitions` (Section 6.3) or `BasicReasoning` (Section 6.4).

In the first step, the toolchain uses LaTeXML 0.7.0 (Miller [67]) to generate an XML representation closely reflecting the structure of the original LaTeX. (For practical reasons, the used copy of LaTeXML was frozen to a fixed version.) LaTeXML requires some information about every LaTeX stylesheet used in the document, to know how it should process the macros. (Should it expand them? Should it produce some specific XML for them?)

An extensive collection of such LaTeXML information for many common and not so common LaTeX stylesheets has been collected in the **arXMLiv** project (Stamerjohanns & Kohlhase [90], Kohlhase & Ginev [60]). It can be checked out from `https://svn.kwarc.info/repos/arXMLiv/trunk/sty/`. In order to avoid working on a moving target, the `TextDocument` toolchain project settled on a snapshot (a checkout of the repository) from October 3, 2010. That revision was verified to work with the LaTeXML 0.7.0 release.

Some semantic information that is needed later in the parsing step cannot be satisfactorily expressed in terms of existing LaTeX packages. Therefore, I wrote a custom **FMathL LaTeX stylesheet** (`FMathL.sty`) and a corresponding LaTeXML description `FMathL.sty.ltxml`. This allows extending the LaTeX language with custom macros that can be handled in a special way by LaTeXML. `FMathL.sty` currently defines the following macros, each taking one argument:

- `\define`: specifies a definition defining the term given as the argument. In LaTeX, this simply turns the argument bold. In LaTeXML, the macro puts the argument into a special text class *"define"*. This preserves the information needed to create dedicated *"define"* objects in the semantic representation built in the next step.
- `\raw`: specifies comments that should be left unparsed. In LaTeX, this simply turns the argument italic. In LaTeXML, the macro puts the argument into a special text class *"raw"*. This preserves the information needed to create dedicated *"raw"* objects in the semantic representation built in the next step.
- `\email`: specifies an e-mail address. Since e-mail addresses follow basically the same formatting rules as URLs, it is common practice in LaTeX to simply put the e-mail address into a `\url` tag. However, this is bad in a semantic representation because an e-mail address is in fact not, by itself, a valid URL. To turn an e-mail address into a valid URL, `mailto:` must be prepended. That, however, is usually unwanted in printed documents. The `\email` macro solves this dilemma. In LaTeX, it is simply equivalent to `\url`. In LaTeXML, `mailto:` is additionally prepended, e.g., `\email{foo@example.com}` is equivalent to `\url{mailto:foo@example.com}`.

The second processing step uses two programs implemented by me to convert the output of LaTeXML to a Concise record representing the document structure. Both programs are implemented in C++ using the `QtCore` and `QtXml` libraries. The `processxml` tool is an XML to XML transformation from the LaTeXML output format to the XML representation of a Concise record. Using XSLT (eXtensible Stylesheet Language Transformations) (W3C [100]) for the task was considered, but imperative code turned out to be more convenient in this case. The `xmltocnr` tool converts the record from the XML representation to the record sheet representation. The reason the conversion is implemented as a two-step process, with two separate executables, is technical: `processxml` works using the **SAX** (**Simple API for XML**) API, which walks the input from left to right and triggers events when an opening or closing tag is encountered. This is fast, but producing well-indented record sheets this way is tricky. `xmltocnr` walks the **DOM** (**Document Object Model**) tree, an abstract syntax tree, of the input XML and outputs the record sheet recursively, increasing the indentation with the recursion depth. The `QtXml` module implements both APIs. The `xmltocnr` tool is also used when importing a type sheet into Concise (see Section 4.1).

The `TextDocument` type system, defined in the `TextDocument.cnt` type sheet, represents the organizational structure of a LaTeX document. A text document consists of the **content** in the form of an **element list** and of a **layout**. The layout field is currently not used because my work focused on the content. The element list can contain any of the following **referenceable items** (`refItems`) as elements:

- *displays*, created using the `$$` command or the `equation` or `eqnarray` environments. A display contains one or more formulas.
- *verbatim* environments.
- *figures*, created using the `figure` environment.
- (text) *paragraphs*. These are typically not inside any particular environment. (However, the `quote` environment is also considered a text paragraph.) Their contents are detailed below. Note that the `paragraph` and `subparagraph` environments are considered sections, not paragraphs. They can, and usually do, contain more than one actual paragraph of text.
- *tables*, created using the `table` environment.
- *(sub)sections*, created using the `chapter`, `section`, `subsection`, `subsubsection`, `paragraph`, or `subparagraph` environments. Each section contains an element list that can contain lower-level sections, so that the document forms a tree of elements.

In addition, each element list may optionally contain a **header**, which must contain a *title* and may contain an *abstract*, *keywords*, *authors*, a *date*, a *copyright* statement, and a *comment*. All the fields except for the title are optional and can appear in any combination. The element list may also optionally contain a **trailer**, which may contain a *bibliography* and one or more *indices*. Both fields in the trailer are optional. For sections, only the title in the header is used. All other header fields and the trailer are filled in by `processxml` only for the top-level element list.

A **paragraph** can be in either an unparsed or a parsed (semantic) representation. The `TextDocument` toolchain initially produces all paragraphs in the unparsed representation. Then, a type sheet such as `BasicDefinitions` (Section 6.3) or `BasicReasoning` (Section 6.4) can be used to parse the paragraphs into a semantic representation defined by the used type sheet. Only the unparsed representation is defined in the `TextDocument` type sheet itself. It consists of a list of **terms**, which can be any of the following:

- a *definition*, marked up with the `\define` command defined in `FMathL.sty` (see above).
- a *raw comment* that should not be parsed, marked up with the `\raw` command defined in `FMathL.sty` (see above).
- a *footnote*, consisting of a paragraph.
- a *reference* (`\ref`), pointing to a referenceable item (see the list above).
- a *citation* (`\cite`), pointing to an entry in the bibliography.
- a *newline*.
- a *word*.

- a *mark*, e.g., a punctuation mark. A mark can also be any other symbol that is not a word or a formula.
- a *formula*, created using the `$` command.
- a *URL* (Uniform Resource Locator, i.e., an Internet address).
- a *text block* of inline verbatim text, created using the `\verb` command.
- an inline *figure*, created using the `\includegraphics` command or the `picture` environment outside of a `figure` environment.
- an inline *table*, created using the `tabular` environment outside of a `table` environment.
- an inline *display*, which can come up in quote paragraphs. The entire `quote` environment is treated by LaTeXML as a paragraph. This also includes displays. In contrast, for normal paragraphs, a display is treated by LaTeXML as a paragraph delimiter and never considered part of the paragraph.

No semantic meaning is assigned to the terms in the unparsed paragraph. That is the purpose of the semantic representations produced by parsing.

At the end of these two processing steps, the LaTeX document is in the form of a Concise record that can be imported into the semantic memory. That record fully represents the structure of the LaTeX document, while letting the contents in a mostly unprocessed form. As explained above, paragraphs are left unparsed. Likewise, formulas are kept in their LaTeX string representation. (Only the transformations that LaTeXML applies are done.) In the Concise GUI, the menu entry *Read> LaTeX document* automates the above two processing steps, invoking the external executables of the `TextDocument` toolchain.

The third, and final, processing step, operates on the semantic memory and is fully implemented within Concise. In that step, every paragraph and every formula is sent to DynGenPar for parsing. As a result, the record is transformed from a record representing only the document structure to a record representing both the structure and the semantics of the document.

Currently, it is necessary to first parse the paragraphs and then, separately, the formulas. A future version of the `TextDocument` toolchain shall use the hierarchical parsing feature of DynGenPar (see Section 2.2.3.5) to do this as part of the parsing process, as is already implemented in the Naproche parser (see Section 6.1).

The paragraphs are parsed one at a time using a type sheet such as `BasicDefinitions` (Section 6.3) or `BasicReasoning` (Section 6.4). That type sheet specifies not only the grammar, but also the resulting semantic representation. The type sheet is converted to a Concise grammar only once, and the resulting grammar is reused for all the paragraphs. Not only is this more efficient, but it also allows dynamic rule additions from a previous paragraph (as done, e.g., in the `BasicDefinitions` grammar, see Section 6.3) to carry over onto later paragraphs.

Concretely, a **`TextParagraphIterator`** walks the `TextDocument` record and returns the object IDs of the unparsed paragraphs in the record, in document order. Each paragraph is then sent to DynGenPar for parsing. There is, however a complication: The unparsed

paragraphs are subrecords. DynGenPar, however, operates on a stream of tokens, not on a record. Therefore, the paragraph records have to be linearized to a token stream first.

Linearizing a paragraph record to a stream of DynGenPar tokens is the task of the **`ConciseTokenSource`**. In the simplest case, the paragraph is simply a linked list of words and punctuation signs. Such a linked list trivially corresponds to a sequence of tokens, which the `ConciseTokenSource` can return one after the other. However, a paragraph can also contain items that require a more complex treatment. Therefore, the `TextDocument` typesheet contains special `tokensource` usages (see Section 3.2) that tell the `ConciseTokenSource` how to linearize each item type. The syntax of the usages is given in Section 4.1.

The linearization done by the `ConciseTokenSource` differs from other linearization tasks in Concise in that the required output is not text, but a sequence of tokens. The grammars operating on `TextDocument` paragraphs are not scannerless, they operate on word tokens. In addition, the tokens can have a value attached. (The attached value is typically an object ID.) This is important because it allows retaining information that is not parsed, such as formulas and comments, across the parsing process. The object ID of the subrecord that should be copied verbatim is attached to the token by the `ConciseTokenSource`. The object ID remains attached to the token in the parse tree produced by DynGenPar. During the generation of the resulting record, the object ID is retrieved from the token, and the object it identifies is attached to the correct location in the output record. What tokens should be output and with what value (if any) is encoded in the usages through the usage function `#!out`, detailed in Section 4.1. The function takes, as a required parameter, the ID of the token to output and, as an optional parameter, the ID of the value to attach to it.

The `ConciseTokenSource` linearizes the item types in paragraphs as follows:

- A *definition* is initially linearized as a special token `DEFINE`, with the object ID of the concept being defined attached as a value. However, it is treated specially by a token hook, as explained below.
- A *raw comment* is linearized as a special token `RAW`, with the object ID of the comment attached as a value.
- A *footnote* is currently treated as if its contents were part of the text of the paragraph.
- A *reference* is linearized as a special token `REF`, with a pair, consisting of the object ID of the referenced item and of the external ID of the optional label string, attached as a value.
- A *citation* is linearized as a special token `CITE`, with a pair, consisting of the object ID of the referenced item and of the external ID of the optional label string, attached as a value.
- A *newline* is ignored, i.e., it produces no token at all.
- A *word* is a token by itself. Its token ID is the external ID of the external of type `UniqueString` representing the word.

- A *mark* is a token by itself. Its token ID is the external ID of the external of type `Character` representing the mark.
- A *formula* is linearized as a special token `FORMULA`, with the external ID of the string representation of the formula (without the enclosing dollar signs) attached as a value.
- A *URL* is linearized as a special token `URL`, with a pair, consisting of the external ID of the referenced URL and of the object ID of the optional label (a paragraph), attached as a value.
- A verbatim *text block* is linearized as a special token `TEXTBLOCK`, with the external ID of external of type `String` representing the text block attached as a value.
- An inline *figure* is currently ignored, i.e., it produces no token at all.
- An inline *table* is currently ignored, i.e., it produces no token at all.
- An inline *display* is linearized as the sequence of formulas composing it, i.e., a sequence of one or more `FORMULA` tokens, with the external ID of the string representation of the formula (without the enclosing dollar signs) attached as a value.

A special feature of the `ConciseTokenSource` is that it allows installing a **token hook**, a method that is called for every encountered token and that can act on them and modify them before they are returned to DynGenPar. Currently, there is exactly one such token hook, which is always enabled. That token hook processes the special `DEFINE` token. If that token is encountered:

1. The attached value is retrieved. It is the object ID of the concept being defined. The concept is of the same type as a paragraph, i.e., a list of paragraph items.
2. The concept is linearized into a stream of tokens by using another `ConciseTokenSource` instance on it.
3. The resulting stream of tokens is converted to a string.
4. That string is converted to an external of type `Name`, the name of the concept.
5. A new rule is added to the grammar, which derives, from the special category `Name`, the stream of tokens produced in step 2. The rule label is an object of type `ExternalNameBuilder` (see Section 3.2), producing the name of the concept. Thus, if the grammar accepts the category `Name`, any further appearance of the stream of tokens defining the concept will be recognized as the name of the concept. If the grammar does not reference the category `Name`, the added rule has no effect.
6. Finally, instead of the original `DEFINE` token, a `NEWCONCEPT` token is returned, with as attached value the name of the concept. `NEWCONCEPT` is a different token than `DEFINE` because `NEWCONCEPT` has an external `Name` attached, whereas `DEFINE` has a paragraph object attached. For this to work, the grammar must accept the `NEWCONCEPT` token. Otherwise, a parse error is produced.

The `BasicDefinitions` grammar from Section 6.3 uses this feature.

DynGenPar is run on the token stream produced by the `ConciseTokenSource`, using the grammar previously chosen for the document. The resulting parsed record is a semantic

representation of the paragraph. It is attached to the `TextDocument` record in lieu of the original unparsed paragraph.

The above parsing of paragraphs was initially implemented in a test program called `LaTeXTest`. It is now also available from the Concise GUI, by right-clicking on a record of type `TextDocument` and choosing the *Edit> Record> Parse paragraphs* menu entry.

Parsing the formulas is currently only implemented in the `LaTeXTest` test program. It is planned to add it to the user interface once the `LaTeXFormulas` grammar (see Section 6.5) is sufficiently finalized. However, as previously mentioned, the ultimate goal is to not require a separate formula parsing step at all, but to use the hierarchical parsing feature of DynGenPar (see Section 2.2.3.5) to do this as part of parsing the paragraphs.

The `LatexTest` test program can optionally send all formulas in the `TextDocument` record to an instance of DynGenPar using the formula grammar from Section 6.5. In order to do so, it uses a **`TextFormulaIterator`** that walks the entire `TextDocument` record and returns the object IDs of every formula found, in the order of appearance in the document. Both unparsed and parsed paragraphs are supported. Respecting the order of appearance in a parsed paragraph relies on `order` usages in the type sheet that defines the parsed paragraphs. For linked lists (as found in unparsed paragraphs), the `TextFormulaIterator` defaults to walking in the natural list order (`next` field last), which is usually the expected ordering. For other types, if no `order` usage is given, the order is arbitrary. The contents of `\raw` comments are ignored. In particular, formulas contained in them are not returned for parsing. This complies to the intent of the `\raw` tag. Each formula found by the `TextFormulaIterator` is sent to DynGenPar, and the result is stored in the formula object in addition to the original string representation.

It is planned to allow the user to select the formula grammar to use for parsing. In that way, both sTEX (KOHLHASE [59]) (using a formal language grammar that is yet to be written) and a subset of the traditional LaTeX formula syntax (using the grammar from Section 6.5) can be accepted as input. It will also allow customizing the formula grammar for specific applications.

6.3 `BasicDefinitions` – A Proof of Concept for Dynamic Rule Additions through Mathematical Definitions

This section describes a Concise type sheet that serves as a proof of concept for the handling of mathematical definitions, called **`BasicDefinitions`**. The `BasicDefinitions` type sheet was the first type sheet aiming at implementing functionality required to parse natural language. It has been written cooperatively by Arnold Neumaier, Ferenc Domes, and me. Its main feature is that definitions encountered automatically trigger a hook in the token source that dynamically adds a rule at runtime. Unlike the `OptProbl` grammar (see Section 4.5) that also dynamically adds rules to the grammar (through parse actions, see Section 2.2.3.4), the `BasicDefinitions` grammar is implemented within the Concise framework and operates on `TextDocument` (see Section 6.2) paragraphs.

Unlike the Naproche grammar documented in the previous section, the `BasicDefinitions` grammar operate neither on raw text, nor on raw LaTeX. Instead, it operates on a paragraph in the Concise `TextDocument` (see Section 6.2) representation. Unparsed paragraphs are represented there essentially as a linked list of words. Therefore, the `BasicDefinitions` grammar does not need a classical lexer as its token source. Instead, a special token source, the `ConciseTokenSource`, walks the linked list of words and returns each word to DynGenPar as a token. Thus, the `BasicDefinitions` grammar operates on word tokens. The `ConciseTokenSource` is described in more details in Section 6.2.

The static part of the `BasicDefinitions` grammar covers just enough idiomatic mathematical English to represent basic mathematical definitions. The most complex sentence the grammar was tested with is a sentence defining a table:

> A **table** A with **rows** R and **columns** C (short: an $\boldsymbol{R \times C}$ **table**) associates with every **row** $i \in R$ and every **column** $k \in C$ the **entry** A_{ik}.

The text printed in bold is marked up with the `\define` macro declared in the `FMathL.sty` style sheet (see Section 6.2). The whole expression "$R \times C$ table" is enclosed in a single `\define` macro.

The `BasicDefinitions` grammar represents a parsed paragraph as a linked list of sentences. Every sentence is either a comment, marked up with the `\raw` macro declared in the `FMathL.sty` style sheet (see Section 6.2), or one of three kinds of definitions. By design, comments are not parsed. Thus, their contents remain stored in the unparsed paragraph format. Note that a comment may actually consist of multiple natural language sentences, but is treated as one logical sentence.

The grammar initially knows only two built-in **concepts**: *object* and *statement*. Any additional concept has to be defined in the input text through a mathematical definition.

The following three kinds of definitions are recognized:

1. `definition`, e.g.: A **domain** D associates with every object x a statement $x \in D$.
2. `phraseDef`, e.g.: If $x \in D$ we say that $\boldsymbol{x}$ **is in** $\boldsymbol{D}$.
3. `formulaDef`, e.g.: In place of $x \in P$ we also write $P(x)$.

In the examples, the text in bold is marked up with the `\define` macro.

Formulas are not parsed by the `BasicDefinitions` grammar. They are treated as one token that has the LaTeX string representation attached. That string representation is kept unchanged in the output. However, the formulas can be sent to the grammar for LaTeX formulas (see Section 6.5) in a subsequent separate parsing step.

New rules are added to the grammar whenever a term enclosed in the `\define` macro is encountered. The term can consist of more than one word, e.g., `\define{new concept}`. It is assumed to be a new term: defining the same term again will create an ambiguous rule. Once the term is defined, it will be recognized as a concept by the new rule, so all further uses of the term need not, and in fact must not, be marked up with `\define`.

The `\define` macro is represented by a special *"define"* object in the semantic memory, which produces a special `DEFINE` token in the `ConciseTokenSource` (see Section 6.2). Therefore, there is no need for parse actions (see Section 2.2.3.4) to recognize the definition, it can be done directly at token level. For this purpose, the `ConciseTokenSource` allows installing a **token hook**, a method that is called for every encountered token and that can act on them and modify them before they are returned to DynGenPar.

The token hook is not specified in the `BasicDefinitions` type sheet. Instead, it is hardcoded in the Java code of Concise itself. It is installed by the `concise.parser.Grammar` class documented in Section 3.2. The token hook acts on tokens with the special ID `DEFINE`. (Actually, strictly speaking, the ID is an integer: the external ID of the `UniqueString` external `"DEFINE"`.) Any other tokens are returned to DynGenPar unchanged. If a `DEFINE` token is encountered, the following steps are performed:

1. The value attached to the token is retrieved. It is also an integer: the object ID of the entry in the `concept` field of type `paragraph` of the `define` object. This corresponds to the text passed as a parameter (within the curly braces) to the `\define` LaTeX macro. It is of type `paragraph` because it can contain more than one word, and even other tokens that can occur in a paragraph, such as formulas.
2. A string is formed from the contents of that concept. E.g., for `\define{new concept}`, the string `"new concept"` is built. That string is converted to the external type `Name`, and the ID of the resulting external is retrieved.
3. A rule is added to the grammar that detects the paragraph tokens from step 1 as a `Name`, e.g., `Name` → `new concept`. The attached rule label is an `ExternalNameBuilder` (see Section 3.2) that produces the name from step 2. This rule ensures that the tokens from step 1, whenever they are encountered again following the definition, produce the name from step 2. Such a `Name` is one of the types of concepts accepted by the `BasicDefinitions` grammar.
4. Finally, instead of the `DEFINE` token that has an unparsed paragraph attached, the token hook returns a `NEWCONCEPT` token with the ID of the name attached. Such a token is recognized by the parser usages of the `newConcept` type in the `BasicDefinitions` type sheet, which is also one of the types of concepts accepted by the `BasicDefinitions` grammar.

Therefore, the definition of a term is represented as a `newConcept` record that has the actual concept name attached as a field, any further references to the defined term simply refer to the name directly.

There is limited support for definitions containing formulas, e.g. `\define{$x$ is in $D$}`. In that case, the name contains the raw LaTeX text of the formula, including the enclosing dollar signs, e.g. `$x$ is in $D$`. Unfortunately, it is rendered in text views exactly in this form. An even bigger issue is that the content of the formulas is not currently looked at, e.g., anything matching the token sequence `FORMULA is in FORMULA` will be recognized as the concept `$x$ is in $D$`. This implies that the actual variable names are lost, e.g., "y is in C" will be stored as `$x$ is in $D$` as well, the actual formulas (i.e., the values attached to the `FORMULA` token recognized as part of a concept)

are not currently stored. An even more elaborate example, which also appears in the test input, is `\define{$R \times C$ table}`. In that case, the formula actually needs to be parsed to recognize the parameters R and C, and pattern matching should be done to avoid false matches. Neither is done at this time. Hence, currently, something like "M table" would be incorrectly accepted as matching this definition. It is planned to address these issues and fully support parametrized concept definitions in the future. It was not possible to do this so far because the formula grammar (see Section 6.5) had to be written first.

The `BasicDefinitions` grammar was designed with the goal to augment it over time to be able to represent more and more mathematical definitions. The plan is to do this interactively by making use of DynGenPar's incremental properties. The combination of that extended `BasicDefinitions` grammar and a grammar for common mathematical idioms, such as the `BasicReasoning` grammar that will be presented in the next section, shall ultimately form the FMathL (Neumaier [70]) language, the future controlled natural language for mathematics.

For his master thesis, Andreas Pichler has developed a Concise type sheet based on `BasicDefinitions` called **`ConceptsFromLatex`** (Pichler [73]). The `ConceptsFromLatex` type sheet extends the `BasicDefinitions` grammar, enabling Concise to parse a text that introduces matrix calculus. In addition to the elementary definitions supported by `BasicDefinitions`, which cover basic set theory and tables, Pichler's `ConceptsFromLatex` type sheet adds types of definitions and statements that allow conveniently defining semirings, natural numbers, and matrices on semirings. The details of the `ConceptsFromLatex` grammar can be found in Pichler [73].

6.4 `BasicReasoning` – A Concise Grammar based on MathNat

This section describes a work-in-progress Concise type sheet for basic mathematical reasoning. The type sheet is based on the **MathNat** (Humayoun & Raffalli [41], Humayoun [40, 39]) language, originally implemented in the Grammatical Framework (GF) programming language by Muhammad Humayoun.

The initial language research for the FMathL project (Neumaier [70]) was done by Peter Schodl and Arnold Neumaier on the latter's lecture notes on Analysis and Linear Algebra in German (Neumaier [69]). They produced a preliminary BNF grammar for the text sentences in the Neumaier [69] lecture notes, without considering the formulas. That work is outside of the scope of this thesis, but can be read in Schodl & Neumaier [85].

The next step of research was then aimed at obtaining a production grammar. The target language in this case was English. Arnold Neumaier and I evaluated several possible ways of achieving that goal. Humayoun's existing MathNat language looked very promising to us. Therefore, we decided to base our next grammar on MathNat.

Due to previous unsuccessful attempts at interfacing the Concise framework with GF (see Section 5.5), Arnold Neumaier and I implemented a version of MathNat as a Concise type

sheet. The grammar was not taken from the source code in MathNat in the GF language, but from the document Humayoun [38]. That document contains the English text of the mathematical idioms in clear text, whereas it is hidden behind an abstract API (the GF resource grammar API) in the GF source code. E.g., Humayoun [38] contains table rows such as

Function	**Type**	**Explanation**
MkHIfProve	CmnStmnt -> ThmStmnt	prove [that] CmnStmnt.

that map directly to the Concise type sheet notation

```
ThmStmnt:
union> ..., MkHIfProve, ...

MkHIfProve:
allOf> #x1:claim=CmnStmnt
#l> prove &[that&] #x1.
```

Although `BasicReasoning` is a working type sheet, the corresponding grammar is not yet complete. For one, the lack of funding for this part of the project shifted the priorities to tasks closer to goals that funding was obtained for. Moreover, we had decided that the user interface allowing interactive incremental rule addition should be completed first. The plan is to extend the grammar incrementally as new constructs are encountered in the input, via the insights gained from our own language research. It shall ultimately be merged with the `BasicDefinitions` grammar from the previous section and combined with the LaTeX formula grammar from the next section to form the FMathL (Neumaier [70]) language, the future controlled natural language for mathematics.

6.5 A Grammar for LaTeX Formulas

This section describes a Concise type sheet that parses LaTeX formulas in natural notation. The goal is to parse formulas written the way a mathematician typically writes them, without requiring special semantic markup such as sTeX (Kohlhase [59]). The grammar is not unambiguous by design, because natural formula notation has some inherent ambiguities.

Prior attempts at semantically parsing LaTeX require writing the LaTeX in a way that can be easily and unambiguously parsed. Approaches include:

- requiring a notation known from programming languages, where, e.g., multiplication always uses the `*` operator – I implemented that concept myself in the `OptProbl` grammar (see Section 4.5) and in the Naproche formula grammar (see Section 6.1), or
- defining dedicated LaTeX macros that (ideally) expand to the usual notation for rendering, but carry semantic markup – an example of that approach is sTeX (Kohlhase [59]).

The big drawback of those approaches is that, by restricting the allowed LaTeX input, they are not able to parse the formulas in mathematicians' existing LaTeX source files without manually converting them all. A tool to automatically or semi-automatically (interactively) convert existing LaTeX formulas to the semantic notation would be desired, but such a tool can also only exist if a parser for existing LaTeX formulas is available. Of course, the authors of LaTeX extensions such as sTeX hope that authors will eventually get used to them and write their formulas directly in them, but this will only happen once those extensions have caught on, so a transition plan is needed in the meantime.

Therefore, I implemented, as a Concise type sheet, a grammar for LaTeX formulas in the notation in which mathematicians typically write them in their papers. For instance, the type sheet deliberately allows implied multiplication, even though there are several cases where it is inherently ambiguous. By design, not all ambiguities are resolved during parsing. Some of them can be resolved in a later stage through semantic analysis, if only one of the syntactically valid interpretations also makes sense semantically. Only the remaining ambiguities must be resolved interactively by the user.

The original plan in Concise was to use DynGenPar only for the natural-language parts of mathematical texts and to leave the formulas to an external parser. The hierarchical parsing functionality of DynGenPar (see Section 2.2.3.5) would also have accomodated such a design. David Langer worked on parsing LaTeX formulas for his diploma thesis (Langer [63]). He produced a parser in C++ based on a simple lexer for LaTeX tags, on the shunting-yard parsing algorithm, and on postprocessing on the parse trees. The details of his algorithm are outside of the scope of this thesis and can be found in Langer [63]. I cleaned up the implementation, integrated the documentation from the diploma thesis into the source code, and translated it into English. I then produced Java bindings using the binding generator SWIG (Beazley [8], Fulton et al. [32]) and integrated them into Concise for testing. I picked SWIG rather than Qt Jambi in this case because the C++ code uses the C++ standard template library (STL), not Qt classes. Unfortunately, the testing uncovered several weaknesses in the implementation, which were ultimately due to the limitations of the shunting-yard algorithm: some of the test cases that were expected to parse did not actually parse successfully, some produced incorrect parse trees because the postprocessing was unable to fix the bad parse trees coming out of the shunting-yard algorithm. Therefore, it was decided to try using the more powerful parsing algorithm DynGenPar already available in Concise. While DynGenPar was not originally designed for formulas, it supports a strict superset of the grammars supported by the shunting-yard algorithm: DynGenPar supports all context-free grammars and some additional extensions, while the shunting-yard algorithm supports only a subset of context-free grammars.

The plan was thus changed to keep the concepts of the lexing algorithm from Langer [63], but to replace the implementation of the lexer and the complete parsing step. David Langer's implementation of the lexer requires to read in the entire input string first, and then performs several transformation steps on the entire list of tokens. I implemented a new lexer (called **FormulaTokenSource**) that operates incrementally on the input and performs the same transformations on the fly. This is more efficient and allows resuming an interrupted parsing process interactively. However, of course, interrupting and resuming

within a token will not work as expected. The revised plan was also discarded, because of the complexity introduced by the lexer: in particular, the conversion of the type sheet to a DynGenPar grammar (see Section 3.2) would have had to take the new token source into account.

The implementation that was finally successful is a scannerless grammar, i.e., using character tokens. It is implemented in the `LaTeXFormulas.cnt` type sheet. The FormulaTokenSource is not used. The scannerless approach does not support transformations such as converting `$\mathbb{Q \subset R}$` to `$\mathbb{Q} \subset \mathbb{R}$` (both render as $\mathbb{Q} \subset \mathbb{R}$), but the Concise team agreed that such a transformation is not needed and that the former, semantically incorrect, version need not be accepted. The big advantage of the scannerless approach is its simplicity. To match LaTeX tags correctly, the same trick based on next token constraints as in the `OptProbl` grammar (see Section 4.5) is used: the lexical category `##tagend` matches either the empty string when not followed by a tag character (which would be part of the tag) or by a whitespace character (because that whitespace must be consumed as per the LaTeX syntax rules), or a nonempty whitespace sequence (which is consumed).

The core of the grammar is based on the expression grammar pattern. Most formulas are expressions of some type. Initially, I attempted to distinguish between different kinds of expressions: statements (boolean expressions), numeric expressions, and set expressions. Unfortunately, it turned out that such a distinction does not work out well. For simple expressions such as x or $P(x)$, it cannot be determined without surrounding context whether they are booleans, numbers, or even sets. For sets, there is additionally the possibility to do elementwise arithmetic, such as $2\mathbb{N}$ or $-\mathbb{N}$, which would require duplicating all the relevant arithmetic operators for both expressions and sets. Therefore, I decided to have a single expression grammar, and to leave type analysis and elimination of parses that do not make sense when considering the types to a subsequent semantic analysis, to be implemented in the near future.

Operator priorities. This implies that there is a global list of operator priorities, which required careful tuning to fit mathematicians' expectations. In order of decreasing priority, the operators are as follows:

1. atomic expressions: parenthesized or LaTeX-brace-enclosed expressions, variables, constants (including set constants), function calls, tuples, vector or matrix subscripts, intervals, braced sets (enumerated, e.g., $\{x, y, z\}$, or defined by a condition, e.g. $\{x \in \mathbb{R} \mid x > 0\}$) expressions grouped by LaTeX tags (`\frac`, `\sqrt`, `\binom`, `\overline`, `\bar`), absolute values and norms, and the ellipsis (treated as a special constant or variable),
2. powers and other postfixes: factorials, superscripts (treated as the exponent of a power), superscript (adjoint) or subscript (cumulation point) asterisks, prime, double prime, and triple prime,
3. nabla expressions: directional derivatives, gradients, divergences, and curls, when denoted with the nabla (∇) symbol,
4. functor calls, i.e., function calls that do not require explicit parentheses, only a prefix functor: `\sin`, `\cos`, `\tan`, `\cot`, `\sec`, `\csc`, `\sinh`, `\cosh`, `\tanh`, `\coth`,

`\arcsin`, `\arccos`, `\arctan`, `\exp`, `\ln`, `\log`, `\lg`, `\arg`, `\det`,

5. simple products: dot product ($\cdot$), cross product ($\times$), asterisk product ($*$), implied multiplication, slash division (/)
6. special products: wedge ($\wedge$) and circle ($\circ$) operators
7. summations and similar expressions: limits (including lim inf and lim sup), Σ sums, Π products, $\bigcap$ intersections, $\bigcup$ unions, integrals
8. simple sums: unary and binary versions of $+$, $-$, $\pm$, and $\mp$,
9. ranges in MATLAB syntax: *start* : *end* or *start* : *increment* : *end*,
10. set complements, i.e., the set difference operator $\setminus$,
11. set intersections ($\cap$),
12. set unions ($\cup$),
13. equality, inequality, and divisibility relations: $=$, $>$, $<$, $\geq$, $\leq$, $\neq$, $\approx$, $\subset$, $\supset$, $\subseteq$, $\supseteq$, $\subsetneq$, $\supsetneq$, $|$,
14. element containment statements: $\in$, $\notin$,
15. boolean not ($\neg$),
16. boolean and ($\wedge$),
17. boolean or ($\vee$),
18. implications ($\Rightarrow$, $\Leftarrow$),
19. equivalences ($\Leftrightarrow$),
20. quantifier expressions: $\forall$, $\exists$, $\exists!$,
21. global operator expressions: the `\choose` operator, which is semantically the same as `\binom`, but must be parsed with a completely different priority to match the behavior of LaTeX.

At this stage, no attempt is made at determining precise semantics, such as whether a `\binom` or `\choose` is actually a binomial coefficient (as both macro names would imply) or just a two-element vector. That issue is addressed further down in this section.

A construct that frequently appears in mathematicians' common usage is **informal lists**: comma-separated lists that are typically a shortcut for repeating the formula, once for every term in a list. An example is the x, y in $x, y \in \mathbb{R}$. A challenge is that the list can appear at different priority levels in the expression, e.g., the same formula can equivalently be written as $x \in \mathbb{R}, y \in \mathbb{R}$. In principle, such an informal list could appear at any of the priority levels above, but that would introduce a tremendous amount of ambiguity. Thus, as a compromise, I looked closely at several examples to decide on the priority levels at which best to allow such lists. I determined two suitable levels:

- the level of simple products (level 5), allowing, e.g., $2x, 3y \in \mathbb{N}$, and of course anything with higher priority (levels 1 to 4), hence this also covers the $x, y \in \mathbb{R}$ example, and
- the level of complete expressions (level 21), covering the other example $x \in \mathbb{R}, y \in \mathbb{R}$.

Since the grammar, for the reasons explained above, does not distinguish between boolean and numeric expressions, $x, y \in \mathbb{R}$ is ambiguous at syntactic level (x could also be a complete expression). Semantic type analysis is needed to discard the incorrect alternative. There are, however, cases that are genuinely ambiguous even if semantic analysis is done, such as $a < x, y < b$, which could mean either "$a < x < b$ and $a < y < b$" or "$a < x$ and $y < b$". Some ambiguities are eliminated by restricting expressions to non-lists in some contexts, e.g., expressions in contexts that are inherently lists, such as tuples, must not themselves be informal lists. But the aforementioned ambiguous examples cannot be disambiguated in that way.

Common LaTeX usage also includes various forms of whitespace in formulas. There is both

- **invisible whitespace**, i.e., whitespace in the LaTeX source code that is not rendered in the output, and
- **visible whitespace**, i.e., LaTeX directives producing whitespace in the output.

The LaTeXFormulas grammar allows invisible whitespace almost everywhere. In most places (e.g., around operators), visible whitespace is also permitted. Both are currently ignored completely. That implies the assumption that the whitespace has no semantic meaning. For invisible whitespace, this is almost always the case. Visible whitespace can actually have a meaning depending on the context, which is not currently recognized. E.g., often, an expression within parentheses, preceded by wide whitespace, is either an explanatory comment or a condition rather than an implicitly multiplied expression or a function call argument. The current grammar mistakenly treats it as implied multiplication and/or as a function call. Handling the semantics of whitespace constitutes a potential future extension to the grammar. The `\displaystyle` formatting directive is also treated as visible whitespace because it behaves essentially the same way. It cannot be treated as invisible whitespace because the latter is allowed between the `_` character and the subscript or between the `^` character and the superscript, but `\displaystyle` is not.

The start category of the LaTeXFormulas grammar is a linked list of formulas, corresponding to a comma-separated list in the input. Every formula is one of the :

- An expression, as per the expression grammar defined above. The expression cannot be an informal list as a whole (because the list is already treated by the linked list of formulas), but may of course contain informal lists.
- A continued relation, i.e., a special relation expression where the left hand side is missing (assumed to come either from the line above or from informal text). The relation starts with the relation operator, any relation operator except $|$.
- A definition, of the form $lhs := rhs$.
- A function declaration, of the form $function : domain \to range$.
- A convergence statement, of the form $var \to towards$. It can semantically also be an incomplete function declaration $domain \to range$ or an incomplete function definition $var \to value$.

The biggest difficulty with parsing natural LaTeX notation is that it is inherently ambiguous. Even frequently used constructs can be ambiguous, e.g.:

- Is $a(x+1)$ an implied multiplication or a function call?
- Is A_{12} the matrix subscript $A_{1,2}$, or is A a vector and A_{12} its twelfth element, or is it the twelfth row or column of the matrix A?
- Is A_{ij} the matrix subscript $A_{i,j}$ or the vector subscript $A_{i \cdot j}$ (with implied multiplication)?
- Is $x \wedge y$ a wedge product or a boolean AND? This must be distinguished at the syntactic level because they have very different priorities.
- The typical notation for absolute values and norms is ambiguous because it does not distinguish between the opening and closing tag. E.g., does $|xy| - |x||y|$ mean $\mathrm{abs}(x \cdot y) - \mathrm{abs}(x) \cdot \mathrm{abs}(y)$ or $\mathrm{abs}(x \cdot y \cdot \mathrm{abs}(-\mathrm{abs}(x)) \cdot y)$? The "divides" relation can introduce additional ambiguities in some cases. E.g., the example can be parsed syntactically as $\mathrm{abs}(x \cdot y \text{ divides } -\mathrm{abs}(x) \text{ divides } y)$, but in this case, a type analysis will be able to reject the absolute value of the relation chain.

Other constructs, even frequently used ones, that look clear at the syntactic level have ambiguous or unclear semantics, e.g.:

- Is x^i an exponentiation (x to the ith power) or a superscript index (the ith element of the vector x)?
- Is f' the derivative of f or just a different function or variable that may or may not be directly related to f?
- Is $\binom{x}{y}$, written as either `$\binom{x}{y}$` or `$x \choose y$`, a binomial coefficient or a column vector with two elements?
- What exactly does x^* mean? Most commonly, it is some form of conjugate or adjoint (e.g., the conjugate transpose of a complex vector), but the notation can have several other meanings.
- Is $\sin^2 x$ the common shorthand for $(\sin x)^2$ or does it stand for the iterated sine $\sin(\sin x)$?

By design, the LaTeXFormulas grammar does not attempt to resolve these ambiguities at the syntactic level. For the ambiguities requiring a syntactic distinction (those in the first of the above lists), both (or in general, all) possible parse trees are returned. For the ambiguities in the second list, one parse tree is returned, but no attempt is made at determining the precise semantics during the parsing step. In both cases, the correct semantics must be determined by a later processing step.

In the current implementation, any ambiguities produced by the parser must be interactively resolved by the user. Several of them could, however, be automatically resolved by a semantic analysis. Automated disambiguation techniques planned to be implemented in the near future include:

- type analysis: If it can be determined, e.g., that a is a scalar constant, then it is clear that $a(x+1)$ cannot be a function call and thus is an implied multiplication.
- document-specific notational preferences, to be specified globally by the user: E.g., a paper or book on boolean logic will not contain any wedge products, and thus $x \wedge y$ is necessarily a boolean AND.

Those techniques can also be useful in combination, e.g., if A is known to be a matrix, and if the notational preferences do not allow the A_i notation for the row $A_{i,:}$ (the ith row of A), it follows that A_{12} and A_{ij} must be matrix subscripts. They can also be used to give clear semantics to constructs with unclear semantics such as f'.

For any ambiguities that remain after the semantic analysis (i.e., currently, all that the parser produced), all alternative parse trees are stored in the semantic memory. Before performing any further transformations that rely on an unambiguous parse tree, the ambiguities must be resolved interactively. (If that is not done, it is undefined which of the alternatives is the primary alternative seen by ambiguity-unaware postprocessing steps.) This interactive resolution can be done through the Concise text view. The text view displays a text representation of a Concise record, in this case, of a formula. The `text.EN.write` usage is used to produce the text. If a portion of the text is ambiguous, that portion is highlighted in red. The displayed text corresponds to the primary alternative, which is picked arbitrarily by the parser, but that is typically not an issue because the output is usually the same ambiguous text as the input. Clicking on the highlighted text selects it and opens a popup in which one of the alternatives can be chosen. To let the user know which is which, the alternatives are linearized using a dedicated `disambiguation.EN` usage designed to produce unambiguous output. Ambiguities can also be nested. In that case, the ambiguous portion will be highlighted in red even in the popup, and linearized using the usual `text.EN.write` usage that leaves it deliberately ambiguous. However, clicking on the nested ambiguity does not open a nested popup. An example of such a popup is shown in Figure 6.1. Clicking on an alternative from the popup deletes all other alternatives for this ambiguity from the record in the semantic memory, so that any further processing sees only the selected alternative, and refreshes the text view. Thus, the resolved ambiguity is no longer highlighted. Nested ambiguities are kept and become highlighted and clickable in the text view, allowing the user to also resolve them.

```
|xy|-|x||y|=0
\abs{xy|-|x||y}=0
\abs{x \impmul y}-\abs{x} \impmul \abs{y}=0
```

Figure 6.1: Disambiguation popup from the Concise text view

To test the formula parser, I extracted the formulas from the LaTeX manuscripts, which were provided to me by the authors, of two real-world German text books (Neumaier [69], Schichl & Steinbauer [83]). In both cases, the success rate was around 71%. The detailed results are presented in Section 7.3.

Once the formula is parsed into a parser-oriented Concise record and the ambiguities resolved, it needs to be converted to an internal representation more suited for computation, reasoning, and conversion to other formats. E.g., as all parser-oriented type systems, the LaTeXFormulas type system treats variable references as strings containing the variable name. In contrast, in the internal representation, variable references shall point directly to a unique object representing the variable. In addition, in the internal representation, all constructs must have clear semantics, e.g., f' must be represented differently if it is a

derivative than if it is a separate function or variable name. An initial type system for expressions suitable for computation and reasoning was designed by Peter Schodl. That type system will have to be extended to cover all the concepts that the LaTeXFormulas grammar can parse. For an early, very restricted, version of the LaTeXFormulas grammar, I wrote a record transformation sheet called `LaTeXFormulas.cnrt` converting records in the LaTeXFormulas type system to Peter Schodl's type system for expressions. Work on that record transformation sheet was put on hold, in order to focus on the grammar first, and pending the design of the extended internal representation. It is planned to extend it together with the type system for expressions.

As a final step, having the formula in an internal representation will allow the conversion into one of many output formats. Suggested general output formats include

- sTEX (Kohlhase [59]), i.e., semantically marked up LATEX,
- the input syntax for computer algebra systems such as Mathematica or Maxima, and
- code to evaluate the expression efficiently in a programming language.

Note that we do not perceive the LaTeXFormulas grammar as competition for sTEX. There will always be a place for unambiguous semantic markup. By using the LaTeX-Formulas grammar to convert legacy LATEX markup to sTEX semantic markup, the two approaches will complement each other.

In addition to the above general output formats, for certain kinds of formulas and for certain applications, specialized output formats are useful. In particular, for optimization problems, it is planned to convert them to

- the RobustOptProb type system from Section 4.6,
- modeling languages: AMPL (Fourer et al. [27], AMPL Optimization inc. [2]), GAMS (Brooke et al. [9], McCarl et al. [65], GAMS Development Corp. [33]), Modelica (Mattsson et al. [64], Modelica Association [68]), and
- the DAG (directed acyclic graph) representations of GloptLab (Domes [24, 23]) and COCONUT (Schichl [82], Neumaier & Schichl [71]).

Implementing these output formats is planned once the internal representation and the record transformation from the LaTeXFormulas type system to it are complete.

Chapter 7

Conclusion: Performance Results, Achievements, Extensions

This chapter wraps up the thesis by first presenting some practical results obtained with DynGenPar, then summarizing what was achieved through DynGenPar, then proposing possible future extensions.

The aim of this chapter is to evaluate how DynGenPar performs in practical settings, in order to conclude the thesis with some practical results. The parsing speed of DynGenPar is measured in a few benchmarks and compared to the state of the art. Another performance metric that is evaluated is the success rate on a set of practical inputs. That obviously depends on the individual grammar. The grammar whose success rate is considered in this chapter is the LaTeX formula grammar from Section 6.5, which is already ready to be used on unmodified formulas from the LaTeX manuscripts of two university-level introductory mathematics text books.

The chapter is composed of five sections. The first section presents some performance results for DynGenPar, first published in Kofler & Neumaier [56], but updated for the current releases of the involved software. The results show that the performance is competitive with the state-of-the-art parsers GNU Bison (Free Software Foundation [28]). The comparison was performed on the Naproche grammar (Cramer et al. [17], Koepke et al. [46]) (Section 6.1). The results also show that DynGenPar is competitive with the Grammatical Framework (GF) (Ranta [77, 78], Ranta et al. [79]) on the example grammar *Phrasebook* that is included in GF releases. The second section discusses the performance of dynamic rule addition. It presents timings obtained on the `OptProbl` (Section 4.5) and `BasicDefinitions` (Section 6.3) grammars. The third section presents results obtained using the grammar for LaTeX formulas (see Section 6.5) on the complete collection of formulas in the original notation from two university-level introductory mathematics textbooks (Neumaier [69], Schichl & Steinbauer [83]). It shows the success rates of the grammar on those formulas. The fourth section summarizes what was achieved through DynGenPar. Finally, the fifth section proposes some possible future extensions to the algorithm and the implementation.

7.1 Performance Benchmark

This section is based on Kofler & Neumaier [56], but the results presented below are updated. The tests were repeated in April 2017 with the latest stable releases of the parsing software and on newer hardware. All results reported below are from the April 2017 tests.

I performed two performance comparisons of DynGenPar with other state-of-the-art parsers: a benchmark of DynGenPar against GNU Bison (Free Software Foundation [28]) on a grammar for the Naproche language (Cramer et al. [17], Koepke et al. [46]) (see Section 6.1), and a comparison of DynGenPar against the Grammatical Framework (GF) (Ranta [77, 78], Ranta et al. [79]) on the GF *Phrasebook* example grammar.

7.1.1 Benchmark of DynGenPar vs. Bison on the Naproche Grammar

I compared the speed of my implementation to the well-established GNU Bison (Free Software Foundation [28]) parser on the hierarchical (two-layer) grammar I devised for the Naproche language (Cramer et al. [17], Koepke et al. [46]) (see Section 6.1): There are 2 context-free grammars, one for text and one for formulas, each using a lexer based on Flex (Flex Project [26]). In one version of my Naproche parser, the 2 context-free grammars are processed with Bison 3.0.4 (using its support for GLR parsing), in the other with DynGenPar release 10. I measured the times required to compile the code to an executable (using GCC with `-O2` optimization), to convert the grammar rules to the internal representation (GLR tables for Bison, initial graphs for DynGenPar), and to actually parse a sample input (representing the Burali-Forti paradox in Naproche). Note that for Bison, the grammar conversion is done before the compilation, so, when working with dynamically changing grammars, the compilation time also has to be considered, whereas DynGenPar can convert grammars at runtime. My test machine was a desktop computer with a Core i7-2600K ($2 \times 4 \times 3.40$ GHz) and 16 GiB RAM running Fedora 25 x86_64. Only one thread was used in the tests because the algorithms are single-threaded. I used the system versions of GCC (6.3.1), Flex (2.6.0), and Bison (3.0.4). For each measurement, I averaged the execution times of 100 tests and took the median of 3 attempts. My results are summarized in Table 7.1.

time	compilation	grammar conversion	grammar update**	parsing
Bison	1089 ms	153 ms*	1242 ms	1.60 ms
DynGenPar	8851 ms	5.34 ms	5.34 ms	9.37 ms***
ratio	8.1	0.034	0.0043	5.9

* ... at compile time, thus requires recompilation
** ... estimated total time to process a grammar update (includes recompilation for Bison)
*** ... total execution time of 14.71 ms minus grammar conversion time

Table 7.1: Benchmarking results on the Naproche grammar (input: Burali-Forti paradox)

I conclude that, while Bison is around 8 times faster at compilation and around 6 times faster at pure parsing, DynGenPar is much faster at adapting to changed grammars. The slower compilation of DynGenPar is not an issue because DynGenPar need not be recompiled if the grammar changes. On the other hand, the time required to compile modified grammars makes Bison entirely unsuitable for applications where the grammar can change dynamically. Even if Bison were changed to allow loading a different LR table at runtime, it would still take 29 times longer than DynGenPar to process my fairly small two-layered grammar, and I expect the discrepancy to only grow as the grammar sizes increase. (Moreover, DynGenPar can handle dynamic rule addition, so in many cases even the 5.34 ms for grammar conversion can be saved. The performance of dynamic rule additions is discussed in the next section.) In the worst case, where we have a new input for an existing grammar and do not have the initial graph in memory, DynGenPar (14.71 ms) is still only 9 times slower than Bison (1.60 ms), even though the latter was optimized specifically for this usecase and DynGenPar was not.

7.1.2 Benchmark of DynGenPar vs. the Grammatical Framework (GF)

I also benchmarked my support for PGF grammar files (Angelov et al. [4]) produced by the Grammatical Framework (GF) (Ranta [77, 78], Ranta et al. [79]) (see Section 5) against the software provided by the GF project itself. GF works by first compiling grammars into the binary PGF format. Those PGF files are then loaded together with the input into a program called the **PGF runtime**, which interprets such a PGF grammar and serves as a parser for that grammar. DynGenPar, with its PGF support, acts like a PGF runtime. Two other PGF runtimes are provided by the GF project itself: the original one written in Haskell and a new runtime written in C. I benchmarked DynGenPar against those two PGF runtimes. (I used DynGenPar release 11 and the GF 3.8 release.) As an example grammar, I used GF's *Phrasebook* example, which was the one explicitly documented as being supported by the initial versions of GF's C runtime, and which I kept as the test case to be able to compare with my earlier results from Kofler & Neumaier [56]. As input, I used the sample sentence *See you in the best Italian restaurant tomorrow!*, a valid sentence in the *Phrasebook* grammar. (I also tried parsing with the full English resource grammar, but DynGenPar would not scale to such huge grammars and did not terminate in a reasonable time.) I measured the time to produce the syntax tree only, without outputting it. The tests were run on the same Core i7 desktop computer as above. Again, for each measurement, I averaged the execution times of 100 tests and took the median of 3 attempts. My results are summarized in Table 7.2.

I conclude that DynGenPar is within an order of magnitude in speed compared to both GF runtimes on practical application grammars.

Regarding the faithfulness of the grammar, note that, at the time of testing, both GF runtimes incorrectly accepted the input *Where is an restaurant?* (should be *a restaurant*), whereas DynGenPar can enforce the next token constraint. (My testing confirmed that this known limitation of the GF runtimes still persists in GF 3.8.)

	parsing time
GF Haskell runtime	43.4 ms
GF C runtime	17.8 ms
DynGenPar	121.8 ms

Table 7.2: Benchmarking results on the GF *Phrasebook* grammar (input sentence: *See you in the best Italian restaurant tomorrow!*)

Comparing to the older (2012) results from Kofler & Neumaier [56], one can see that the newer releases of both the GF Haskell runtime and DynGenPar take longer on the GF benchmark than the older releases that were tested in 2012, even on a faster computer. This is because the *Phrasebook* grammar itself and the resource grammar it relies on have changed, too. I have tried the current DynGenPar release 11 on the version of the *Phrasebook* grammar (`Phrasebook.pgf` file) tested in 2012. It takes 66.7 ms on a Core i7 at 3.40 GHz per core. Release 2 took 81 ms on a Core 2 Duo at 2.40 GHz per core (Kofler & Neumaier [56]). Unfortunately, I cannot test the current GF runtimes on the old *Phrasebook* grammar because the PGF format has changed. (DynGenPar still also supports the old format version.) Recompiling the old source code of the *Phrasebook* grammar with the current GF compiler to obtain a PGF file in the new format would also not reproduce the same grammar because the resource grammar has changed significantly.

7.2 Performance of Dynamic Rule Addition

In order to evaluate the performance of dynamic rule addition, I made some performance measurements on the `OptProbl` grammar from Section 4.5. As a sample input, I used the simple input problem

```
\min (1/2)*x^{2}\\
\st x \in [-1,1]
```

and I prepended a variable number of dummy `\newcommand` commands of the form

```
\newcommand{test1}[0]{}
\newcommand{test2}[0]{}
\newcommand{test3}[0]{}
⋮
```

to it. Each `\newcommand` command adds two rules to the grammar. I measured the time required to parse the input and produce an output DAG in the COCONUT format. (The output DAG was always the same because the defined dummy commands are not used anywhere.) My test machine was a desktop computer with a Core i7-2600K ($2 \times 4 \times$ 3.40 GHz) and 16 GiB RAM running Fedora 25 x86_64. Only one thread was used in the tests because the algorithm is single-threaded. I used the system version of GCC (6.3.1).

number of `\newcommand` commands	0	1	10	100	1000
no rule addition (`\newcommand` ignored)	10 ms	11 ms	16 ms	80 ms	2166 ms
dynamic rule addition (DynGenPar release 11)	10 ms	11 ms	20 ms	116 ms	2462 ms
dynamic rule addition (DynGenPar release 10)	10 ms	11 ms	22 ms	307 ms	15310 ms
rule addition with full regeneration of the initial graph	10 ms	11 ms	23 ms	338 ms	16064 ms

Table 7.3: Parsing times on the `OptProbl` grammar, using a simple test problem with a variable number of dummy `\newcommand` commands prepended, comparing two versions of DynGenPar (release 10 vs. release 11)

For each measurement, I averaged the execution times of 100 tests and took the median of 3 attempts. My results are summarized in Table 7.3.

The only difference between the older, slower DynGenPar release 10 and the newer, faster DynGenPar release 11 is that I modified the method performing dynamic rule addition. (Therefore, in Table 7.3, the results for dynamic rule addition are given for both versions, whereas the other cases were measured only with release 11.) In release 10, the cache of neighborhoods (as defined in Section 2.1.2) was completely cleared when a rule was added. This made the dynamic rule addition itself really fast (often less than 10 microseconds), but the overall performance was hurt by the recomputation of the neighborhoods. In release 11, the dynamic rule addition now performs a quick check whether a cached neighborhood needs to be invalidated. Only neigborhoods that need to be invalidated are removed from the cache. This significantly improves the overall performance, up to a factor of 6.22 in the largest tested instance (1000 `\newcommand` commands).

One can compute from the numbers in Table 7.3 that the dynamic addition of a rule can be done in about 150 to 200 microseconds, whereas regenerating the initial graph and all caches takes up to 14 milliseconds per instance. (In my test, the initial graph was regenerated only once for both rules in each `\newcommand` command.) The larger the grammar grows, the longer it takes to regenerate the initial graph and the neighborhoods. That explains why the average time to rebuild the initial graph and the neighborhoods is only around 0.7 milliseconds for 10 `\newcommand` commands, around 2.6 milliseconds for 100 `\newcommand` commands, and around 13.9 milliseconds for 1000 `\newcommand` commands. In the last case, the dynamic rule addition is almost 50 times faster (because the rules are added in pairs, 100 times otherwise).

I also made some performance measurements on the `BasicDefinitions` grammar from Section 6.3. As a sample input, I created a simple LaTeX document skeleton with a single section containing a variable number of paragraphs of the form

> A **domain1** D associates with every object x a statement $x \in D$.

where the term in bold is marked up as a definition using the `\define` macro (see Section 6.2) and where the number is incremented each time: **domain1**, **domain2**, **domain3**,

etc. In the case with zero definitions (and only in that case), since the `TextDocument` type system does not allow empty sections, the section contains a dummy paragraph consisting of an empty raw comment (see Section 6.2): `\raw{}`.

Each definition adds a rule to the grammar. As explained in Section 6.2, every paragraph is parsed to a separate parse tree, but additions to the grammar are retained from one paragraph to the next. Thus, the grammar grows incrementally with each encountered definition. I measured the time required to process all paragraphs, i.e., iterating over the paragraphs and, each time:

1. parsing the input record (unparsed paragraph),
2. producing an output record (parsed paragraph), and
3. replacing the input record with the output record in the `TextDocument` record.

My test machine was the same Core i7 desktop computer as above. I used the DynGenPar release 11 Java bindings on a development snapshot of Concise and the system version of Java (OpenJDK 1.8.0_121-b14). Again, for each measurement, I averaged the execution times of 100 tests and took the median of 3 attempts. My results are summarized in Table 7.4.

number of definitions	0	1	10	100	1000
parsing time	7 ms	9 ms	31 ms	228 ms	2298 ms

Table 7.4: Parsing times on the `BasicDefinitions` grammar, using a simple synthetic test document containing a variable number of definitions

From the last two results, one can extrapolate that the combination of parsing a paragraph of the form given above and adding a rule to the grammar takes only approximately 2.3 milliseconds altogether. The scalability to 1000 definitions is significantly better than in the `OptProbl` case because no long lists are built in this case. Instead, each paragraph produces a separate parse tree. Therefore, only the grammar grows linearly with the number of definitions, the parse trees do not.

7.3 Test Results on Real-World LaTeX Formulas

In Section 6.5, I presented a Concise type sheet for LaTeX formulas in natural notation. To test the resulting DynGenPar grammar on real-world formulas, I extracted the list of all formulas from the LaTeX manuscripts, which were provided to me by the authors, of the German text books

- *ALA* (*Analysis und lineare Algebra*, Neumaier [69]), covering the first year content of real analysis and linear algebra courses in typical European undergraduate mathematics curricula, and
- *Einf* (*Einführung in das mathematische Arbeiten*, Schichl & Steinbauer [83]), a general introduction to university-level mathematics for first-semester mathematics students.

Multiple instances of the exact same formula have been deleted, so all formulas are unique in the final two lists.

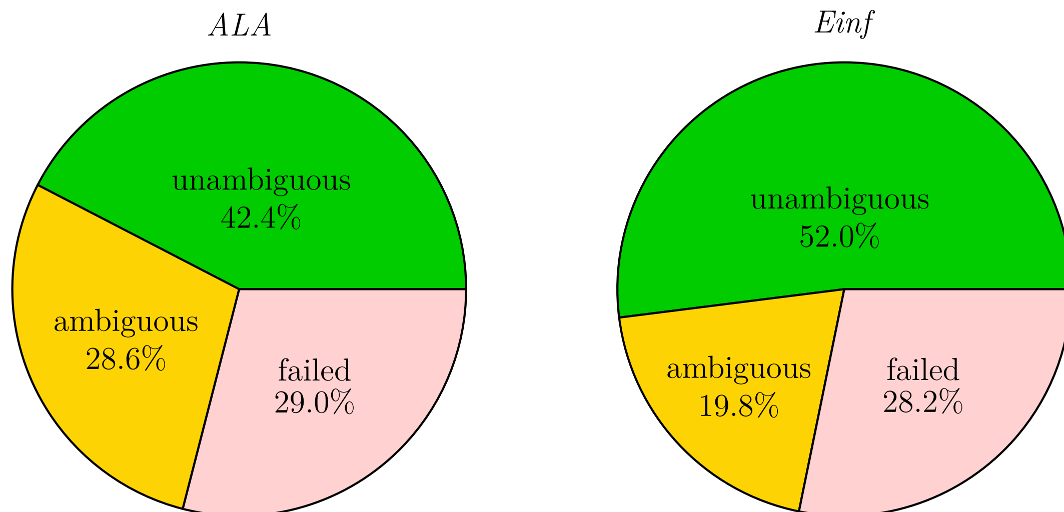


Figure 7.1: LaTeXFormulas success rates on *ALA* (left) and *Einf* (right)

The results are illustrated in Figure 7.1. For *ALA*, out of 9504 unique formulas, 4030 (42.4 %) were successfully parsed without ambiguities, 2715 (28.6 %) were successfully parsed with ambiguities, and 2759 (29.0 %) failed to parse. Thus, 71.0% of the formulas from *ALA* were successfully parsed, and of these, 59.7% were unambiguous. For *Einf*, out of 5886 unique formulas, 3063 (52.0%) were successfully parsed without ambiguities, 1166 (19.8%) were successfully parsed with ambiguities, and 1657 (28.2%) failed to parse. Thus, 71.8% of the formulas from *Einf* were successfully parsed, and of these, 72.4% were unambiguous.

An example of an ambiguous formula is the expression $|xy| - |x||y|$. The resulting Concise record representation of the parse tree is shown in Figure 7.2. One can see that there is one ambiguity at the top level, and the absolute value expression in the second alternative has itself two alternatives. This results in a total of three possible interpretations, namely:

1. `\abs{x \impmul y}-\abs{x} \impmul \abs{y}`,
2. `\abs{x \impmul y \impmul \abs{-\abs{x}} \impmul y}`, and
3. `\abs{x \impmul y \divides -\abs{x} \divides y}`.

The notation used here is the one used by the text view for interactive disambiguation (see Figure 6.1 in Section 6.5).

One can see that, while more work is needed to extend the grammar and to resolve ambiguities, my current grammar already achieves a success rate of over 70%.

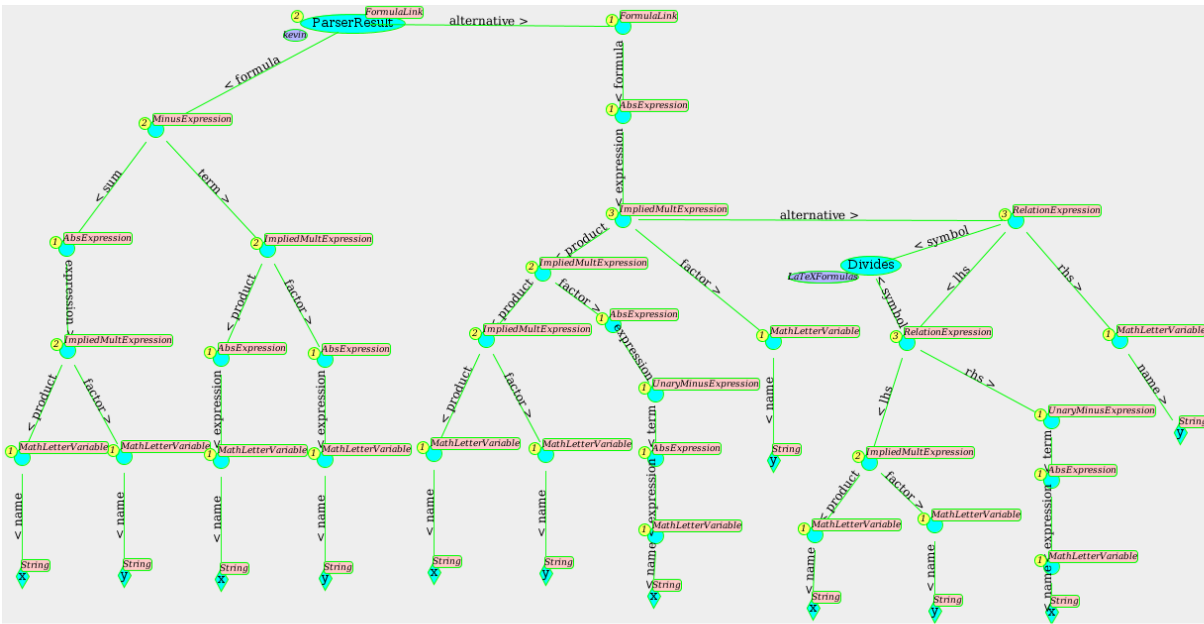


Figure 7.2: Concise record for the ambiguous formula $|xy| - |x||y|$, showing the three possible interpretations

7.4 Achievements

This thesis introduced DynGenPar, a dynamic generalized parser for common mathematical language. It presented its requirements, the basics of the algorithm and the tweaks required for an efficient implementation. It then described several applications DynGenPar is already used for, both for formal languages and for controlled natural language grammars, resulting in some important achievements.

The DynGenPar algorithm keeps efficiency by combining enough bottom-up techniques to avoid trouble with left recursion with sufficient top-down operation to avoid the need for states and tables. The initial graph ensures that the bottom-up steps never try to reduce unreachable rules, which is the main inefficiency in existing tableless bottom-up algorithms such as CYK (Kasami [43], Younger [101]). It can also be dynamically updated. This allows dynamic rule changes. Conflicts are not fatal because the DynGenPar is exhaustive and designed to keep its efficiency even in the presence of conflicts.

Through interaction with Concise, grammars can be imported at runtime. The grammars are given as Concise type sheets, which are automatically converted to grammar rules suitable for DynGenPar, which can then parse documents using the converted grammar. Therefore, user-written rules can be fully read into the parser at runtime, rather than hardcoding them as C++ or Java code or compiling them to some other precompiled format. Concise type sheets represent a user-friendly mechanism for specifying rules that can be easily converted to the DynGenPar internal representation. This feature is thus an ideal showcase for the dynamic properties of DynGenPar. Concise type sheets are typically scannerless grammars, thus DynGenPar has special support for scannerless parsing. In particular, DynGenPar supports next token constraints, which can be used

to express maximally-matched character sequences.

DynGenPar is the parser of choice in Concise. It is successfully used to parse several formal languages. In particular, Concise type sheets are parsed using DynGenPar. There is both a handwritten DynGenPar grammar in C++ that is used in practice because of bootstrapping concerns and a Concise type sheet. The type sheet for type sheets can bootstrap itself and passes bootstrap comparison, i.e., it can parse itself, the obtained grammar can again parse itself etc., and the output from all stages corresponds. Concise code sheets, the programming language in Concise, are also parsed with DynGenPar using a Concise type sheet.

A commonly recurring task in Concise is to convert a record from one type system to another. In particular, parsing with DynGenPar can only produce tree-structured records. It is often desirable to produce an internal representation where variable names are replaced by direct pointers to the actual variable objects. For this task, I designed a domain-specific record transformation language and implemented it as a Concise type sheet. The record transformation sheets are parsed with DynGenPar and executed by Concise.

Another formal language DynGenPar is successfully used on is `ChemProcMod`, a specialized modeling language for chemical processes designed to follow the conception of the chemical engineer rather than of a mathematician or computer scientist. This makes the `ChemProcMod` language significantly more convenient to use for this field of application than a generic modeling language like Modelica (Mattsson et al. [64], Modelica Association [68]).

I also implemented a DynGenPar grammar for a language I called *robust AMPL* as a Concise type sheet. *Robust AMPL* is mainly a subset of the AMPL (A Mathematical Programming Language) (Fourer et al. [27], AMPL Optimization inc. [2]) modeling language for optimization problems. However, *robust AMPL* was extended to allow intervals wherever AMPL normally expects a number. The accepted subset of the AMPL language is growing over time. This allows inputting optimization problems into Concise. When requested by the user, the input is done rigorously with outward rounding, a feature not supported by the official AMPL implementation.

DynGenPar can also handle languages where rules are added to the grammar at runtime. A formal language where this is the case was developed as a working proof of concept: `OptProbl`. The `OptProbl` language is a subset of LaTeX that supports `\newcommand`. The DynGenPar-based `OptProbl` parsing application converts that input to a textual DAG representation. The language demonstrates how dynamic rule addition, originally designed for natural language, can also be put to practical use elsewhere.

The DynGenPar algorithm also supports a type of grammars that is not context-free: parallel multiple context-free grammars (PMCFGs) (Seki et al. [88]). This feature is used in the implementation to import PGF (Portable Grammar Format) (Angelov et al. [4]) grammar files produced by the Grammatical Framework (GF) (Ranta [77, 78], Ranta et al. [79]), a compiled representation of PMCFGs. A GF-compatible lexer is also provided. Thus, natural language grammars described in the GF language can be compiled to PGF files using the GF compiler and reused with DynGenPar.

The ultimate target of DynGenPar research is natural language. Moving towards that

goal, DynGenPar was used on a few controlled natural languages. The first such language is Naproche (Cramer et al. [17], Koepke et al. [46]), a controlled natural language for mathematical logic. The Naproche parser using DynGenPar is implemented using another feature of the DynGenPar implementation: hierarchical parsing. There is a text parser and a subordinate formula parser, both using Flex (Flex Project [26]) and DynGenPar. For comparison, a version using two Bison (Free Software Foundation [28]) grammars was also implemented. DynGenPar is only 6 to 8 times slower at raw parsing than Bison, and several orders of magnitude faster at converting the grammar into something usable for parsing.

Another controlled natural language parsed using DynGenPar is the `BasicDefinitions` language, a Concise type sheet that serves as a proof of concept for the handling of mathematical definitions. As in the `OptProbl` grammar, encountered definitions trigger a parse action that dynamically adds a rule at runtime. Only, in this case, the added rule defines a natural language term rather than a LaTeX command.

Finally, a controlled natural language that is the subject of ongoing research is `BasicReasoning`, a work-in-progress Concise type sheet for Humayoun's MathNat (Humayoun & Raffalli [41], Humayoun [40, 39]) language. The plan is to extend that grammar incrementally via the insights gained from our own language research.

Most recently, the natural language parsing goals for DynGenPar were extended to include not only text, but also formulas. (Originally, it was planned to delegate that task to a dedicated formula parser by Langer [63], which never reached production level.) I implemented a Concise type sheet to parse LaTeX formulas in natural notation, the way a mathematician typically writes them. The grammar is scannerless. I used the full power of DynGenPar, including next token constraints (to correctly detect the end of a LaTeX tag) and PMCFG rules (to handle matched redundant braces without complicating the resulting parse tree).

I also developed a `TextDocument` toolchain that leverages the existing LaTeXML (Miller [67]) tool, Concise and DynGenPar to automatically process complete LaTeX documents. The toolchain 1. converts the LaTeX input to XML using LaTeXML, 2. turns the XML into a Concise record (of type `TextDocument`) representing the document structure, and 3. sends every paragraph to DynGenPar to convert the document structure record to a semantic one, using a grammar such as `BasicDefinitions` or `BasicReasoning`. Optionally, it can also send every formula to DynGenPar using the LaTeX formula grammar.

The advancements made are also evidenced by a performance benchmark, showing that the efficiency of DynGenPar compares favorably to the state of the art.

7.5 Future Extensions

There is still room for even more features, which will advance the state of the art further towards the goal of computerizing a library of mathematical knowledge:

- context-sensitive constraints on rules: Currently, my implementation supports only some very specific types of context-sensitive constraints, i.e., PMCFG constraints

and next token constraints. I would like to support more general types of constraints, and my algorithm is designed to accomodate that. The main research objective here will be to figure out the class of constraints that is actually needed.

- stateful parse actions: Custom parse actions currently have access only to minimal state information. I plan to make more state information available to parse actions to provide as much flexibility as will be needed for the target applications.
- a runtime parser for rules: Reading rules into the parser from a user-writable format at runtime, rather than from precompiled formats such as machine code or PGF grammars, is currently possible through the Concise GUI (Schodl et al. [87], Domes [22]). I am considering implementing a mechanism for specifying rules at runtime within DynGenPar. However, this has low priority for the FMathL project because in that application we use the mechanism provided by Concise.
- scalability to larger PMCFGs: Currently, I have several optimizations which improve scalability, but they only apply in the context-free case. In order to be able to process huge PMCFGs such as the resource grammars of the Grammatical Framework, I need to find ways to improve scalability also in the presence of constraints.
- error correction: At this time, DynGenPar only has basic error detection and reporting: A parse error happens when a shifted token is invalid for all pending parse stacks. I would like to design intelligent ways to actually correct the errors, or suggest corrections to the user. This is a long-term research goal.

My hope is that the above features will make it easy to parse enough mathematical text to build a large database of mathematical knowledge, as well as adapting to a huge variety of applications in mathematics and beyond (Kofler & Neumaier [56]).

There are several applications of DynGenPar and Concise that can be envisioned in the future. In particular, having a semantic representation of the mathematical documents allows building a semantics-aware search engine for Mathematics. Some existing algorithms, e.g. **MathWebSearch** (Kohlhase & Sucan [61]), come very close to semantics-awareness for formulas (that have to be input in a formal language such as **MathML** (W3C [97])). Using tricks such as term substitution, they can find formulas that look different at first sight, but have the same meaning. However, none of the existing methods even attempts to do the same for text. Representing the text semantically with the help of DynGenPar will allow Concise to do exactly that. The representation can also be exported to several formats, such as **OMDoc** (**Open Mathematical Documents**) (Kohlhase [58]), an XML representation for whole documents using the more widely known **OpenMath** (Buswell et al. [11]) XML format for formulas.

Appendix A

The TypeSheets.cnt Type Sheet

This appendix reproduces the source code of the grammar for type sheets described in the thesis in Section 4.1. It is itself given in the form of a Concise type sheet (see Section 4.1).

For a quick reading of the type sheet, you can at first skip the lines starting with a # sign. The type sheet is complete and consistent even without those lines.

The lines starting with a # sign are *usages* (see Sections 3.2 and 4.1). They provide additional information complementing the type system, such as grammatical information used by DynGenPar. See Section 3.2 for details about the different types of usages.

Additional examples of type sheets annotated with usages for DynGenPar can be found in Kofler & Neumaier [57] and Kofler & Baharev [51].

```
TypeSheets(English,TypeSheets)::ExternalTypes.String, ExternalTypes.Integer

! Type sheets
! -----------
!
! Arnold Neumaier and Kevin Kofler
!
! May 22, 2017
!
! This is a grammar for annotated type sheets, including a specification
! of both the lexical and the context free part.
!
! Initially, the (#t #tr) part of this grammar had to be encoded by hand
! in the cnttoxml converter. Now, the parser can be fed with info created
! from typesheets already read, so the grammar is now reproducible
! automatically from the present type sheet. However, in practice, we
! still rely on cnttoxml for practical bootstrapping reasons.
!
! View info is specified by productions.
! Literal productions are global to the type system and follow a line
! consisting only of a colon : (and optional comments).
! Categorical productions are local to a category and follow the type
! definition of that category.
!
```

```
!
! Within productions (only), the exclamation mark !, the hash #,
! the ampersand &, the newline characters and the blanks are escaped,
! since they have syntactic meaning in the grammar for the type sheet.
! &! encodes the exclamation mark, and &# encodes the hash #.
! &n encodes a newline. &b encodes a blank, &t encodes a tab, and &c encodes a
! backspace.
! &0, ..., &9 encode strings consisting of 0, ..., 9 ampersands, each
! of which is fed to the parser as an individual token.
! Both &0 and && followed by a newline denote the empty string.
!
! regular expression syntax:
! &[...&] is optional.
! &(...&) appears once.
! &{...&} appears n times, n>=0.
! &<...&> appears n times, n>=1.
! &| separates alternatives in a pattern.
! Note that the ampersands before the brackets or | may not be missing,
! since otherwise the symbol is treated as a character!
!
! syntax for lexical matching conditions:
! &^ accepts preceding single-character pattern match iff the following
!    pattern doesn't match the same character (set difference)
!    This consumes both patterns.
! &E accepts preceding pattern match iff the following single-character
!    pattern matches the next character ("expect" constraint), otherwise
!    the pattern fails to match entirely (i.e. this fails the entire
!    rule unless there's another &| alternative which matches)
!    This consumes the preceding pattern only.
! &T accepts preceding pattern match iff the following single-character
!    pattern does not match the next character ("taboo" constraint),
!    otherwise the pattern fails to match entirely (i.e. this fails the
!    entire rule unless there's another &| alternative which matches)
!    This consumes the preceding pattern only.
! &+ maximally extends preceding pattern of the form &{#char&} or
!    &<#char&>, where #char is a single-character pattern.
! Single-character patterns may contain any EBNF (regular expression
! syntax and variable references), but no further &^, &E, &T or &+.
!
! special syntax:
! &=&(pattern&) encodes the external ID (<0) of the external value 'pattern'
!
! syntax for variables:
! #fieldName  - jump to the entry referenced by the field of name 'fieldName'
!               and process the usage information for the new object.
!               The special word '&this' is not allowed as a fieldName.
! #>fieldName - the name of entry referenced by the field of name
!               'fieldName', or the value in case the entry is an
!               external object. The fieldName '&this' is used for
!               referencing the source handle
! #=fieldName - the id of entry referenced by the field of name
!               'fieldName' (<0 for external objects). The fieldName
!               '&this' is used for referencing the source handle
! ##name      - reference of a literal variable of name 'name'
!
```

```
! syntax for functions:
! #!name&( a list of patterns separated by the &, separator &)
! calls a function of name 'name' with arguments separated by &,
! The currently available functions are listed in ExternalFunctions.cnp.
!
! We use a special convention for specifying irregular concrete syntax
! for linked lists. This is indicated by a hash # in place of a
! production rule, expressing that the composition info for a linked
! list is at the calling field.
!
! Then the productions are defined in the category declaration whose
! entries are the head of the linked list. The corresponding productions
! are translated to context-free productions (as always) by replacing
! different categorical variables by different metavariables, but
! the object identified as corresponding to a categorical variable is
! stored not (as otherwise) in the entry of the position determined by
! the field but as entry in the linked list starting at that position.
!
! The objects are entered in the order from left to right; a new link is
! begun whenever the current link has the recognized position already
! filled.
!
! Example: Consider the pair of declarations
!
! CapList:
! allOf>entries=CapLink
! #t>> #entries=&(#cap&[ and #cap&]&|#cap, &<#cap, &>and #cap&)
!
! CapLink:
! allOf>cap=Capital
! optional> next=CapLink
! #t> #
!
! The # in the last line indicates that CapLink is to be treated via the
! generic link structure, using a rule from CapList. This rule
! corresponds to the extended BNF rule
!     E = CAP [ " and " CAP ] | CAP ", " CAP { CAP ", " } "and " CAP
!     CAP = ? upper case letter ?
! where E, CAP are the context-free metavariables corresponding to
! #entries and #cap.
! The inputs
!   (i) ''A''
!  (ii) ''A and B''
! (iii) ''A, B, C, and D''
! result in parse trees where E has children corresponding to
!   (i) A
!  (ii) A, B
! (iii) A, B, C, D
! The resulting linked lists are
!   (i) entries=$1, $1.cap=A
!  (ii) entries=$1, $1.cap=A, $1.next=$2, $2.cap=B
! (iii) entries=$1, $1.cap=A, $1.next=$2, $2.cap=B, $2.next=$3,
!       $3.cap=C, $3.next=$4,  $4.cap=D
!
!
```

```
! Example: Consider the pair of declarations
!
! CLList:
! allOf>entries=CLLink
! #t>> #entries=&{#cap&[ and #low,&]&}
!
! CLLink:
! allOf>cap=Capital
! optional> low=LowerCase
! optional> next=CLLink
! #t> #
!
! Now we get the extended BNF rule
!     S = { CAP [ " and " LOW "," ] }
!     CAP = ? upper case letter ?
!     LOW = ? lower case letter ?
! where S, CAP, LOW are the context-free metavariables corresponding to
! #entries, #cap, and #low.
! The inputs
!   (i) ''A and x''
!  (ii) ''ABC and x''
! (iii) ''A and x, B and y''
! result in parse trees where S has children corresponding to
!   (i) A, x
!  (ii) A, B, C, x
! (iii) A, x, B, y
! The resulting linked lists are
!   (i) entries=$1, $1.cap=A, $1.low=x
!  (ii) entries=$1, $1.cap=A, $1.next=$2, $2.cap=B, $2.next=$3,
!       $3.cap=C, $3.low=x
! (iii) entries=$1, $1.cap=A, $1.low=x, $1.next=$2, $2.cap=B, $2.low=y


! Target specification
!
! A target is a name for view info, and consists of a hash # followed by
! an identifier characterizing the kind of view, such as #t (used in
! this type sheet) for text view information.
! View info of kind #t, say, is defined in productions following lines
! starting with #t>> or #t>, depending on whether the production has
! or hasn't an explicit left hand side.
! This view info is placed in
!         secretary.library.views.#TS.#target
! where #TS is the name of the current type sheet (TypeSheets), and
! #target is to be substituted (before evaluation of the address from
! left to right) by the field sequence defined in the following
! specification. The same field sequence may have several targets,
! and a target may contribute to several field sequences.
! secretary.library.views.#TS.#target.label contains the label used to
! denote the kind of view (thus for the text view the letter t).
!
! New view info can be created by copying the type sheet and adding
! new targets; the old target info need not be repeated.
!
:TARGET> text.read=#t #tr           ! text read view in #t> and #tr>
```

```
         text.write=#t #tw          ! text write view in #t> and #tw>
         completion.filters.EN=#cf  ! completion filters in #cf>
:START=TypeSheet


! ##char denotes an arbitrary character. Among the characters are
! - the newline character, escaped as &n,
! - horizontal characters, collectively referred to as ##hchar
! - letters, collectively referred to as ##letter, and
! - digits, collectively refered to as ##digit.
!
! ##hchar can theoretically be defined as ##hchar=##char&^&n, but
! this leads to an unsupported nested context-sensitive constraint.
! These and some more basic literal variables are defined now:
:
#tr>> ##hchar=#[0-9, 11-12, 14-255]  ! horizontal character (UTF-8)
#tw>> ##hchar=#[0-9, 11-12, 14-65535]  ! horizontal character (UTF-16)
#cf>> ##hchar=Any character except newline
#t>> ##char=##hchar&|&n  ! general character
#t>> ##letter=#[65-90, 97-122]  ! Latin letter (upper- or lowercase)
#cf>> ##letter=Any letter
#t>> ##digit=#[48-57]  ! Western "Arabic" digit
#cf>> ##digit=Any digit
#tr>> ##blanks=&{&b&}&+  ! zero or more blanks
#tw>> ##blanks=&0  ! nothing
#tr>> ##eol=##blanks&<&n&>&+  ! trailing blanks followed by newlines
#tw>> ##eol=&n  ! single newline
#tr>> ##hascii=#[0-9, 11-12, 14-127]  ! horizontal character (ASCII only)
#cf>> ##hascii=Any ASCII character except newline
#tr>> ##utf8s2=#[192-223]  ! start byte of 2-byte UTF-8 sequence
#cf>> ##utf8s2=Any non-ASCII Unicode character
#tr>> ##utf8s3=#[224-239]  ! start byte of 3-byte UTF-8 sequence
#cf>> ##utf8s3=Any non-ASCII Unicode character
#tr>> ##utf8s4=#[240-247]  ! start byte of 4-byte UTF-8 sequence
#cf>> ##utf8s4=Any non-ASCII Unicode character
#tr>> ##utf8cont=#[128-191]  ! UTF-8 continuation byte
#t>> ##line=&{##hchar&}&+  ! string not containing a newline
#t>> ##id=##letter&{##letter&|##digit&}&+  ! alphanumeric string starting with
                                           ! a letter
#t>> ##digits=&<##digit&>&+  ! string of all digits
#t>> ##foreignword=&<##hchar&^:&>  ! colon-terminated string


! escape rules
:
#t>> ##outchar=##char&^"
#tr>> ##litchar=##hascii&^&(&!&|&#&|&1&|&b&)&|&&
                ##utf8s2##utf8cont&|&&
                ##utf8s3##utf8cont##utf8cont&|&&
                ##utf8s4##utf8cont##utf8cont##utf8cont
#tw>> ##litchar=##hchar&^&(&!&|&#&|&1&|&b&)
#t>> ##litchar=&1&!&"!"
#t>> ##litchar=&1&#&"#"
#t>> ##litchar=&1n&"
"
```

```
#t>> ##litchar=&1b&" "
#t>> ##litchar=&1t&"  "
#t>> ##litchar=&1c&"backspace"
#t>> ##litchar=&10&""
#t>> ##litchar=&11&"&"
#t>> ##litchar=&12&"&&"
#t>> ##litchar=&13&"&&&"
#t>> ##litchar=&14&"&&&&"
#t>> ##litchar=&15&"&&&&&"
#t>> ##litchar=&16&"&&&&&&"
#t>> ##litchar=&17&"&&&&&&&"
#t>> ##litchar=&18&"&&&&&&&&"
#t>> ##litchar=&19&"&&&&&&&&&"




! TypeSheet is the type of a handle containing a record encoding the
! result of parsing a type sheet
!
TypeSheet:
allOf> Header=Header
       entries=EntryLink
optional> Targets=Targets
optional> StartCategory=StartCategory
optional> postComments=CommentLink
#t> #Header&[#Targets&]&[#StartCategory&]#entries&[##eol#postComments&]&[##eol&]


! The header declares which type system is defined, which imports are
! part of the declared type system, and contains an obligatory
! introductory comment explaining the type sheet
!
Header:
allOf> TypeSystem=Id
       language=Id
       authority=Id
       comments=CommentLink
optional> imports=ImportLink
optional> translations=TranslationLink
#t> #TypeSystem(#language,##blanks#authority)::&&
&[&(##blanks&|##eol##blanks&)#imports&]##eol#comments&[#translations&]


! An identifier starts with a letter, followed by a (maximal) word
!
Id:
union> String
#t> ##id


! A positive integer
!
PosInt:
union> Integer
```

```
#t> ##digits


! An identifier in a foreign language
!
ForeignId:
union> String
#t> ##foreignword


! Each link of a linked list of imports is an ImportLink
!
ImportLink:
allOf> TypeSystem=Id
optional> Category=Id
          newName=Id
optional> next=ImportLink
#t> #TypeSystem&[.#Category&[->#newName&]&]&[,&(##blanks&|##eol##blanks&)#next&]


! Each link of a linked list of translations is a TranslationLink
!
TranslationLink:
allOf> language=Id
       idTranslations=IdTranslationLink
optional> preComments=CommentLink
optional> next=TranslationLink
#t> ##eol&[#preComments&]:LANG:##blanks#language##blanks&n&&
    #idTranslations&[#next&]


! Each link of a linked list of identifier translations is an IdTranslationLink
!
IdTranslationLink:
allOf> ForeignId=ForeignId
       Id=Id
optional> next=IdTranslationLink
#t> ::#ForeignId:##blanks#Id##blanks&n&[#next&]


! Each link of a linked list of comments is a CommentLink
!
CommentLink:
allOf> Comment=Comment
optional> next=CommentLink
#t> #Comment&[#next&]


! Comments start with a nonescaped ! and optional blanks and ends with a
! nonescaped newline. Stored is only the string in between, without
! the leading blanks.
!
Comment:
allOf> text=Line
#t> ##blanks&!##blanks#text&n
```

```
! A line is a string without newline characters
!
Line:
union>String
#t> ##line


! Targets
!
Targets:
allOf> targets=TargetLink
optional> preComments=CommentLink
#t> ##eol&[#preComments&]:TARGET>##blanks#targets


! Each link of a linked list of targets is a TargetLink
!
TargetLink:
allOf> Target=Target
optional> next=TargetLink
#t> #Target&[##blanks#next&]


! Target specifications
!
Target:
allOf> sequence=FieldLink
       vars=VarLink
optional> comment=Comment
#t> #sequence=#vars&(#comment&|##blanks&n&)


! Each link of a linked list of fields is a FieldLink
!
FieldLink:
allOf> field=Id
optional> next=FieldLink
#t> #field&[.#next&]


! Each link of a linked list of variables is a VarLink
!
VarLink:
allOf> name=Id
optional> next=VarLink
#t> &##name&[&b#next&]


! Start category specification
!
StartCategory:
allOf> start=Id
optional> comment=Comment
```

```
#t> :START=#start&(#comment&|##blanks&n&)


! Each link of a linked list of entries in the type system is an
! EntryLink
!
EntryLink:
oneOf> LitDef=LitDef
       CatDef=CatDef
optional> next=EntryLink
#t> &(#LitDef&|#CatDef&)&[#next&]


! Literal variable definitions
!
LitDef:
allOf> productions=LitLink
optional> preComments=CommentLink
optional> postComments=CommentLink
#t> ##eol&[#preComments&]:&(#postComments&|##blanks&n&)#productions


! Each link of a linked list of literal productions is a LitLink
!
LitLink:
allOf> production=LitProduction
optional> next=LitLink
#t> #production&[#next&]


! Literal productions
!
LitProduction:
allOf> target=Id
       field=Id
oneOf> Substitution=Substitution
       production=AlternativeLink
       cClass=CRangeLink
optional> comments=CommentLink
#t> ##blanks&##target>>&b&#&##field=#Substitution&(#comments&|##blanks&n&)
#t> ##blanks&##target>>&b&#&##field=#production&(#comments&|&n&)
#t> ##blanks&##target>>&b&#&##field=&#[#cClass]&(#comments&|##blanks&n&)


! A substitution in a literal production
!
Substitution:
allOf> input=CharLink
optional> output=OutCharLink
#t> #input&1"&[#output&]"


! A character range. If end is not specified, it is the same as start.
!
CRange:
```

```
allOf> start=PosInt
optional> end=PosInt
#t> #start&[-#end&]


! A list of character ranges, encodes a character class in a literal production
!
CRangeLink:
allOf> CRange=CRange
optional> next=CRangeLink
#t> #CRange&[,##blanks#next&]


! Literal characters
!
Char:
union> String
#t> ##litchar


! Each link of a linked list of literal characters is a CharLink
!
CharLink:
allOf> Char=Char
optional> next=CharLink
#t> #Char&[#next&]


! Output characters
!
OutChar:
union> String
#t> ##outchar


! Each link of a linked list of output characters is an OutCharLink
!
OutCharLink:
allOf> OutChar=OutChar
optional> next=OutCharLink
#t> #OutChar&[#next&]


! An alternative in a pattern (same as ElementLink)
!
Alternative:
union> #e:ElementLink
#t> #e


! Each link of a linked list of alternatives is an AlternativeLink
! &| separates alternatives in a pattern.
!
AlternativeLink:
allOf> Alternative=Alternative
```

```
optional> next=AlternativeLink
#t> #Alternative&[&1|#next&]


! Each link of a linked list of elements is an ElementLink
! This represents the sequence of elements in a single alternative.
!
ElementLink:
allOf> element=Element
optional> next=ElementLink
#t> #element&[#next&]


! Elements are the basic unit of a categorical production
!
! regular expression syntax
! &[...&] is optional.
! &(...&) appears once.
! &{...&} appears n times, n>=0.
! &<...&> appears n times, n>=1 .
!
Element:
oneOf> #e:CatVar=CatVar
       #e:LitVar=LitVar
       #e:literal=Char
       #e:LitId=LitId
       #e:Function=Function
       #d:Blanks=Blanks
       #d:LineBreak=LineBreak
       optional=AlternativeLink
       once=AlternativeLink
       anyTimes=AlternativeLink
       multiple=AlternativeLink
optional> match=MatchCase
#t> #e&[#match&]
#t> #d
#t> &1[#optional&1]&[#match&]
#t> &1(#once&1)&[#match&]
#t> &1{#anyTimes&1}&[#match&]
#t> &1<#multiple&1>&[#match&]


! Dummy element consisting only of blanks, acts as a silent separator
!
Blanks:
union> String
#t> &b##blanks&T&!


! Dummy element representing an escaped newline (&& + newline) used to
! break lines inside productions
!
LineBreak:
nothing>
#t> &2&n
```

```
! Match cases store a lexical matching condition
!
! syntax for lexical matching conditions:
! &^ accepts preceding single-character pattern match iff the following
!    pattern doesn't match the same character (set difference)
!    This consumes both patterns.
! &E accepts preceding pattern match iff the following single-character
!    pattern matches the next character ("expect" constraint), otherwise
!    the pattern fails to match entirely (i.e. this fails the entire
!    rule unless there's another &| alternative which matches)
!    This consumes the preceding pattern only.
! &T accepts preceding pattern match iff the following single-character
!    pattern does not match the next character ("taboo" constraint),
!    otherwise the pattern fails to match entirely (i.e. this fails the
!    entire rule unless there's another &| alternative which matches)
!    This consumes the preceding pattern only.
! &+ maximally extends preceding pattern of the form &{#char&} or
!    &<#char&>, where #char is a single-character pattern.
! Single-character patterns may contain any EBNF (regular expression
! syntax and variable references), but no further &^, &e, &t or &+.
!
MatchCase:
union> #e:Maximal, #e:Expect, #e:Taboo, #e:Except
#t> ##blanks#e


! Match required to be maximal
! &+
Maximal:
union> String
#t> &1+


! Match requiring specific next characters
! &E
Expect:
allOf> pattern=Element
#t> &1E#pattern


! Match excluding specific next characters
! &T
Taboo:
allOf> pattern=Element
#t> &1T#pattern



! Match excluding specific characters
! &^
Except:
allOf> pattern=Element
#t> &1^#pattern
```

```
! The different types of references to categorical variables.
! They all start with a single unescaped hash.
!
CatVar:
union> #e:CatVarRec, #e:CatVarName, #e:CatVarId
#t> &##e


! Regular references to categorical (meta)variables are denoted by an
! unescaped hash #  followed by an alphanumerical string starting with
! a letter.
! They must be followed by an explicit &0 when the next character is a
! letter or a digit.
! All categorical variables are defined by the fields of a category and
! are locally valid only within the category in which they are defined.
!
CatVarRec:
allOf> name=Id
#t> #name


! #>fieldName - the name of entry referenced by the field of name
!               'fieldName', or the value in case the entry is an
!               external object. The fieldName '&this' is used for
!               referencing the source handle.
! An empty name field corresponds to '&this', i.e. the source handle.
!
CatVarName:
optional> name=Id
#t> >&(#name&|&1this&)


! #=fieldName - the id of entry referenced by the field of name
!               'fieldName' (<0 for external objects). The fieldName
!               '&this' is used for referencing the source handle
! An empty name field corresponds to '&this', i.e. the source handle.
!
CatVarId:
optional> name=Id
#t> =&(#name&|&1this&)



! Literal (meta)variables are denoted by unescaped ## followed
! by an alphanumerical string starting with a letter.
! They must be followed by an explicit &0 when the next character is a
! letter or a digit.
! All other literal variables must be declared in productions that are
! globally valid within a typesystem.
!
LitVar:
allOf> name=Id
#t> &##&##name
```

```
! &=&(pattern&) encodes the node ID of the external value 'pattern'
!
LitId:
allOf> pattern=ElementLink
#t> &1=&1(#pattern&1)


! A function argument (same as ElementLink)
!
FunArg:
union> #e:ElementLink
#t> #e


! A list of function arguments
! This differs from AlternativeLink in that the separator is &, rather than &|.
!
FunArgLink:
allOf> FunArg=FunArg
optional> next=FunArgLink
#t> #FunArg&[&1,#next&]


! A function call (see ExternalFunctions.cnp for the available functions)
! #!name&( list of function arguments &)
! The list of function arguments can be empty.
!
Function:
allOf> name=Id
optional> args=FunArgLink
#t> &#&!#name##blanks&1(&[#args&]&1)


! Categorical variable definition
!
CatDef:
allOf> Category=Id
       specifications=SpecLink
optional> extends=Id
          productions=CatLink
          irregular=IrrLink
          preComments=CommentLink
          postComments=CommentLink
#t> ##eol&[#preComments&]#Category:&[&b#extends+&]&n&[#postComments&]&&
#specifications&[#productions&[#irregular&]&]


! Each link of a linked list of specifications is a SpecLink
!
SpecLink:
oneOf> Spec=Spec
optional> next=SpecLink
#t> #Spec&[#next&]
```

```
! Specifications of fields or field types
!
Spec:
union> #s:AllOfSpec, #s:OneOfSpec, #s:SomeOfSpec, #s:OptionalSpec, #s:FixedSpec,
       #s:OnlySpec, #s:SomeOfTypeSpec, #s:ItselfSpec, #s:ArraySpec,
       #s:IndexSpec, #s:TemplateSpec, #s:NothingElseSpec, #s:NothingSpec,
       #s:UnionSpec, #s:AtomicSpec, #s:CompleteSpec
#t> #s


! allOf qualifier (takes a list of equations)
!
AllOfSpec:
allOf> equations=EqLink
#t> allOf>##blanks#equations


! oneOf qualifier (takes a list of equations)
!
OneOfSpec:
allOf> equations=EqLink
#t> oneOf>##blanks#equations


! someOf qualifier (takes a list of equations)
!
SomeOfSpec:
allOf> equations=EqLink
#t> someOf>##blanks#equations


! optional qualifier (takes a list of equations)
!
OptionalSpec:
allOf> equations=EqLink
#t> optional>##blanks#equations


! fixed qualifier (takes a list of equations)
!
FixedSpec:
allOf> equations=EqLink
#t> fixed>##blanks#equations


! only qualifier (takes a list of equations)
!
OnlySpec:
allOf> equations=EqLink
#t> only>##blanks#equations


! someOfType qualifier (takes a list of equations)
```

```
!
SomeOfTypeSpec:
allOf> equations=EqLink
#t> someOfType>##blanks#equations


! itself qualifier (takes a list of names)
!
ItselfSpec:
allOf> names=NameLink
#t> itself>##blanks#names


! array qualifier (takes a list of equations)
!
ArraySpec:
allOf> equations=EqLink
#t> array>##blanks#equations


! index qualifier (takes a list of equations)
!
IndexSpec:
allOf> equations=EqLink
#t> index>##blanks#equations


! template qualifier (takes the name of the template)
!
TemplateSpec:
allOf> name=Id
#t> template>##blanks#name##blanks&n


! nothingElse qualifier (takes no arguments)
!
NothingElseSpec:
nothing>
#t> nothingElse>##blanks&n


! nothing qualifier (takes no arguments)
!
NothingSpec:
nothing>
#t> nothing>##blanks&n


! union qualifier (takes a list of names)
!
UnionSpec:
allOf> names=NameLink
#t> union>##blanks#names
```

```
! atomic qualifier (takes a list of names)
!
AtomicSpec:
allOf> names=NameLink
#t> atomic>##blanks#names


! complete qualifier (takes no arguments)
!
CompleteSpec:
nothing>
#t> complete>##blanks&n


! Each link of a linked list of equations is an EqLink
!
! The optional #name defines a local categorical variable to use in
! the concrete syntax instead of the #field which follows.
!
EqLink:
allOf> field=Id
       Category=Id
optional> name=Id
          comment=Comment
          next=EqLink
#t> &[&##name:&]#field=#Category&(#comment&|##blanks&n&)&[##blanks#next&]


! Each link of a linked list of names is a NameLink
!
! The optional #name defines a local categorical variable to use in
! the concrete syntax instead of the #field which follows.
!
NameLink:
allOf> field=Id
optional> name=Id
          next=NameLink
#t> &[&##name:&]#field&(,&(&b&|##eol##blanks&)#next&|&n&)


! Each link of a linked list of categorical productions is a CatLink
!
CatLink:
allOf> production=CatProduction
optional> next=CatLink
#t> #production&[#next&]


! Categorical production
! production is empty if it is declared as an IrrProduction elsewhere.
!
CatProduction:
allOf> target=Id
optional> production=AlternativeLink ! composition info is here
          comments=CommentLink
```

```
#t> ##blanks&##target>&b&(#production&(#comments&|&n&)&|&&
                          &#&(&b#comments&|##blanks&n&)&)


! Each link of a linked list of irregular productions is an IrrLink
!
IrrLink:
allOf> production=IrrProduction
optional> next=IrrLink
#t> #production&[#next&]


! Irregular productions define concrete syntax for irregular linked
! lists
!
IrrProduction:
allOf> target=Id
       field=Id
       production=AlternativeLink
optional> comments=CommentLink
#t> ##blanks&##target>>&b&##field=#production&(#comments&|&n&)
```

Appendix B

Internal Representation (BNF) of the TypeSheets Grammar

Converting the type sheet `TypeSheets.cnt` (Appendix A) documented in Section 4.1 through the process from Section 3.2 yields the internal representation reproduced in text form below. It is a grammar in the standard Backus-Naur form (BNF), with the following extensions:

- Curly braces indicate rule labels (see Section 2.2.3.3), which are actually references to the Java objects from Section 3.2.
- Square brackets indicate next token constraints (see Section 2.2.3.7).

```
Start category: TypeSheet
Token: SOH
Token: STX
Token: ETX
Token: EOT
Token: ENQ
Token: ACK
Token: BEL
Token: BS
Token: HT
Token: LF
Token: VT
Token: FF
Token: CR
Token: SO
Token: SI
Token: DLE
Token: DC1
Token: DC2
Token: DC3
Token: DC4
Token: NAK
Token: SYN
Token: ETB
Token: CAN
```

```
Token: EM
Token: SUB
Token: ESC
Token: FS
Token: GS
Token: RS
Token: US
Token: SP
Token: !
Token: "
Token: #
Token: $
Token: %
Token: &
Token: '
Token: (
Token: )
Token: *
Token: +
Token: ,
Token: -
Token: .
Token: /
Token: 0
Token: 1
Token: 2
Token: 3
Token: 4
Token: 5
Token: 6
Token: 7
Token: 8
Token: 9
Token: :
Token: ;
Token: <
Token: =
Token: >
Token: ?
Token: @
Token: A
Token: B
Token: C
Token: D
Token: E
Token: F
Token: G
Token: H
Token: I
Token: J
Token: K
Token: L
Token: M
Token: N
Token: O
```

```
Token: P
Token: Q
Token: R
Token: S
Token: T
Token: U
Token: V
Token: W
Token: X
Token: Y
Token: Z
Token: [
Token: \
Token: ]
Token: ^
Token: _
Token: `
Token: a
Token: b
Token: c
Token: d
Token: e
Token: f
Token: g
Token: h
Token: i
Token: j
Token: k
Token: l
Token: m
Token: n
Token: o
Token: p
Token: q
Token: r
Token: s
Token: t
Token: u
Token: v
Token: w
Token: x
Token: y
Token: z
Token: {
Token: |
Token: }
Token: ~
Token: DEL
Token: \x80
Token: \x81
Token: \x82
Token: \x83
Token: \x84
Token: \x85
Token: \x86
```

```
Token: \x87
Token: \x88
Token: \x89
Token: \x8a
Token: \x8b
Token: \x8c
Token: \x8d
Token: \x8e
Token: \x8f
Token: \x90
Token: \x91
Token: \x92
Token: \x93
Token: \x94
Token: \x95
Token: \x96
Token: \x97
Token: \x98
Token: \x99
Token: \x9a
Token: \x9b
Token: \x9c
Token: \x9d
Token: \x9e
Token: \x9f
Token: \xa0
Token: \xa1
Token: \xa2
Token: \xa3
Token: \xa4
Token: \xa5
Token: \xa6
Token: \xa7
Token: \xa8
Token: \xa9
Token: \xaa
Token: \xab
Token: \xac
Token: \xad
Token: \xae
Token: \xaf
Token: \xb0
Token: \xb1
Token: \xb2
Token: \xb3
Token: \xb4
Token: \xb5
Token: \xb6
Token: \xb7
Token: \xb8
Token: \xb9
Token: \xba
Token: \xbb
Token: \xbc
Token: \xbd
```

```
Token: \xbe
Token: \xbf
Token: \xc0
Token: \xc1
Token: \xc2
Token: \xc3
Token: \xc4
Token: \xc5
Token: \xc6
Token: \xc7
Token: \xc8
Token: \xc9
Token: \xca
Token: \xcb
Token: \xcc
Token: \xcd
Token: \xce
Token: \xcf
Token: \xd0
Token: \xd1
Token: \xd2
Token: \xd3
Token: \xd4
Token: \xd5
Token: \xd6
Token: \xd7
Token: \xd8
Token: \xd9
Token: \xda
Token: \xdb
Token: \xdc
Token: \xdd
Token: \xde
Token: \xdf
Token: \xe0
Token: \xe1
Token: \xe2
Token: \xe3
Token: \xe4
Token: \xe5
Token: \xe6
Token: \xe7
Token: \xe8
Token: \xe9
Token: \xea
Token: \xeb
Token: \xec
Token: \xed
Token: \xee
Token: \xef
Token: \xf0
Token: \xf1
Token: \xf2
Token: \xf3
Token: \xf4
```

```
Token: \xf5
Token: \xf6
Token: \xf7
Token: \xf8
Token: \xf9
Token: \xfa
Token: \xfb
Token: \xfc
Token: \xfd
Token: \xfe
Token: \xff
Token: NUL
##blanks -> ##blanks#1 {TokenCollector}
##char -> ##hchar {TokenCollector}
  | LF {TokenCollector}
##digit -> 0 {TokenCollector}
  | 1 {TokenCollector}
  | 2 {TokenCollector}
  | 3 {TokenCollector}
  | 4 {TokenCollector}
  | 5 {TokenCollector}
  | 6 {TokenCollector}
  | 7 {TokenCollector}
  | 8 {TokenCollector}
  | 9 {TokenCollector}
##digits -> ##digits#1 {TokenCollector}
##eol -> ##blanks ##eol#1 {TokenCollector}
##foreignword -> ##foreignword#1 {TokenCollector}
##hascii -> NUL {TokenCollector}
  | SOH {TokenCollector}
  | STX {TokenCollector}
  | ETX {TokenCollector}
  | EOT {TokenCollector}
  | ENQ {TokenCollector}
  | ACK {TokenCollector}
  | BEL {TokenCollector}
  | BS {TokenCollector}
  | HT {TokenCollector}
  | VT {TokenCollector}
  | FF {TokenCollector}
  | SO {TokenCollector}
  | SI {TokenCollector}
  | DLE {TokenCollector}
  | DC1 {TokenCollector}
  | DC2 {TokenCollector}
  | DC3 {TokenCollector}
  | DC4 {TokenCollector}
  | NAK {TokenCollector}
  | SYN {TokenCollector}
  | ETB {TokenCollector}
  | CAN {TokenCollector}
  | EM {TokenCollector}
  | SUB {TokenCollector}
  | ESC {TokenCollector}
  | FS {TokenCollector}
```

```
| GS {TokenCollector}
| RS {TokenCollector}
| US {TokenCollector}
| SP {TokenCollector}
| ! {TokenCollector}
| " {TokenCollector}
| # {TokenCollector}
| $ {TokenCollector}
| % {TokenCollector}
| & {TokenCollector}
| ' {TokenCollector}
| ( {TokenCollector}
| ) {TokenCollector}
| * {TokenCollector}
| + {TokenCollector}
| , {TokenCollector}
| - {TokenCollector}
| . {TokenCollector}
| / {TokenCollector}
| 0 {TokenCollector}
| 1 {TokenCollector}
| 2 {TokenCollector}
| 3 {TokenCollector}
| 4 {TokenCollector}
| 5 {TokenCollector}
| 6 {TokenCollector}
| 7 {TokenCollector}
| 8 {TokenCollector}
| 9 {TokenCollector}
| : {TokenCollector}
| ; {TokenCollector}
| < {TokenCollector}
| = {TokenCollector}
| > {TokenCollector}
| ? {TokenCollector}
| @ {TokenCollector}
| A {TokenCollector}
| B {TokenCollector}
| C {TokenCollector}
| D {TokenCollector}
| E {TokenCollector}
| F {TokenCollector}
| G {TokenCollector}
| H {TokenCollector}
| I {TokenCollector}
| J {TokenCollector}
| K {TokenCollector}
| L {TokenCollector}
| M {TokenCollector}
| N {TokenCollector}
| O {TokenCollector}
| P {TokenCollector}
| Q {TokenCollector}
| R {TokenCollector}
| S {TokenCollector}
```

```
  | T {TokenCollector}
  | U {TokenCollector}
  | V {TokenCollector}
  | W {TokenCollector}
  | X {TokenCollector}
  | Y {TokenCollector}
  | Z {TokenCollector}
  | [ {TokenCollector}
  | \ {TokenCollector}
  | ] {TokenCollector}
  | ^ {TokenCollector}
  | _ {TokenCollector}
  | ` {TokenCollector}
  | a {TokenCollector}
  | b {TokenCollector}
  | c {TokenCollector}
  | d {TokenCollector}
  | e {TokenCollector}
  | f {TokenCollector}
  | g {TokenCollector}
  | h {TokenCollector}
  | i {TokenCollector}
  | j {TokenCollector}
  | k {TokenCollector}
  | l {TokenCollector}
  | m {TokenCollector}
  | n {TokenCollector}
  | o {TokenCollector}
  | p {TokenCollector}
  | q {TokenCollector}
  | r {TokenCollector}
  | s {TokenCollector}
  | t {TokenCollector}
  | u {TokenCollector}
  | v {TokenCollector}
  | w {TokenCollector}
  | x {TokenCollector}
  | y {TokenCollector}
  | z {TokenCollector}
  | { {TokenCollector}
  | | {TokenCollector}
  | } {TokenCollector}
  | ~ {TokenCollector}
  | DEL {TokenCollector}
##hchar -> NUL {TokenCollector}
  | SOH {TokenCollector}
  | STX {TokenCollector}
  | ETX {TokenCollector}
  | EOT {TokenCollector}
  | ENQ {TokenCollector}
  | ACK {TokenCollector}
  | BEL {TokenCollector}
  | BS {TokenCollector}
  | HT {TokenCollector}
  | VT {TokenCollector}
```

```
| FF {TokenCollector}
| SO {TokenCollector}
| SI {TokenCollector}
| DLE {TokenCollector}
| DC1 {TokenCollector}
| DC2 {TokenCollector}
| DC3 {TokenCollector}
| DC4 {TokenCollector}
| NAK {TokenCollector}
| SYN {TokenCollector}
| ETB {TokenCollector}
| CAN {TokenCollector}
| EM {TokenCollector}
| SUB {TokenCollector}
| ESC {TokenCollector}
| FS {TokenCollector}
| GS {TokenCollector}
| RS {TokenCollector}
| US {TokenCollector}
| SP {TokenCollector}
| ! {TokenCollector}
| " {TokenCollector}
| # {TokenCollector}
| $ {TokenCollector}
| % {TokenCollector}
| & {TokenCollector}
| ' {TokenCollector}
| ( {TokenCollector}
| ) {TokenCollector}
| * {TokenCollector}
| + {TokenCollector}
| , {TokenCollector}
| - {TokenCollector}
| . {TokenCollector}
| / {TokenCollector}
| 0 {TokenCollector}
| 1 {TokenCollector}
| 2 {TokenCollector}
| 3 {TokenCollector}
| 4 {TokenCollector}
| 5 {TokenCollector}
| 6 {TokenCollector}
| 7 {TokenCollector}
| 8 {TokenCollector}
| 9 {TokenCollector}
| : {TokenCollector}
| ; {TokenCollector}
| < {TokenCollector}
| = {TokenCollector}
| > {TokenCollector}
| ? {TokenCollector}
| @ {TokenCollector}
| A {TokenCollector}
| B {TokenCollector}
| C {TokenCollector}
```

```
| D {TokenCollector}
| E {TokenCollector}
| F {TokenCollector}
| G {TokenCollector}
| H {TokenCollector}
| I {TokenCollector}
| J {TokenCollector}
| K {TokenCollector}
| L {TokenCollector}
| M {TokenCollector}
| N {TokenCollector}
| O {TokenCollector}
| P {TokenCollector}
| Q {TokenCollector}
| R {TokenCollector}
| S {TokenCollector}
| T {TokenCollector}
| U {TokenCollector}
| V {TokenCollector}
| W {TokenCollector}
| X {TokenCollector}
| Y {TokenCollector}
| Z {TokenCollector}
| [ {TokenCollector}
| \ {TokenCollector}
| ] {TokenCollector}
| ^ {TokenCollector}
| _ {TokenCollector}
| ` {TokenCollector}
| a {TokenCollector}
| b {TokenCollector}
| c {TokenCollector}
| d {TokenCollector}
| e {TokenCollector}
| f {TokenCollector}
| g {TokenCollector}
| h {TokenCollector}
| i {TokenCollector}
| j {TokenCollector}
| k {TokenCollector}
| l {TokenCollector}
| m {TokenCollector}
| n {TokenCollector}
| o {TokenCollector}
| p {TokenCollector}
| q {TokenCollector}
| r {TokenCollector}
| s {TokenCollector}
| t {TokenCollector}
| u {TokenCollector}
| v {TokenCollector}
| w {TokenCollector}
| x {TokenCollector}
| y {TokenCollector}
| z {TokenCollector}
```

```
| { {TokenCollector}
| | {TokenCollector}
| } {TokenCollector}
| ~ {TokenCollector}
| DEL {TokenCollector}
| \x80 {TokenCollector}
| \x81 {TokenCollector}
| \x82 {TokenCollector}
| \x83 {TokenCollector}
| \x84 {TokenCollector}
| \x85 {TokenCollector}
| \x86 {TokenCollector}
| \x87 {TokenCollector}
| \x88 {TokenCollector}
| \x89 {TokenCollector}
| \x8a {TokenCollector}
| \x8b {TokenCollector}
| \x8c {TokenCollector}
| \x8d {TokenCollector}
| \x8e {TokenCollector}
| \x8f {TokenCollector}
| \x90 {TokenCollector}
| \x91 {TokenCollector}
| \x92 {TokenCollector}
| \x93 {TokenCollector}
| \x94 {TokenCollector}
| \x95 {TokenCollector}
| \x96 {TokenCollector}
| \x97 {TokenCollector}
| \x98 {TokenCollector}
| \x99 {TokenCollector}
| \x9a {TokenCollector}
| \x9b {TokenCollector}
| \x9c {TokenCollector}
| \x9d {TokenCollector}
| \x9e {TokenCollector}
| \x9f {TokenCollector}
| \xa0 {TokenCollector}
| \xa1 {TokenCollector}
| \xa2 {TokenCollector}
| \xa3 {TokenCollector}
| \xa4 {TokenCollector}
| \xa5 {TokenCollector}
| \xa6 {TokenCollector}
| \xa7 {TokenCollector}
| \xa8 {TokenCollector}
| \xa9 {TokenCollector}
| \xaa {TokenCollector}
| \xab {TokenCollector}
| \xac {TokenCollector}
| \xad {TokenCollector}
| \xae {TokenCollector}
| \xaf {TokenCollector}
| \xb0 {TokenCollector}
| \xb1 {TokenCollector}
```

```
| \xb2 {TokenCollector}
| \xb3 {TokenCollector}
| \xb4 {TokenCollector}
| \xb5 {TokenCollector}
| \xb6 {TokenCollector}
| \xb7 {TokenCollector}
| \xb8 {TokenCollector}
| \xb9 {TokenCollector}
| \xba {TokenCollector}
| \xbb {TokenCollector}
| \xbc {TokenCollector}
| \xbd {TokenCollector}
| \xbe {TokenCollector}
| \xbf {TokenCollector}
| \xc0 {TokenCollector}
| \xc1 {TokenCollector}
| \xc2 {TokenCollector}
| \xc3 {TokenCollector}
| \xc4 {TokenCollector}
| \xc5 {TokenCollector}
| \xc6 {TokenCollector}
| \xc7 {TokenCollector}
| \xc8 {TokenCollector}
| \xc9 {TokenCollector}
| \xca {TokenCollector}
| \xcb {TokenCollector}
| \xcc {TokenCollector}
| \xcd {TokenCollector}
| \xce {TokenCollector}
| \xcf {TokenCollector}
| \xd0 {TokenCollector}
| \xd1 {TokenCollector}
| \xd2 {TokenCollector}
| \xd3 {TokenCollector}
| \xd4 {TokenCollector}
| \xd5 {TokenCollector}
| \xd6 {TokenCollector}
| \xd7 {TokenCollector}
| \xd8 {TokenCollector}
| \xd9 {TokenCollector}
| \xda {TokenCollector}
| \xdb {TokenCollector}
| \xdc {TokenCollector}
| \xdd {TokenCollector}
| \xde {TokenCollector}
| \xdf {TokenCollector}
| \xe0 {TokenCollector}
| \xe1 {TokenCollector}
| \xe2 {TokenCollector}
| \xe3 {TokenCollector}
| \xe4 {TokenCollector}
| \xe5 {TokenCollector}
| \xe6 {TokenCollector}
| \xe7 {TokenCollector}
| \xe8 {TokenCollector}
```

```
  | \xe9 {TokenCollector}
  | \xea {TokenCollector}
  | \xeb {TokenCollector}
  | \xec {TokenCollector}
  | \xed {TokenCollector}
  | \xee {TokenCollector}
  | \xef {TokenCollector}
  | \xf0 {TokenCollector}
  | \xf1 {TokenCollector}
  | \xf2 {TokenCollector}
  | \xf3 {TokenCollector}
  | \xf4 {TokenCollector}
  | \xf5 {TokenCollector}
  | \xf6 {TokenCollector}
  | \xf7 {TokenCollector}
  | \xf8 {TokenCollector}
  | \xf9 {TokenCollector}
  | \xfa {TokenCollector}
  | \xfb {TokenCollector}
  | \xfc {TokenCollector}
  | \xfd {TokenCollector}
  | \xfe {TokenCollector}
  | \xff {TokenCollector}
##id -> ##letter ##id#1 {TokenCollector}
##letter -> A {TokenCollector}
  | B {TokenCollector}
  | C {TokenCollector}
  | D {TokenCollector}
  | E {TokenCollector}
  | F {TokenCollector}
  | G {TokenCollector}
  | H {TokenCollector}
  | I {TokenCollector}
  | J {TokenCollector}
  | K {TokenCollector}
  | L {TokenCollector}
  | M {TokenCollector}
  | N {TokenCollector}
  | O {TokenCollector}
  | P {TokenCollector}
  | Q {TokenCollector}
  | R {TokenCollector}
  | S {TokenCollector}
  | T {TokenCollector}
  | U {TokenCollector}
  | V {TokenCollector}
  | W {TokenCollector}
  | X {TokenCollector}
  | Y {TokenCollector}
  | Z {TokenCollector}
  | a {TokenCollector}
  | b {TokenCollector}
  | c {TokenCollector}
  | d {TokenCollector}
  | e {TokenCollector}
```

```
  | f {TokenCollector}
  | g {TokenCollector}
  | h {TokenCollector}
  | i {TokenCollector}
  | j {TokenCollector}
  | k {TokenCollector}
  | l {TokenCollector}
  | m {TokenCollector}
  | n {TokenCollector}
  | o {TokenCollector}
  | p {TokenCollector}
  | q {TokenCollector}
  | r {TokenCollector}
  | s {TokenCollector}
  | t {TokenCollector}
  | u {TokenCollector}
  | v {TokenCollector}
  | w {TokenCollector}
  | x {TokenCollector}
  | y {TokenCollector}
  | z {TokenCollector}
##line -> ##line#1 {TokenCollector}
##litchar -> & ! {TokenEmitter: !}
  | & # {TokenEmitter: #}
  | & n {TokenEmitter: LF}
  | & b {TokenEmitter: SP}
  | & t {TokenEmitter: HT}
  | & c {TokenEmitter: BS}
  | & 0 {TokenEmitter:}
  | & 1 {TokenEmitter: &}
  | & 2 {TokenEmitter: & &}
  | & 3 {TokenEmitter: & & &}
  | & 4 {TokenEmitter: & & & &}
  | & 5 {TokenEmitter: & & & & &}
  | & 6 {TokenEmitter: & & & & & &}
  | & 7 {TokenEmitter: & & & & & & &}
  | & 8 {TokenEmitter: & & & & & & & &}
  | & 9 {TokenEmitter: & & & & & & & & &}
  | ##litchar#3 ##hascii {TokenCollector}
  | ##utf8s2 ##utf8cont {TokenCollector}
  | ##utf8s3 ##utf8cont ##utf8cont {TokenCollector}
  | ##utf8s4 ##utf8cont ##utf8cont ##utf8cont {TokenCollector}
##outchar -> ##outchar#2 ##char {TokenCollector}
##utf8cont -> \x80 {TokenCollector}
  | \x81 {TokenCollector}
  | \x82 {TokenCollector}
  | \x83 {TokenCollector}
  | \x84 {TokenCollector}
  | \x85 {TokenCollector}
  | \x86 {TokenCollector}
  | \x87 {TokenCollector}
  | \x88 {TokenCollector}
  | \x89 {TokenCollector}
  | \x8a {TokenCollector}
  | \x8b {TokenCollector}
```

```
  | \x8c {TokenCollector}
  | \x8d {TokenCollector}
  | \x8e {TokenCollector}
  | \x8f {TokenCollector}
  | \x90 {TokenCollector}
  | \x91 {TokenCollector}
  | \x92 {TokenCollector}
  | \x93 {TokenCollector}
  | \x94 {TokenCollector}
  | \x95 {TokenCollector}
  | \x96 {TokenCollector}
  | \x97 {TokenCollector}
  | \x98 {TokenCollector}
  | \x99 {TokenCollector}
  | \x9a {TokenCollector}
  | \x9b {TokenCollector}
  | \x9c {TokenCollector}
  | \x9d {TokenCollector}
  | \x9e {TokenCollector}
  | \x9f {TokenCollector}
  | \xa0 {TokenCollector}
  | \xa1 {TokenCollector}
  | \xa2 {TokenCollector}
  | \xa3 {TokenCollector}
  | \xa4 {TokenCollector}
  | \xa5 {TokenCollector}
  | \xa6 {TokenCollector}
  | \xa7 {TokenCollector}
  | \xa8 {TokenCollector}
  | \xa9 {TokenCollector}
  | \xaa {TokenCollector}
  | \xab {TokenCollector}
  | \xac {TokenCollector}
  | \xad {TokenCollector}
  | \xae {TokenCollector}
  | \xaf {TokenCollector}
  | \xb0 {TokenCollector}
  | \xb1 {TokenCollector}
  | \xb2 {TokenCollector}
  | \xb3 {TokenCollector}
  | \xb4 {TokenCollector}
  | \xb5 {TokenCollector}
  | \xb6 {TokenCollector}
  | \xb7 {TokenCollector}
  | \xb8 {TokenCollector}
  | \xb9 {TokenCollector}
  | \xba {TokenCollector}
  | \xbb {TokenCollector}
  | \xbc {TokenCollector}
  | \xbd {TokenCollector}
  | \xbe {TokenCollector}
  | \xbf {TokenCollector}
##utf8s2 -> \xc0 {TokenCollector}
  | \xc1 {TokenCollector}
  | \xc2 {TokenCollector}
```

```
  | \xc3 {TokenCollector}
  | \xc4 {TokenCollector}
  | \xc5 {TokenCollector}
  | \xc6 {TokenCollector}
  | \xc7 {TokenCollector}
  | \xc8 {TokenCollector}
  | \xc9 {TokenCollector}
  | \xca {TokenCollector}
  | \xcb {TokenCollector}
  | \xcc {TokenCollector}
  | \xcd {TokenCollector}
  | \xce {TokenCollector}
  | \xcf {TokenCollector}
  | \xd0 {TokenCollector}
  | \xd1 {TokenCollector}
  | \xd2 {TokenCollector}
  | \xd3 {TokenCollector}
  | \xd4 {TokenCollector}
  | \xd5 {TokenCollector}
  | \xd6 {TokenCollector}
  | \xd7 {TokenCollector}
  | \xd8 {TokenCollector}
  | \xd9 {TokenCollector}
  | \xda {TokenCollector}
  | \xdb {TokenCollector}
  | \xdc {TokenCollector}
  | \xdd {TokenCollector}
  | \xde {TokenCollector}
  | \xdf {TokenCollector}
##utf8s3 -> \xe0 {TokenCollector}
  | \xe1 {TokenCollector}
  | \xe2 {TokenCollector}
  | \xe3 {TokenCollector}
  | \xe4 {TokenCollector}
  | \xe5 {TokenCollector}
  | \xe6 {TokenCollector}
  | \xe7 {TokenCollector}
  | \xe8 {TokenCollector}
  | \xe9 {TokenCollector}
  | \xea {TokenCollector}
  | \xeb {TokenCollector}
  | \xec {TokenCollector}
  | \xed {TokenCollector}
  | \xee {TokenCollector}
  | \xef {TokenCollector}
##utf8s4 -> \xf0 {TokenCollector}
  | \xf1 {TokenCollector}
  | \xf2 {TokenCollector}
  | \xf3 {TokenCollector}
  | \xf4 {TokenCollector}
  | \xf5 {TokenCollector}
  | \xf6 {TokenCollector}
  | \xf7 {TokenCollector}
AllOfSpec -> a l l O f > ##blanks EqLink {TypeSheets::AllOfSpec(7=equations)}
Alternative -> ElementLink {RecordCaster: 0 -> TypeSheets::Alternative}
```

```
AlternativeLink -> Alternative AlternativeLink#1
                     {TypeSheets::AlternativeLink(0=Alternative, 1=*)}
ArraySpec -> a r r a y > ##blanks EqLink {TypeSheets::ArraySpec(7=equations)}
AtomicSpec -> a t o m i c > ##blanks NameLink {TypeSheets::AtomicSpec(8=names)}
Blanks -> SP ##blanks [taboo: Blanks#1]
            {ExternalCaster: String -> TypeSheets::Blanks}
CRange -> PosInt CRange#1 {TypeSheets::CRange(0=start, 1=*)}
CRangeLink -> CRange CRangeLink#1 {TypeSheets::CRangeLink(0=CRange, 1=*)}
CatDef -> ##eol CatDef#1 Id : CatDef#2 LF CatDef#3 SpecLink CatDef#4
            {TypeSheets::CatDef(2=Category, 7=specifications, 1=*, 4=*, 6=*,
                                8=*)}
CatLink -> CatProduction CatLink#1 {TypeSheets::CatLink(0=production, 1=*)}
CatProduction -> ##blanks # Id > SP CatProduction#1
                   {TypeSheets::CatProduction(2=target, 5=*)}
CatVar -> # CatVarRec {RecordCaster: 1 -> TypeSheets::CatVar}
  | # CatVarName {RecordCaster: 1 -> TypeSheets::CatVar}
  | # CatVarId {RecordCaster: 1 -> TypeSheets::CatVar}
CatVarId -> = CatVarId#1 {TypeSheets::CatVarId(1=*)}
CatVarName -> > CatVarName#1 {TypeSheets::CatVarName(1=*)}
CatVarRec -> Id {TypeSheets::CatVarRec(0=name)}
Char -> ##litchar {ExternalCaster: String -> TypeSheets::Char}
CharLink -> Char CharLink#1 {TypeSheets::CharLink(0=Char, 1=*)}
Comment -> ##blanks ! ##blanks Line LF {TypeSheets::Comment(3=text)}
CommentLink -> Comment CommentLink#1 {TypeSheets::CommentLink(0=Comment, 1=*)}
CompleteSpec -> c o m p l e t e > ##blanks LF {TypeSheets::CompleteSpec()}
Element -> CatVar Element#1 {TypeSheets::Element(0=CatVar, 1=*)}
  | LitVar Element#1 {TypeSheets::Element(0=LitVar, 1=*)}
  | Char Element#1 {TypeSheets::Element(0=literal, 1=*)}
  | LitId Element#1 {TypeSheets::Element(0=LitId, 1=*)}
  | Function Element#1 {TypeSheets::Element(0=Function, 1=*)}
  | Blanks {TypeSheets::Element(0=Blanks)}
  | LineBreak {TypeSheets::Element(0=LineBreak)}
  | & [ AlternativeLink & ] Element#2 {TypeSheets::Element(2=optional, 5=*)}
  | & ( AlternativeLink & ) Element#3 {TypeSheets::Element(2=once, 5=*)}
  | & { AlternativeLink & } Element#4 {TypeSheets::Element(2=anyTimes, 5=*)}
  | & < AlternativeLink & > Element#5 {TypeSheets::Element(2=multiple, 5=*)}
ElementLink -> Element ElementLink#1 {TypeSheets::ElementLink(0=element, 1=*)}
EntryLink -> EntryLink#1 EntryLink#2 {TypeSheets::EntryLink(0=*, 1=*)}
EqLink -> EqLink#1 Id = Id EqLink#2 EqLink#3
            {TypeSheets::EqLink(1=field, 3=Category, 0=*, 4=*, 5=*)}
Except -> & ^ Element {TypeSheets::Except(2=pattern)}
Expect -> & E Element {TypeSheets::Expect(2=pattern)}
FieldLink -> Id FieldLink#1 {TypeSheets::FieldLink(0=field, 1=*)}
FixedSpec -> f i x e d > ##blanks EqLink {TypeSheets::FixedSpec(7=equations)}
ForeignId -> ##foreignword {ExternalCaster: String -> TypeSheets::ForeignId}
FunArg -> ElementLink {RecordCaster: 0 -> TypeSheets::FunArg}
FunArgLink -> FunArg FunArgLink#1 {TypeSheets::FunArgLink(0=FunArg, 1=*)}
Function -> # ! Id ##blanks & ( Function#1 & ) {TypeSheets::Function(2=name, 6=*)}
Header -> Id ( Id , ##blanks Id ) : : Header#1 ##eol CommentLink Header#3
            {TypeSheets::Header(0=TypeSystem, 2=language, 5=authority,
                                11=comments, 9=*, 12=*)}
Id -> ##id {ExternalCaster: String -> TypeSheets::Id}
IdTranslationLink -> : : ForeignId : ##blanks Id ##blanks LF IdTranslationLink#1
                       {TypeSheets::IdTranslationLink(2=ForeignId, 5=Id, 8=*)}
ImportLink -> Id ImportLink#1 ImportLink#3
```

```
                {TypeSheets::ImportLink(0=TypeSystem, 1=*, 2=*)}
IndexSpec -> i n d e x > ##blanks EqLink {TypeSheets::IndexSpec(7=equations)}
IrrLink -> IrrProduction IrrLink#1 {TypeSheets::IrrLink(0=production, 1=*)}
IrrProduction -> ##blanks # Id > > SP # Id = AlternativeLink IrrProduction#1
                   {TypeSheets::IrrProduction(2=target, 7=field, 9=production,
                                                    10=*)}
ItselfSpec -> i t s e l f > ##blanks NameLink {TypeSheets::ItselfSpec(8=names)}
Line -> ##line {ExternalCaster: String -> TypeSheets::Line}
LineBreak -> & & LF {TypeSheets::LineBreak()}
LitDef -> ##eol LitDef#1 : LitDef#2 LitLink
            {TypeSheets::LitDef(4=productions, 1=*, 3=*)}
LitId -> & = & ( ElementLink & ) {TypeSheets::LitId(4=pattern)}
LitLink -> LitProduction LitLink#1 {TypeSheets::LitLink(0=production, 1=*)}
LitProduction -> ##blanks # Id > > SP # # Id = Substitution LitProduction#1
                   {TypeSheets::LitProduction(2=target, 8=field,
                                                    10=Substitution, 11=*)}
  | ##blanks # Id > > SP # # Id = AlternativeLink LitProduction#2
      {TypeSheets::LitProduction(2=target, 8=field, 10=production, 11=*)}
  | ##blanks # Id > > SP # # Id = # [ CRangeLink ] LitProduction#3
      {TypeSheets::LitProduction(2=target, 8=field, 12=cClass, 14=*)}
LitVar -> # # Id {TypeSheets::LitVar(2=name)}
MatchCase -> ##blanks Maximal {RecordCaster: 1 -> TypeSheets::MatchCase}
  | ##blanks Expect {RecordCaster: 1 -> TypeSheets::MatchCase}
  | ##blanks Taboo {RecordCaster: 1 -> TypeSheets::MatchCase}
  | ##blanks Except {RecordCaster: 1 -> TypeSheets::MatchCase}
Maximal -> & + {ExternalCaster: String -> TypeSheets::Maximal}
NameLink -> NameLink#1 Id NameLink#2 {TypeSheets::NameLink(1=field, 0=*, 2=*)}
NothingElseSpec -> n o t h i n g E l s e > ##blanks LF
                     {TypeSheets::NothingElseSpec()}
NothingSpec -> n o t h i n g > ##blanks LF {TypeSheets::NothingSpec()}
OneOfSpec -> o n e O f > ##blanks EqLink {TypeSheets::OneOfSpec(7=equations)}
OnlySpec -> o n l y > ##blanks EqLink {TypeSheets::OnlySpec(6=equations)}
OptionalSpec -> o p t i o n a l > ##blanks EqLink
                  {TypeSheets::OptionalSpec(10=equations)}
OutChar -> ##outchar {ExternalCaster: String -> TypeSheets::OutChar}
OutCharLink -> OutChar OutCharLink#1 {TypeSheets::OutCharLink(0=OutChar, 1=*)}
PosInt -> ##digits {ExternalCaster: Integer -> TypeSheets::PosInt}
SomeOfSpec -> s o m e O f > ##blanks EqLink
                {TypeSheets::SomeOfSpec(8=equations)}
SomeOfTypeSpec -> s o m e O f T y p e > ##blanks EqLink
                    {TypeSheets::SomeOfTypeSpec(12=equations)}
Spec -> AllOfSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | OneOfSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | SomeOfSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | OptionalSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | FixedSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | OnlySpec {RecordCaster: 0 -> TypeSheets::Spec}
  | SomeOfTypeSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | ItselfSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | ArraySpec {RecordCaster: 0 -> TypeSheets::Spec}
  | IndexSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | TemplateSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | NothingElseSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | NothingSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | UnionSpec {RecordCaster: 0 -> TypeSheets::Spec}
```

```
  | AtomicSpec {RecordCaster: 0 -> TypeSheets::Spec}
  | CompleteSpec {RecordCaster: 0 -> TypeSheets::Spec}
SpecLink -> Spec SpecLink#1 {TypeSheets::SpecLink(0=Spec, 1=*)}
StartCategory -> : S T A R T = Id StartCategory#1
                   {TypeSheets::StartCategory(7=start, 8=*)}
Substitution -> CharLink & " Substitution#1 "
                  {TypeSheets::Substitution(0=input, 3=*)}
Taboo -> & T Element {TypeSheets::Taboo(2=pattern)}
Target -> FieldLink = VarLink Target#1
            {TypeSheets::Target(0=sequence, 2=vars, 3=*)}
TargetLink -> Target TargetLink#1 {TypeSheets::TargetLink(0=Target, 1=*)}
Targets -> ##eol Targets#1 : T A R G E T > ##blanks TargetLink
             {TypeSheets::Targets(11=targets, 1=*)}
TemplateSpec -> t e m p l a t e > ##blanks Id ##blanks LF
                  {TypeSheets::TemplateSpec(10=name)}
TranslationLink -> ##eol TranslationLink#1 : L A N G : ##blanks Id ##blanks LF
                     IdTranslationLink TranslationLink#2
                     {TypeSheets::TranslationLink(9=language, 12=idTranslations,
                                                  1=*, 13=*)}
TypeSheet -> Header TypeSheet#1 TypeSheet#2 EntryLink TypeSheet#3 TypeSheet#4
               {TypeSheets::TypeSheet(0=Header, 3=entries, 1=*, 2=*, 4=*, 5=*)}
UnionSpec -> u n i o n > ##blanks NameLink {TypeSheets::UnionSpec(7=names)}
VarLink -> # Id VarLink#1 {TypeSheets::VarLink(1=name, 2=*)}
##blanks#1 -> SP ##blanks#1 {TokenCollector}
  | [taboo: ##blanks#2] {TokenCollector}
##blanks#2 -> SP {TokenCollector}
##digits#1 -> ##digit ##digits#1 {TokenCollector}
  | ##digit [taboo: ##digits#2] {TokenCollector}
##digits#2 -> ##digit {TokenCollector}
##eol#1 -> LF ##eol#1 {TokenCollector}
  | LF [taboo: ##eol#2] {TokenCollector}
##eol#2 -> LF {TokenCollector}
##foreignword#1 -> ##foreignword#3 ##hchar ##foreignword#1 {TokenCollector}
  | ##foreignword#3 ##hchar {TokenCollector}
##foreignword#2 -> : {TokenCollector}
##foreignword#3 -> [taboo: ##foreignword#2] {TokenCollector}
##id#1 -> ##id#2 ##id#1 {TokenCollector}
  | [taboo: ##id#3] {TokenCollector}
##id#2 -> ##letter {TokenCollector}
  | ##digit {TokenCollector}
##id#3 -> ##id#2 {TokenCollector}
##line#1 -> ##hchar ##line#1 {TokenCollector}
  | [taboo: ##line#2] {TokenCollector}
##line#2 -> ##hchar {TokenCollector}
##litchar#1 -> ##litchar#2 {TokenCollector}
##litchar#2 -> ! {TokenCollector}
  | # {TokenCollector}
  | & {TokenCollector}
  | SP {TokenCollector}
##litchar#3 -> [taboo: ##litchar#1] {TokenCollector}
##outchar#1 -> " {TokenCollector}
##outchar#2 -> [taboo: ##outchar#1] {TokenCollector}
AlternativeLink#1 -> & | AlternativeLink {2=next}
  | {}
Blanks#1 -> ! {TokenCollector}
```

```
CRange#1 -> - PosInt {1=end}
  | {}
CRangeLink#1 -> , ##blanks CRangeLink {2=next}
  | {}
CatDef#1 -> CommentLink {0=preComments}
  | {}
CatDef#2 -> SP Id + {1=extends}
  | {}
CatDef#3 -> CommentLink {0=postComments}
  | {}
CatDef#4 -> CatLink CatDef#5 {0=productions, 1=*}
  | {}
CatDef#5 -> IrrLink {0=irregular}
  | {}
CatLink#1 -> CatLink {0=next}
  | {}
CatProduction#1 -> AlternativeLink CatProduction#2 {0=production, 1=*}
  | # CatProduction#3 {1=*}
CatProduction#2 -> CommentLink {0=comments}
  | LF {}
CatProduction#3 -> SP CommentLink {1=comments}
  | ##blanks LF {}
CatVarId#1 -> Id {0=name}
  | & t h i s {}
CatVarName#1 -> Id {0=name}
  | & t h i s {}
CharLink#1 -> CharLink {0=next}
  | {}
CommentLink#1 -> CommentLink {0=next}
  | {}
Element#1 -> MatchCase {0=match}
  | {}
Element#2 -> MatchCase {0=match}
  | {}
Element#3 -> MatchCase {0=match}
  | {}
Element#4 -> MatchCase {0=match}
  | {}
Element#5 -> MatchCase {0=match}
  | {}
ElementLink#1 -> ElementLink {0=next}
  | {}
EntryLink#1 -> LitDef {0=LitDef}
  | CatDef {0=CatDef}
EntryLink#2 -> EntryLink {0=next}
  | {}
EqLink#1 -> # Id : {1=name}
  | {}
EqLink#2 -> Comment {0=comment}
  | ##blanks LF {}
EqLink#3 -> ##blanks EqLink {1=next}
  | {}
FieldLink#1 -> . FieldLink {1=next}
  | {}
FunArgLink#1 -> & , FunArgLink {2=next}
```

```
  | {}
Function#1 -> FunArgLink {0=args}
  | {}
Header#1 -> Header#2 ImportLink {1=imports, 0=*}
  | {}
Header#2 -> ##blanks {}
  | ##eol ##blanks {}
Header#3 -> TranslationLink {0=translations}
  | {}
IdTranslationLink#1 -> IdTranslationLink {0=next}
  | {}
ImportLink#1 -> . Id ImportLink#2 {1=Category, 2=*}
  | {}
ImportLink#2 -> - > Id {2=newName}
  | {}
ImportLink#3 -> , ImportLink#4 ImportLink {2=next, 1=*}
  | {}
ImportLink#4 -> ##blanks {}
  | ##eol ##blanks {}
IrrLink#1 -> IrrLink {0=next}
  | {}
IrrProduction#1 -> CommentLink {0=comments}
  | LF {}
LitDef#1 -> CommentLink {0=preComments}
  | {}
LitDef#2 -> CommentLink {0=postComments}
  | ##blanks LF {}
LitLink#1 -> LitLink {0=next}
  | {}
LitProduction#1 -> CommentLink {0=comments}
  | ##blanks LF {}
LitProduction#2 -> CommentLink {0=comments}
  | LF {}
LitProduction#3 -> CommentLink {0=comments}
  | ##blanks LF {}
NameLink#1 -> # Id : {1=name}
  | {}
NameLink#2 -> , NameLink#3 NameLink {2=next, 1=*}
  | LF {}
NameLink#3 -> SP {}
  | ##eol ##blanks {}
OutCharLink#1 -> OutCharLink {0=next}
  | {}
SpecLink#1 -> SpecLink {0=next}
  | {}
StartCategory#1 -> Comment {0=comment}
  | ##blanks LF {}
Substitution#1 -> OutCharLink {0=output}
  | {}
Target#1 -> Comment {0=comment}
  | ##blanks LF {}
TargetLink#1 -> ##blanks TargetLink {1=next}
  | {}
Targets#1 -> CommentLink {0=preComments}
  | {}
```

```
TranslationLink#1 -> CommentLink {0=preComments}
  | {}
TranslationLink#2 -> TranslationLink {0=next}
  | {}
TypeSheet#1 -> Targets {0=Targets}
  | {}
TypeSheet#2 -> StartCategory {0=StartCategory}
  | {}
TypeSheet#3 -> ##eol CommentLink {1=postComments}
  | {}
TypeSheet#4 -> ##eol {}
  | {}
VarLink#1 -> SP VarLink {1=next}
  | {}
```

Appendix C

Technical Reports

This appendix lists and briefly summarizes the technical reports authored or co-authored by me that relate to the research for this thesis.

[TR1] *K. Kofler. FMathL Formal Mathematical Language and how it relates to the Grammatical framework (GF). Slides, 2009. 12 slides.* (Kofler [48])
`http://www.grammaticalframework.org/doc/gfss/gfss-kevin.pdf`
These slides were presented by me at the GF Resource Grammar Summer School 2009. They give an introduction to the FMathL project and to its state of research as of 2009. Then, they present ways to interoperate with GF that were considered at the time. The solution proposed there was eventually implemented: support for PGF files (compiled grammars from GF) in DynGenPar.

[TR2] *K. Kofler and A. Neumaier. Limitations in Content MathML. Technical report, University of Vienna, 2009. 5 pages.* (Kofler & Neumaier [52])
`http://www.mat.univie.ac.at/~neum/FMathL/content-mathml-limitations.pdf`
This technical report summarizes issues and limitations with Content MathML we encountered during our effort to represent example formulas that reflect common usage in mathematical texts. It concludes that Content MathML is not an adequate representation for the goals of the FMathL and Concise projects, at least at the time the assessment was made.

[TR3] *K. Kofler and A. Neumaier. Limitations in OpenMath. Technical report, University of Vienna, 2010. 3 pages.* (Kofler & Neumaier [53])
`http://www.mat.univie.ac.at/~neum/FMathL/openmath-limitations.pdf`
This technical report summarizes, in the same vein as the one above, the issues and limitations with OpenMath we encountered during our effort to represent example formulas that reflect common usage in mathematical texts. It concludes that, like Content MathML, OpenMath is not an adequate representation for the goals of the FMathL and Concise projects, at least at the time the assessment was made.

[TR4] *K. Kofler and A. Neumaier. A Dynamic Generalized Parser for Common Mathematical Language. Work in Progress paper at CICM, 2011. 10 pages.* (Kofler & Neumaier [54])
`https://www.tigen.org/kevin.kofler/fmathl/dyngenpar-wip.pdf`

This technical report introduced the DynGenPar algorithm and implementation to the community. It was accepted as a work-in-progress paper in the Mathematical Knowledge Management (MKM) track of the Conference on Intelligent Computer Mathematics (CICM) in 2011. It was included in the informal work-in-progress proceedings and presented in a lightning talk. The contents roughly correspond to Chapter 2 of this thesis. A revised and expanded version of this paper (Kofler & Neumaier [56]) was fully reviewed and published in the following year's CICM 2012, in the Digital Mathematics Library (DML) track.

[TR5] *K. Kofler and A. Neumaier. The DynGenPar Algorithm on an Example. Slides, 2011. 27 slides.* (Kofler & Neumaier [55])
`https://www.tigen.org/kevin.kofler/fmathl/dyngenpar-example.pdf`
These slides show, on an example, how the DynGenPar algorithm works and how it compares with existing approaches (pure top-down and pure bottom-up parsing). They serve as an online supplement for the above technical report. I also used them in the corresponding lightning talk and in a few other presentations.

[TR6] *K. Kofler. The Concise Record Transformation Language. Technical report, DAGOPT Optimization Technologies GmbH, 2016. 15 pages.* (Kofler [49])
`https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/rectrans.pdf`
This technical report documents the Concise record transformation language. It is often necessary in Concise to convert records from one representation (in one type system) to another (in a different type system). For this purpose, Concise supports record transformations. They are parsed with DynGenPar and executed by Concise. See Sections 3.3.3 and 4.3. (The latter is based on this technical report.) This technical report introduces the design principles of the record transformation language and completely documents its syntax.

[TR7] *K. Kofler and A. Neumaier. Programming in Concise – Code Sheets and Elementary Acts. Technical report, University of Vienna, 2017. 96 pages.* (Kofler & Neumaier [57])
`https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/codesheets.pdf`
This technical report presents the implementation of the Concise programming language. It contains:

- the type sheet `ElementaryActs.cnt` (by A. Neumaier et al.), defining the low-level elementary acts (see Section 4.2.1),
- the type sheet `CodeSheets.cnt` (by me) defining the code sheets (see Section 4.2.2), i.e., the higher-level text representation,
- my two implementations of the record transformation from code sheets to elementary acts (see Section 4.2.2).

[TR8] *K. Kofler and A. Baharev. A Modeling Language for Chemical Processes. Technical report, University of Vienna, 2017. 42 pages.* (Kofler & Baharev [51])
`https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/chemprocmod.pdf`
This technical report presents the implementation of `ChemProcMod`, the modeling language for chemical processes described in Section 4.4 of this thesis. It contains the `ChemProcMod.cnt` type sheet, as well as several model files in the `ChemProcMod` language:

the unit library `UnitLibrary.cpm` and a few example models.

[TR9] *K. Kofler. DynGenPar API documentation. Technical report, DAGOPT Optimization Technologies GmbH, 2017. 183 pages.* (Kofler [50])
`https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/dyngenpar.pdf`
This technical report was produced from the formatted comments I wrote in the source code of DynGenPar, using the **Doxygen** tool (van Heesch [94]). It documents the complete application programming interface (API) for DynGenPar, i.e., all the classes, methods, and functions available for applications to use. The API is available for use in both C++ and Java applications.

Bibliography

[1] A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972. Cited on page 13.

[2] AMPL Optimization inc. AMPL – Streamlined Modeling for Real Optimization. Website. `http://ampl.com/`. Cited on pages 18, 47, 82, 127, and 137.

[3] K. Angelov. Incremental parsing with parallel multiple context-free grammars. *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 69–76, 2009. Cited on page 16.

[4] K. Angelov, B. Bringert, and A. Ranta. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, 19(2): 201–228, 2010. ISSN 0925-8531. Cited on pages 16, 86, 131, and 137.

[5] A. Baharev, L. Kolev, and E. Rév. Computing multiple steady states in homogeneous azeotropic and ideal two-product distillation. *AIChE Journal*, 57:1485–1495, 2011. Cited on page 74.

[6] A. Baharev and A. Neumaier. Chemical Process Modeling in Modelica. In *Proceedings of the 9th International Modelica Conference*, pp. 955–962, September 2012. Cited on pages 72 and 74.

[7] A. Baharev and A. Neumaier. A globally convergent method for finding all steady-state solutions of distillation columns. *AIChE Journal*, 60:410–414, 2014. Cited on page 72.

[8] D.M. Beazley. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of USENIX Tcl/Tk Workshop*, 1996. Cited on page 121.

[9] A. Brooke, D.A. Kendrick, A. Meeraus, and R.E. Rosenthal. *GAMS: A user's guide*. Course Technology, 1988. Cited on page 127.

[10] C. Burali-Forti. Una questione sui numeri transfiniti. *Rendiconti del Circolo Matematico di Palermo (1884-1940)*, 11(1):154–164, 1897. Cited on page 110.

[11] S. Buswell, O. Caprotti, D.P. Carlisle, M.C. Dewar, M. Gaetano, and M. Kohlhase. The open math standard [version 2.0]. Technical report, The Open Math Society, 2004. `http://www.openmath.org/standard/om20`. Cited on page 139.

[12] N. Chomsky. A note on phrase structure grammars. *Information and control*, 2(4): 393–395, 1959. Cited on page 13.

[13] N. Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959. Cited on page 13.

[14] A.R. Ciric and D. Gu. Synthesis of nonequilibrium reactive distillation processes by MINLP optimization. *AIChE Journal*, 40:1479–1487, 1994. Cited on page 74.

[15] A.R. Ciric and P. Miao. Steady State Multiplicities in an Ethylene Glycol Reactive Distillation Column. *Ind. Eng. Chem. Res.*, 33:2738–2748, 1994. Cited on page 74.

[16] G. Costagliola, V. Deufemia, and G. Polese. Visual language implementation through standard compiler-compiler techniques. *Journal of Visual Languages & Computing*, 18(2):165–226, 2007. Selected papers from Visual Languages and Computing 2005 (VLC '05). Cited on page 16.

[17] M. Cramer, B. Fisseni, P. Koepke, D. Kühlwein, B. Schröder, and J. Veldman. The Naproche Project – Controlled Natural Language Proof Checking of Mathematical Texts. In Norbert E. Fuchs, editor, *CNL*, vol. 5972 of *LNCS*, pp. 170–186. Springer, 2009. ISBN 978-3-642-14417-2. Cited on pages 17, 19, 107, 108, 129, 130, and 138.

[18] M. Cramer and D. Kühlwein. The Controlled Natural Language of Naproche in a nutshell, In *Naproche Wiki*. Website. https://korpora-exp.zim.uni-duisburg-essen.de/NAPROCHE-WIKI/doku.php?id=dokumentation:language. Cited on pages 108 and 109.

[19] L.E. de Souza Amorim, S. Erdweg, G. Wachsmuth, and E. Visser. Principled syntactic code completion using placeholders. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pp. 163–175. ACM, 2016. Cited on page 44.

[20] F.L. DeRemer. *Practical Translators for LR(k) languages.* PhD thesis, Massachusetts Institute of Technology (MIT), 1969. http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-065.pdf. Cited on pages 15 and 108.

[21] A. Desitter, C. Reitzel, B. Höhrmann, et al. HTML Tidy Project Page. Website. http://tidy.sourceforge.net/. Cited on page 53.

[22] F. Domes. Concise. Website. http://www.mat.univie.ac.at/~dferi/concise.html. Cited on pages 14, 17, 18, 28, 29, 33, 60, 62, 63, 64, 65, 66, 72, and 139.

[23] F. Domes. GloptLab. Website. http://www.mat.univie.ac.at/~dferi/gloptlab.html. Cited on pages 83 and 127.

[24] F. Domes. Gloptlab: a configurable framework for the rigorous global solution of quadratic constraint satisfaction problems. *Optimization Methods & Software*, 24 (4-5):727–747, 2009. Cited on pages 83 and 127.

[25] F. Domes, A. Neumaier, K. Kofler, P. Schodl, and H. Schichl. *Concise Manual*, 2012. `http://www.mat.univie.ac.at/~dferi/concise/Manual.pdf`. Cited on page 18.

[26] Flex Project. flex: The Fast Lexical Analyzer. Website. `https://github.com/westes/flex`. Cited on pages 30, 108, 130, and 138.

[27] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A mathematical programming language*. AT&T Bell Laboratories, 1987. Cited on pages 18, 47, 82, 127, and 137.

[28] Free Software Foundation. Bison – GNU parser generator. Website. `http://www.gnu.org/software/bison`. Cited on pages 108, 129, 130, and 138.

[29] Free Software Foundation. GCC, the GNU Compiler Collection. Website. `http://gcc.gnu.org/`. Cited on page 28.

[30] Free Software Foundation. GNU General Public License (GPL) v2.0. Website, June 1991. `http://www.gnu.org/licenses/old-licenses/gpl-2.0`. Cited on page 29.

[31] Free Software Foundation. GNU General Public License (GPL) v3.0. Website, June 2007. `http://www.gnu.org/licenses/gpl-3.0`. Cited on page 29.

[32] W. Fulton et al. Simplified Wrapper and Interface Generator. Website. `http://www.swig.org/`. Cited on page 121.

[33] GAMS Development Corp. GAMS – Cutting Edge Modeling. Website. `https://www.gams.com/`. Cited on page 127.

[34] T. Hallgren. The Proof Editor Alfa. Website. `http://www.cse.chalmers.se/~hallgren/Alfa/`. Cited on page 17.

[35] T. Hallgren and A. Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning*, pp. 70–84. Springer, Springer, 2000. Cited on page 17.

[36] R. Hinze and R. Paterson. Derivation of a typed functional LR parser, 2003. Cited on page 16.

[37] HTACG. HTML Tidy. Website. `http://www.html-tidy.org/`. Cited on page 53.

[38] M. Humayoun. Controlled language of mathematics (CLM). Website. `http://www.lama.univ-savoie.fr/~humayoun/phd/grammar/grammar.html`. Cited on page 120.

[39] M. Humayoun. MathNat (Mathematical texts in Controlled Natural language). Website. `http://www.lama.univ-savoie.fr/~humayoun/phd/mathnat.html`. Cited on pages 17, 19, 107, 119, and 138.

[40] M. Humayoun. *Developing the System MathNat for Automatic Formalization of Mathematical texts*. PhD thesis, University of Grenoble, 2012. `http://www.lama.univ-savoie.fr/~humayoun/phd/thesis.html`. Cited on pages 17, 19, 107, 119, and 138.

[41] M. Humayoun and C. Raffalli. MathNat – Mathematical Text in a Controlled Natural Language. *Journal on Research in Computing Science*, 46(Special issue: Natural Language Processing and its Applications), 2010. Cited on pages 17, 19, 107, 119, and 138.

[42] E.W. Jacobsen and S. Skogestad. Multiple steady states in ideal two-product distillation. *AIChE Journal*, 37:499–511, 1991. Cited on page 74.

[43] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965. Cited on pages 16, 28, and 136.

[44] KDE Webmasters. KDE Community Home – KDE.org. Website. `https://www.kde.org/`. Cited on page 97.

[45] D.E. Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965. Cited on pages 14, 15, 21, and 22.

[46] P. Koepke, B. Schröder, G. Buechel, et al. Naproche – Natural language proof checking. Website. `http://www.naproche.net`. Cited on pages 17, 19, 107, 108, 129, 130, and 138.

[47] K. Kofler. DynGenPar – Dynamic Generalized Parser. Website. `https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/`. Cited on page 29.

[48] K. Kofler. FMathL Formal Mathematical Language and how it relates to the Grammatical framework (GF). Slides, 2009. `http://www.grammaticalframework.org/doc/gfss/gfss-kevin.pdf`. Cited on page 181.

[49] K. Kofler. The Concise Record Transformation Language. Technical report, DAGOPT Optimization Technologies GmbH, 2016. `https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/rectrans.pdf`. Cited on pages 59 and 182.

[50] K. Kofler. DynGenPar API documentation. Technical report, DAGOPT Optimization Technologies GmbH, 2017. `https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/dyngenpar.pdf`. Cited on pages 28 and 183.

[51] K. Kofler and A. Baharev. A Modeling Language for Chemical Processes. Technical report, University of Vienna, 2017. `https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/chemprocmod.pdf`. Cited on pages 73, 74, 141, and 182.

[52] K. Kofler and A. Neumaier. Limitations in Content MathML. Technical report, University of Vienna, 2009. `http://www.mat.univie.ac.at/~neum/FMathL/content-mathml-limitations.pdf`. Cited on page 181.

[53] K. Kofler and A. Neumaier. Limitations in OpenMath. Technical report, University of Vienna, 2010. `http://www.mat.univie.ac.at/~neum/FMathL/openmath-limitations.pdf`. Cited on page 181.

[54] K. Kofler and A. Neumaier. A Dynamic Generalized Parser for Common Mathematical Language. *Work in Progress paper at CICM*, 2011. https://www.tigen.org/kevin.kofler/fmathl/dyngenpar-wip.pdf. Cited on pages 21 and 181.

[55] K. Kofler and A. Neumaier. The DynGenPar Algorithm on an Example. Slides, 2011. https://www.tigen.org/kevin.kofler/fmathl/dyngenpar-example.pdf. Cited on page 182.

[56] K. Kofler and A. Neumaier. DynGenPar – A Dynamic Generalized Parser for Common Mathematical Language. In *Proceedings of CICM 2012 (DML track)*, vol. 7362 of *LNAI*, pp. 386–401. Springer, 2012. Cited on pages 3, 4, 14, 15, 16, 18, 21, 87, 129, 130, 131, 132, 139, and 182.

[57] K. Kofler and A. Neumaier. Programming in Concise – Code Sheets and Elementary Acts. Technical report, University of Vienna, 2017. https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/codesheets.pdf. Cited on pages 55, 56, 59, 141, and 182.

[58] M. Kohlhase. *OMDoc – An Open Markup Format for Mathematical Documents [version 1.2]*, vol. 4180. Springer, 2006. Foreword by Alan Bundy. Cited on page 139.

[59] M. Kohlhase. Using LaTeX as a Semantic Markup Format. *Mathematics in Computer Science*, 2.2:279–304, 2008. Cited on pages 116, 120, and 127.

[60] M. Kohlhase and D. Ginev. arXMLiv. Website. https://kwarc.info/projects/arXMLiv. Cited on page 111.

[61] M. Kohlhase and I. Sucan. A search engine for mathematical formulae. In *Artificial Intelligence and Symbolic Computation*, pp. 241–253. Springer, 2006. Cited on page 139.

[62] D. Kühlwein. Ein Kalkül für Beweisrepräsentationsstrukturen (A calculus for Proof Representation Structures). Master's thesis, University of Bonn, 2009. English with German title and abstract. https://korpora-exp.zim.uni-duisburg-essen.de/naproche/inc/downloads.php. Cited on page 108.

[63] D. Langer. Das automatische Verarbeiten mathematischer Formeln. Master's thesis, University of Vienna, 2013. http://othes.univie.ac.at/27773/. Cited on pages 121 and 138.

[64] S.E. Mattsson, H. Elmqvist, and J.F. Broenink. Modelica: An International effort to design the next generation modelling language. *Benelux Quarterly Journal on Automatic Control*, 38(3):16–19, September 1997. Cited on pages 72, 127, and 137.

[65] B.A. McCarl, A. Meeraus, P. van der Eijk, M. Bussieck, S. Dirkse, P. Steacy, and F. Nelissen. McCarl GAMS user guide. Website. http://www.gams.com/latest/docs/userguides/mccarl/. Cited on pages 82 and 127.

[66] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005. Cited on page 16.

[67] B. Miller. LaTeXML – A LaTeX to XML/HTML/MathML Converter. Website. `http://dlmf.nist.gov/LaTeXML/`. Cited on pages 107, 110, and 138.

[68] Modelica Association. Modelica and the Modelica Association. Website. `https://www.modelica.org/`. Cited on pages 72, 127, and 137.

[69] A. Neumaier. Analysis und lineare Algebra. Unpublished lecture notes. `http://www.mat.univie.ac.at/~neum/FMathL/ALA.pdf`. Cited on pages 18, 19, 119, 126, 129, and 134.

[70] A. Neumaier. FMathL – Formal Mathematical Language. Website. `http://www.mat.univie.ac.at/~neum/FMathL.html`. Cited on pages 14, 17, 33, 99, 119, and 120.

[71] A. Neumaier and H. Schichl. COCONUT – COntinuous CONstraints – Updating the Technology. Website. `http://www.mat.univie.ac.at/~neum/glopt/coconut/`. Cited on pages 75 and 127.

[72] A. Neumaier and H. Schichl. The COCONUT Benchmark. Website. `http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html`. Cited on page 82.

[73] A. Pichler. Semantische Repräsentation mathematischer Konzepte. Master's thesis, University of Vienna, 2016. `http://www.mat.univie.ac.at/~neum/ms/pichlerMaster.pdf`. Cited on pages 107 and 119.

[74] The Qt Company. Qt – Cross-platform software development for embedded & desktop. Website. `https://www.qt.io/`. Cited on pages 28 and 97.

[75] D. Raggett. Clean up your Web pages with HTML TIDY. Website. `https://www.w3.org/People/Raggett/tidy/`. Cited on page 53.

[76] D. Raggett. Clean up your Web pages with HP's HTML tidy. *Computer networks and ISDN systems*, 30(1):730–732, 1998. Cited on page 53.

[77] A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, 2004. ISSN 1469-7653. Cited on pages 15, 16, 18, 85, 86, 129, 130, 131, and 137.

[78] A. Ranta. *Grammatical framework: Programming with multilingual grammars*. CSLI Publications, Center for the Study of Language and Information, 2011. Cited on pages 15, 16, 18, 85, 86, 129, 130, 131, and 137.

[79] A. Ranta, K. Angelov, T. Hallgren, et al. GF – Grammatical Framework. Website. `http://www.grammaticalframework.org`. Cited on pages 15, 16, 18, 85, 86, 129, 130, 131, and 137.

[80] S. Reichelt. The HLM proof assistant. Website. http://hlm.sourceforge.net/. Cited on page 17.

[81] D.J. Salomon and G.V. Cormack. Scannerless NSLR (1) parsing of programming languages. *ACM SIGPLAN Notices*, 24(7):170–178, 1989. Cited on page 17.

[82] H. Schichl. Global Optimization in the COCONUT project. In *Numerical Software with Result Verification (Proceedings of the Dagstuhl Seminar)*, vol. 2991 of *LNCS*, pp. 243–249. Springer, 2004. Cited on pages 75 and 127.

[83] H. Schichl and R. Steinbauer. *Einführung in das mathematische Arbeiten*. Springer, 2012. Cited on pages 19, 126, 129, and 134.

[84] P. Schodl. *Foundations for a self-reflective, context-aware semantic representation of mathematical specification*. PhD thesis, University of Vienna, 2011. http://www.mat.univie.ac.at/~schodl/pdfs/diss_online.pdf. Cited on pages 17, 109, and 110.

[85] P. Schodl and A. Neumaier. An experimental grammar for German mathematical text. Technical report, University of Vienna, 2009. http://www.mat.univie.ac.at/~neum/FMathL/ALA-grammar.pdf. Cited on pages 18 and 119.

[86] P. Schodl and A. Neumaier. The FMathL type system. Technical report, University of Vienna, 2011. http://www.mat.univie.ac.at/~neum/FMathL/types.pdf. Cited on pages 37, 38, 65, 100, and 104.

[87] P. Schodl, A. Neumaier, K. Kofler, F. Domes, and H. Schichl. Towards a Self-reflective, Context-aware Semantic Representation of Mathematical Specifications. In Josef Kallrath, editor, *Algebraic Modeling Systems – Modeling and Solving Real World Optimization Problems*, chapter 2. Springer, 2012. Cited on pages 14, 17, 18, 28, 29, 33, 60, 62, 63, 64, 65, 66, 72, and 139.

[88] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229, 1991. ISSN 0304-3975. Cited on pages 15, 31, 39, 86, and 137.

[89] O. Shcherbina, A. Neumaier, D. Sam-Haroud, X.H. Vu, and T.V. Nguyen. Benchmarking global optimization and constraint satisfaction codes. In *Proceedings of International Workshop on Global Optimization and Constraint Satisfaction*, pp. 211–222. Springer, 2002. Cited on page 82.

[90] H. Stamerjohanns and M. Kohlhase. Transforming the arXiv to XML. In *Proceedings of CICM 2008 (MKM track)*, vol. 5144 of *LNAI*, pp. 574–582. Springer, 2008. Cited on page 111.

[91] M. Tomita. An Efficient Augmented Context-Free Parsing Algorithm. *Computational Linguistics*, 13(1–2):31–46, 1987. Cited on pages 14, 15, 16, 21, 22, and 30.

[92] M. Tomita and S.K. Ng. The Generalized LR parsing algorithm. In M. Tomita, editor, *Generalized LR parsing*, pp. 1–16. Kluwer, 1991. Cited on pages 14, 15, 16, 21, 22, and 30.

[93] M.G.J. Van Den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Proceedings of Compiler Construction 2002*, vol. 2304 of *LNCS*, pp. 143–158. Springer, 2002. Cited on page 17.

[94] D. van Heesch. Doxygen: Main Page. Website. `http://www.stack.nl/~dimitri/doxygen/`. Cited on page 183.

[95] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997. Cited on pages 16 and 17.

[96] S. Voutilainen et al. Qt Jambi – Qt Jambi is the Qt library made available to Java. Website. `http://qtjambi.org`. Cited on pages 28 and 37.

[97] W3C. Mathematical Markup Language (MathML) Version 3.0 2nd Edition. Website. `https://www.w3.org/TR/MathML3/`. Cited on page 139.

[98] W3C. RDF Primer. Website. `https://www.w3.org/TR/rdf-primer`. Cited on page 34.

[99] W3C. XML Path Language (XPath) 2.0 (Second Edition). Website. `https://www.w3.org/TR/xpath20/`. Cited on pages 60, 62, and 65.

[100] W3C. XSL Transformations (XSLT) Version 2.0. Website. `https://www.w3.org/TR/xslt20/`. Cited on pages 59, 60, 62, 65, and 111.

[101] D.H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208, 1967. Cited on pages 16, 28, and 136.

Index

A Mathematical Programming Language, 82
abstract grammar, 100
act, 54
Alfa, 17
alphabet, 13
alphabetic characters, 92
AMPL, 82
API, 28
application grammar, 100
application programming interface, 28
arXMLiv, 111
atomic unit, 73

Backus-Naur form, 39
`BasicDefinitions`, 116
BNF, 39

categorical production, 48
categories, 13
`ChemProcMod`, 72
Cocke-Younger-Kasami, 16, 28
COCONUT, 75
code sheet, 36, 37, 53, 56
coercion function, 88, 90
completion filter, 44
complex act, 54
composite unit, 73
concept, 117
`ConceptsFromLatex`, 119
Concise, 14, 17, 33
`ConciseTokenSource`, 114
concrete grammar, 100
content, 112
context-free grammar, 13
context-sensitive constraints, 13
CYK, 16, 28

DAG, 25, 30
directed acyclic graph, 25, 30
divider, 74
Document Object Model, 53, 111
DOM, 53, 111
Doxygen, 183
Dynamic Generalized Parser, 13, 21
DynGenPar, 13, 21

EBNF, 39
element list, 112
elementary act, 53, 54
elementary operations, 24
entry, 33
extended Backus-Naur form, 39
eXtended Positional LR, 16
eXtensible Markup Language, 37, 110
eXtensible Stylesheet Language Transformations, 59
external object, 34
external type, 34

field, 33
FMathL, 14, 17
FMathL LaTeX stylesheet, 111
Formal Mathematical Language, 14, 17
FormulaTokenSource, 121

Generalized LR, 14–16, 21
GF, 15, 16, 85
GLR, 14–16, 21
grammar, 13
Grammatical Framework, 15, 16, 85
graph view, 35

handle, 33
header, 112
help usage, 38
HLM Proof Assistant, 17
HTML Tidy, 53

informal list, 123
initial graph, 15, 22
inlet, 73
input sentence, 13
input text, 13
input usage, 38
invisible whitespace, 124

Java Native Interface, 37
JNI, 37

LaTeXML, 110
layout, 112
lexeme, 93
lexer state, 92
linearization, 42
literal, 44
literal production, 48
LR, 14, 21

match, 24
MathML, 139
MathNat, 17, 119
MathWebSearch, 139
model equation, 73

Naproche, 17, 108
neighborhood, 24
node, 82
nonterminals, 13

object, 33
OMDoc, 139
Open Mathematical Documents, 139
OpenMath, 139
`OptProbl`, 75
outlet, 73
output usage, 38

package sheet, 37
paragraph, 112
parallel multiple context-free grammar, 15, 31
parsable file view, 36
parse tree, 13
parser, 13
parsing, 13
PGF, 86
PGF GUI, 97
PGF runtime, 131
`PgfTokenSource`, 91
PMCFG, 15, 31
Portable Grammar Format, 86
process, 73
productions, 13

Qt, 28

Qt Jambi, 37

RDF, 34
record sheet, 36
record transformation, 41
record transformation language, 59
record transformation sheet, 41, 59
record view, 35
reduce, 24
reduce-reduce conflict, 26
referenceable items, 112
Resource Description Framework, 34
resource grammar, 85, 99
robust ampl, 82
rules, 13

SAX, 111
sem, 33
semantic graph, 33
semantic memory, 33
shift, 24
Simple API for XML, 111
start category, 13
start symbol, 13
stateful lexer, 92

target, 24
term, 112
text view, 35
text view usage, 38
`TextDocument`, 110
`TextFormulaIterator`, 116
`TextParagraphIterator`, 113
token hook, 115, 118
token source usage, 38
tokens, 13
trailer, 112
tree view, 35
type sheet, 36–38

unit library, 74
unlexer, 43
usage, 38, 41

view sheet, 36
visible whitespace, 124

XML, 37, 110

XpLR, 16
XSLT, 59

Curriculum Vitae

Mag. Kevin Kofler, Bakk.
born July 1st, 1983 in Oberpullendorf (Austria)
Italian citizen

Education and Degrees

2001: High school diploma: French *baccalauréat général* and Austrian *AHS-Matura* at the Lycée Français de Vienne, Vienna, Austria
2001: Enrollment in Mathematics and in Software and Information Engineering at the University of Vienna, Austria
2001: Foreign semester: Fall term 2001 at the Bowling Green State University, Bowling Green, Ohio, USA
2007: Mag.rer.nat. (equivalent to M.Sc.) in Mathematics, University of Vienna, Austria
2007: Enrollment as a Ph.D. student in Mathematics at the University of Vienna, Austria
2009: Bakk.techn. (equivalent to B.Sc.) in Software and Information Engineering (a branch of Computer Science), University of Vienna, Austria (joint study with the Technical University of Vienna, Austria)

Awards

1st prize at the *Rallye mathématique d'Alsace* in 2000 and 2001
5th *accessit* at the French *Concours général de Mathématiques* in 2001

Jobs

Mar 2005 – Jul 2005: Teaching assistant (Tutor).
Faculty of Mathematics, University of Vienna.
Oct 2005 – Feb 2006: Teaching assistant (Tutor).
Faculty of Mathematics, University of Vienna.
May 2006 – Sep 2006: Google Summer of Code 2006.
Dec 2007 – May 2009: FWF project P18704 "A global optimization environment".
Faculty of Mathematics, University of Vienna.
Jun 2009 – May 2011: FWF project P20631 "A modeling system for mathematics".
Faculty of Mathematics, University of Vienna.
May 2011 – Aug 2011: Google Summer of Code 2011.
Sep 2011 – Feb 2012: External lecturer (PS zu Numerische Mathematik 2).
Faculty of Mathematics, University of Vienna.

Sep 2011 – Dec 2012: FWF project P23554 "Structure driven methods for large-scale optimization". Faculty of Mathematics, University of Vienna.
Jan 2013 – Aug 2013: FWF project P22239 "Symmetries in global optimization". Faculty of Mathematics, University of Vienna.
Since Feb 2014: Researcher (mathematician) and software developer. DAGOPT Optimization Technologies GmbH.

Scientific Publications

1. K. Kofler, I. ul Haq, and E. Schikuta: A Parallel Branch and Bound Algorithm for Workflow QoS Optimization. Proceedings of ICPP 2009.
2. I. ul Haq, E. Schikuta, and K. Kofler: Using Blackboard System to Automate and Optimize Workflow Orchestrations. Proceedings of IEEE ICET 2009.
3. K. Kofler, I. ul Haq, and E. Schikuta: User-centric, Heuristic Optimization of Service Composition in Clouds. Proceedings of Euro-Par 2010 Vol. 1 (Springer LNCS 6271), 2010.
4. I. ul Haq, K. Kofler, and E. Schikuta: Dynamic Service Configurations for SLA Negotiation. Proceedings of the CoreGrid workshop at Euro-Par 2010, in Euro-Par 2010 Parallel Processing Workshops (Springer LNCS 6586), 2011.
5. K. Kofler and A. Neumaier: DynGenPar – A Dynamic Generalized Parser for Common Mathematical Language. Proceedings of CICM/DML 2012 (Springer LNAI 7362), 2012.
6. P. Schodl, A. Neumaier, K. Kofler, F. Domes, and H. Schichl: Towards a Self-reflective, Context-aware Semantic Representation of Mathematical Specifications, Chapter 3 in: Modeling Languages in Mathematical Optimization (J. Kallrath, ed.), Springer 2012.
7. H. Schichl, F. Domes, T. Montanher, and K. Kofler: Interval unions. BIT Numerical Mathematics Vol. 57, Springer 2017.

Talks at Scientific Conferences

Sep 2009: ICPP (International Conference on Parallel Processing) 2009 in Vienna, Austria
Sep 2010: Euro-Par 2010 in Ischia, Italy
Jul 2011: MKM (Mathematical Knowledge Management) 2011 at CICM (Conferences on Intelligent Computer Mathematics) 2011 in Bertinoro, Italy
Jul 2012: DML (Digital Mathematical Library) 2012 at CICM (Conferences on Intelligent Computer Mathematics) 2012 in Bremen, Germany
Aug 2012: ISMP (International Symposium on Mathematical Programming) 2012 in Berlin, Germany