DynGenPar – A Dynamic Generalized Parser for Common Mathematical Language[⋆]

Kevin Kofler and Arnold Neumaier

University of Vienna, Austria
Faculty of Mathematics
Nordbergstr. 15, 1090 Wien, Austria
kevin.kofler@chello.at, Arnold.Neumaier@univie.ac.at

Abstract. This paper introduces a dynamic generalized parser aimed primarily at common natural mathematical language. Our algorithm combines the efficiency of 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, we handle constraints on the token following a rule. This allows for grammatically correct English indefinite articles when working with word tokens. It can also represent typical operations for scannerless parsing such as maximal matches when working with character tokens. Our long-term goal is to computerize a large library of existing mathematical knowledge using the new parser, starting from natural language input as found in textbooks or in the papers collected by the digital mathematical library (DML) projects around the world. In this paper, we present the algorithmic ideas behind our approach, give a short overview of the implementation, and present some efficiency results. The new parser is available at http://www.tigen.org/kevin.kofler/fmathl/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

1 Introduction

The primary target application for our parser is the FMathL (Formal Mathematical Language) project [21]. FMathL is the working title for a modeling and documentation language for mathematics, suited to the habits of mathematicians, to be developed in a project at the University of Vienna. The project

[⋆] Support by the Austrian Science Fund (FWF) under contract numbers P20631 and P23554 is gratefully acknowledged.

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.

A major goal of the FMathL project is to build a computer-oriented library of mathematics, in a formalized form the computer can work with, from input as informal as possible. The input language we are working on is a subset of LaTeX, in which the textual parts are written in a controlled natural language, which we are aiming at growing incrementally to become closer and closer to true natural mathematical language as found in common use (common mathematical language). This is the primary target of our parser. Our current main working grammar is a grammar in extended BNF form [25] produced from German sentences in [20]. We also started work on MathNat [12]. Both those grammars are work in progress; we expect to extend them incrementally over time. Of course, mathematical documents also contain formulas: For those, we will be supporting both sTeX [18] and a subset of the traditional LaTeX formula syntax as input. Once we have an internal representation of the input, we can not only process it internally, but also generate output, not only in our input formats, but also in domain-specific languages such as, depending on the problem class, AMPL (in which we can already produce some output for simple optimization problems) or the languages used by proof checkers. Our current results are summarized in [24]. We believe our library of mathematical knowledge will nicely supplant DML's human-oriented one. Additionally, our long term goal is to allow computerizing informal knowledge, such as standard textbooks or papers as collected in the DML projects around the world, with minimal user interaction. We expect some interactive input to be needed when processing documents which have not been formalized yet, but the goal is to keep the required interaction as minimal as possible. Finally, we also believe that our software's eventual understanding of mathematics will help building a semantic search engine for DML (and other mathematical) contents.

Our application imposes several design requirements on our parser. It must:

- allow the efficient incremental addition of new rules to the grammar at any time (e.g., when a definition is encountered in a mathematical text – this possibility is typical for mathematical applications –, or to allow teaching the grammar new rules interactively), without recompiling the whole grammar;
- be able to parse more general grammars than just LR(1) or LALR(1) ones – natural language is usually not LR(1), and being able to parse so-called parallel multiple context-free grammars (PMCFGs) [28] is also a necessity for reusing the natural language processing facilities of the Grammatical Framework (GF) [22, 23];
- 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 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 our implementation) operation.

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

We solved this with an algorithm loosely modeled on Generalized LR [29, 30], but with an important difference: To decide when to shift a new token and which rule to reduce when, GLR uses complex LR states which are mostly opaque entities in practice, which have to be recomputed completely each time the grammar changes and which can grow very large for natural-language grammars. In contrast, we use the initial graph, which is easy and efficient to update incrementally as new rules are added to the grammar, along with runtime top-down information. The details will be presented in the next section.

This approach allows our 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 we 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.

We 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, we envision potential applications outside of mathematics, e.g., for domain-specific languages for special applications [19]. These are currently mainly handled by scannerless parsing using GLR [31] for context-free grammars (CFGs), but would benefit a lot from our incremental approach. The possibility to add rules at any time, even during parsing, also allows users to quickly add rules if the current grammar does not understand the input, which helps design grammars incrementally. Therefore, we believe that its distinctive features will also make DynGenPar extremely useful on its own, independently of the success of the very ambitious FMathL project.

The algorithm was first presented in our technical report [17].

2 State of the Art

No current parser generator combines all partially conflicting requirements mentioned in the introduction.

Ambiguous grammars are usually handled using **Generalized LR (GLR)** [29, 30], 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, our parser also supports PMCFGs, whereas GLR only works for context-free grammars (but it may be possible to extend GLR to PMCFGs using our techniques). Costagliola et al. [5] present a predictive parser **XpLR** 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** algorithm [13, 32] needs no tables, but is very inefficient and handles only CFGs. Hinze & Paterson [11] propose a more efficient tableless parser; their idea hasn't been followed up by others.

The most serious competitor to our parser is Angelov's PMCFG parser [3] as found in the code of the **Grammatical Framework** (GF) [22, 23], which has some support for natural language and predictive parsing. Alanko and Angelov are currently developing a C version in addition to the existing Haskell implementation. Compared to Angelov's parser, we offer similar features with a radically different approach, which we hope will prove better in the long run. In addition, our implementation already supports features such as incremental addition of PMCFG rules which are essential for our application, which are not implemented in Angelov's current code and which may or may not be easy to add to it. Our parser also supports importing the compiled PGF [4] files from GF, allowing to reuse the rest of the framework. When doing so, as evidenced in section 6, it reaches a comparable performance. Unlike the GF code, we can also enforce next token constraints, e.g., *an restaurant* is not allowed.

3 The DynGenPar Algorithm

In this section, we describe the basics of our algorithm. (Details about the implementation of some features will be presented in section 5.) We start by explaining the design considerations which led to our algorithm. Next, we define the fundamental concept of our algorithm: the initial graph. We then describe the algorithm's fundamental operations and give an example of how they work. Finally, we conclude the section by analyzing the algorithm as a whole.

3.1 Design Considerations

Our design was driven by multiple fundamental considerations. Our first observation was that we wanted to handle left recursion in a most natural way, which has driven us to a bottom-up approach such as LR. In addition, the need for supporting general context-free grammars (and even extensions such as PMCFGs) requires a generalized approach such as GLR. However, our 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, we had to restrict ourselves to information which can be produced dynamically.

3.2 The Initial Graph

To fulfill the above requirements, we designed a data structure we call the *initial graph*. Consider a context-free grammar $G = (N, T, P, S)$, where N is the set of nonterminals, T the set of terminals (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 $\varGamma = N \cup T$ of G, defined by the following criteria:

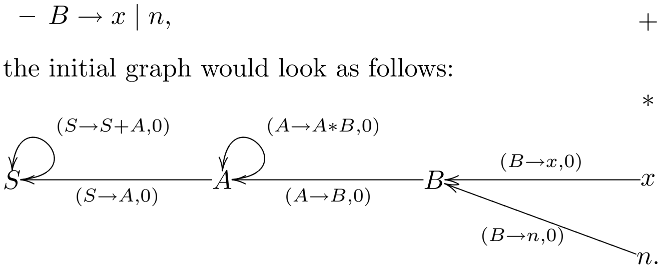
- The tokens T are sources of the graph, i.e., nodes with no incoming edges.
- The graph has an edge from the symbol $s \in \varGamma$ to the nonterminal $n \in N$ if and only if the set of productions P contains a rule p: $n \rightarrow n_1\ n_2\ \ldots\ n_k\ s\ \ldots$ with $n_i \in N_0\ \forall i$, where $N_0 \subseteq N$ is the set of all those nonterminals from which 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, we 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, for the basic expression grammar $G = (N, T, P, S)$ with $N = \{S, A, B\}$, $T = \{+, *, x, n\}$ (where n stands for a constant number, and would in practice have a value, e.g., of `double` type, attached), and P contains the following rules:

- $S \rightarrow S + A \mid A$,
- $A \rightarrow A * B \mid B$,
- $B \rightarrow x \mid n$,

the initial graph would look as follows:



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

To add a new rule, in the simplest case, we just add an edge from the first symbol on the right hand side to the nonterminal on the left hand side. If we have epsilon rules, we may also have to add edges from symbols further on the right to the nonterminal on the left, add the nonterminal on the left to the set of nullable nonterminals if all symbols in the rule can be epsilon, and then go through the rules in the labels of the edges starting at the now nullable nonterminal and recursively repeat the same update process there. But this expensive recursion is rarely needed in practice.

We additionally define *neighborhoods* on the initial graph: 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., $\mathcal{N}(+, S) = \emptyset$ (because there is no path from + to S), $\mathcal{N}(x, S) = \{B \to x\}$, and $\mathcal{N}(A, S) = \{S \to A, A \to A * B\}$. Note that in the last example, we also have to consider the loop, i.e., the left recursion.

3.3 Operations

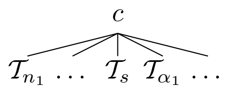
Given these concepts, we define four elementary operations:

- $\mathit{match}_\varepsilon(n), n \in N_0$: 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 we 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.
- $\mathit{reduce}(s, z), s \in \Gamma, z \in N$: 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, our algorithm already reduces after the first symbol. This implies that our *reduce* operation must complete the match. It does this using the next operation:
- $\mathit{match}(s), s \in \Gamma = N \cup T$: 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: $m_\varepsilon := \mathit{match}_\varepsilon(s)$. Now we only need to look for nonempty matches.
  2. Start by shifting a token: $t := \mathit{shift}$.
  3. If $s \in T$, we just need to compare s with t. If they match, we return a leaf as our parse tree, otherwise we return no matches at all.
  4. Otherwise (i.e., if $s \in N$), we return $m_\varepsilon \cup \mathit{reduce}(t, s)$.

Given the above operations, the algorithm for $\mathit{reduce}(s, z)$ 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 $n_i \in N_0$: $\mathcal{T}_{n_i} := \mathit{match}_\varepsilon(n_i)$.
3. s was already recognized, let $\mathcal{T}_s$ be its syntax tree.
4. For each $\alpha_j \in \Gamma = N \cup T$: $\mathcal{T}_{\alpha_j} := \mathit{match}(\alpha_j)$. 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. We also need to consider *reduce*(z, z) to support left recursion; this is the only place in our algorithm where we need to accomodate specifically for left recursion.

If we have a conflict between multiple possible *reduce* operations, we need to consider all the possibilities. We then unify our matched parse trees into 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 5.

Our algorithm is initialized by calling *match*(S) on the start symbol S of the grammar. The rest conceptually happens recursively. The exact sequence of events in our practical implementation, which allows for predictive parsing, is described in section 5.

3.4 Example

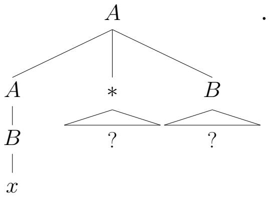
As an example, we show how our algorithm works on the basic expression grammar from section 3.2. The example was chosen to be didactically useful rather than realistic: In practice, we work with grammars significantly more complex than this example. It shall be noted that in this example, the set N_0 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.

We consider the input `x * x`, a valid sentence in the example grammar. We will denote the cursor position by a dot, so the initial input is `.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 x, the input is now `x.* x`, and the next step is *reduce*(x, S), after which the parsing is complete.

It is now the *reduce* task's job to get from x 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 $\mathcal{N}(x, S) = \{B \to x\}$. In this case, there is only one rule in the neighborhood, so we reduce that rule. The right hand side of the rule is just x, so the rule is already completely matched, there are no symbols left to match. We remember the parse tree $B - x$ and proceed recursively with *reduce*(B, S).

Now we have $\mathcal{N}(B, S) = \{A \to B\}$. Again, there is only a single rule that matches and it is fully matched, so we reduce it, remember the parse tree $A - B - x$ and continue the recursion with *reduce*(A, S).

This time, the neighborhood $\mathcal{N}(A, S) = \{S \to A, A \to A * B\}$ contains more than one matching rule, we have a *reduce-reduce conflict*. Therefore, we have to consider both possibilities, as in GLR. If we attempt to reduce $S \to A$, the parsing terminates here (or we try reducing the left-recursive $S \to S + A$ rule and hit an error on the unmatched + token), but the input is not consumed yet, thus we hit an error. Therefore, we retain only the option of reducing the left-recursive $A \to A * B$ rule. This time, there are two remaining tokens: $*$ and B, thus we proceed with *match*$(*)$ and *match*(B). Our parse tree matched so far is



.

The *match*$(*)$ operation is trivial: $*$ is a token, so we only need to *shift* the next token and compare it to $*$. The input is now `x *.x`, and the *match*(B) step proceeds by a last *shift* consuming the last token, and a *reduce*(x, B) which is also trivial because $\mathcal{N}(x, B) = \{B \to x\}$.

Thus the reduction of the left-recursive rule $A \to A * B$ is complete and we recursively proceed with another *reduce*(A, S). This time, attempting to reduce the left-recursive rule again yields an error (there is no input left to match the $*$ against) and we reduce $S \to A$, giving the final parse tree.

A similar, but slightly longer example can be found in the slides [16].

3.5 Analysis

The above algorithm combines enough bottom-up techniques to avoid trouble with left recursion with sufficient top-down operation to avoid the need for tables while keeping efficiency. 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 [13, 32].

One disadvantage of our algorithm is that it produces more conflicts than LR or GLR, for two reasons: Not only are we not able to make use of any lookahead tokens, unlike common LR implementations, which are LR(1) rather than LR(0), but we also already have 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 we need no states nor tables, only the initial graph which can be dynamically updated, which allows dynamic rule changes. In addition, conflicts are not fatal because our algorithm is exhaustive (like GLR), and we designed our implementation to keep its efficiency even in the presence of conflicts; in particular, we never execute the same *match* step at the same text position more than once.

4 Implementation

In this section, we first give an overview of the technologies and the license chosen for our implementation. Then, we describe how it integrates into our main application software.

The DynGenPar implementation is available for free download at [15].

4.1 Technologies and Licensing

Our implementation is written in C++ using the Qt [1] toolkit. It is licensed under the GNU General Public License [9, 10], version 2 or later.

We also implemented Java bindings using the Qt Jambi [2] binding generator to allow its usage in Java programs.

4.2 Integration into FMathL Concise

The Java bindings are used in the Concise [27] GUI of the FMathL project [21]. Concise is a framework for viewing and manipulating, both graphically and programmatically, semantic graphs. It is the main piece of software in our application. Concise offers editable views of semantic content in the form of graphs, records or text, can execute programs operating on that content, and supports importing information from and exporting it to various types of files. It is written in Java.

The Concise GUI fully integrates our DynGenPar parser into our application's workflow. The FMathL type system [26] is represented in the form of text files called *type sheets*. Those type sheets can not only represent a pure type hierarchy, but also carry grammatical annotations, which allow the type system to double as a grammar. Concise can import such type sheets at runtime and automatically convert them to grammar rules suitable for DynGenPar. It can then parse documents using the converted grammar.

This feature allows to read user-written rules into the parser at runtime, rather than hardcoding them as C++ code or compiling them with the Grammatical Framework (GF) to its binary PGF format. Concise type sheets represent a user-friendly mechanism for specifying rules which can be easily converted to our internal representation. The feature is thus an ideal showcase for the dynamic properties of our algorithm.

5 Implementation Considerations

This section documents some tweaks we made to the above basic algorithm to improve efficiency and provide additional desired features. We 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, we briefly introduce our flexible approach to lexing. Finally, we give a short overview on interoperability with the Grammatical Framework (GF).

5.1 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, we 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. We also remember complete matches (where the entire starting symbol S was matched) and return them if the end of input was reached, otherwise we discard 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.

5.2 Efficient Exhaustive Parsing

In order to achieve efficiency in the presence of ambiguities, the parse stacks are organized in a DAG structure similar to the GLR algorithm's graph-structured stacks. [29, 30] In particular, a *match* operation can have multiple parents, and our 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.

5.3 Rule Labels

Our 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.

5.4 Custom Parse Actions

The algorithm as described in section 3 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, we 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.

5.5 Token Sources

The implementation can be interfaced with several different types of token sources, e.g., a Flex [7] 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.

5.6 Natural Language

Natural language, even the subset used for mathematics, poses some additional challenges to our 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 PMCFGs [28].

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 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 [28], 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. Our approach to handling PMCFGs is based on this idea. However, we do not use the naive and inefficient approach of first parsing the context-free approximation and then filtering the result, but we enforce 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.

5.7 Next Token Constraints

Our 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. We 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. We 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, we check whether the shifted token fulfills the pending constraints and reject the stacks whose constraints aren't satisfied.

5.8 Interoperability with GF

Our implementation can import PGF [4] grammar files produced by the Grammatical Framework (GF) [22, 23], a binary format based on PMCFGs. This is handled by converting them to PMCFG standard form, with a few extensions supported by our parser:

- 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 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.

We also implemented a GF-compatible lexer.

6 Results

Our main achievement is the dynamism of the algorithm. However, the algorithm must also be fast enough for practical use (in particular, faster than recompiling the grammar with a static parser). Therefore, we compared the speed of our implementation to the well-established GNU Bison [8] parser on a hierarchical (two-layer) grammar we devised for the Naproche [14, 6] language: There are 2 context-free grammars, one for text and one for formulas, each using a lexer based on Flex [7]. In one version of our Naproche parser, the 2 context-free grammars are processed with Bison (using its support for GLR parsing), in the other with DynGenPar. We 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 paradoxon in Naproche). It shall be noted that for Bison, the grammar conversion is done before the compilation, so the compilation time also has to be considered when working with dynamically changing grammars, whereas DynGenPar can convert grammars at runtime. Our test machine is a notebook with a Core 2 Duo T7700 (2 × 2.40 GHz) and 4 GiB RAM running Fedora 16 x86_64. For each measurement, we averaged the execution times of 100 tests (except for the compilation time of DynGenPar, where we used only 3 tests due to time constraints) and took the median of 3 attempts. Our results are summarized in table 1.

We conclude that, while Bison is 4 to 5 times faster at pure parsing, DynGenPar is much faster at adapting to changed grammars. 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 11 times longer than DynGenPar to process our fairly small two-layered grammar, and we expect the

Table 1. Benchmarking results on the Naproche grammar

	compilation time	grammar conversion time	parsing time (Burali-Forti)
Bison	2722 ms	83 ms*	2.79 ms
DynGenPar	20890 ms	7.54 ms	12.2 ms**

* . . . at compile time, thus requires recompilation

** . . . total execution time of 19.7 ms minus grammar conversion time discrepancy to only grow as the grammar sizes increase. (Moreover, DynGenPar can handle dynamic rule addition, so in many cases even the 7.54 ms for grammar conversion can be saved.) In the worst case, where we have a new input for an existing grammar and do not have the initial graph in memory, DynGenPar (19.7 ms) is still only 7 times slower than Bison (2.79 ms), even though the latter was optimized specifically for this usecase and DynGenPar was not.

We also benchmarked our support for PGF [4] grammar files produced by the Grammatical Framework (GF) [22, 23] against two PGF runtimes provided by the GF project (we used a snapshot of the repository from February 24): the production runtime written in Haskell and the new experimental runtime written in C. As an example grammar, we used GF's *Phrasebook* example, which is the one explicitly documented as being supported by the current version of GF's C runtime, with the sample sentence *See you in the best Italian restaurant tomorrow!*, a valid sentence in the *Phrasebook* grammar. (We 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.) We measured the time to produce the syntax tree only, without outputting it. The tests were run on the same Core 2 Duo notebook as above. Again, for each measurement, we averaged the execution times of 100 tests and took the median of 3 attempts. Our results are summarized in table 2.

We conclude that DynGenPar is competitive in speed with both GF runtimes on practical application grammars. In addition, both GF runtimes happily accept the incorrect input *Where is an restaurant?* (should be *a restaurant*), whereas DynGenPar can enforce the next token constraint.

7 Conclusion

We introduced DynGenPar, a dynamic generalized parser for common mathematical language, presented its requirements, the basics of the algorithm and the tweaks required for an efficient implementation, and compared our approach with the state of the art, evidencing the huge advancements we made.

However, there is still room for even more features, which will bring us further towards our goal of computerizing a library of mathematical knowledge:

– context-sensitive constraints on rules: Currently, we support only some very specific types of context-sensitive constraints, i.e., PMCFG and next token constraints. We would like to support more general types of constraints, and our algorithm is designed to accomodate that. The main research objective here will be to figure out the class of constraints that is actually needed.

Table 2. Benchmarking results on the GF *Phrasebook* grammar

	parsing time (*See you in the best Italian restaurant tomorrow!*)
GF Haskell runtime	37 ms
GF C runtime	84 ms
DynGenPar	81 ms

- stateful parse actions: Custom parse actions currently have access only to minimal state information. We plan to make more state available to parse actions to provide as much flexibility as we find will be needed.
- 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 [27] GUI. We are considering implementing a mechanism for specifying rules at runtime within DynGenPar. However, this has low priority for us because we use the mechanism provided by Concise in our application.
- scalability to larger PMCFGs: Currently, we have several optimizations which improve scalability, but 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, we 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. We would like to design intelligent ways to actually correct the errors, or suggest corrections to the user. This is a long-term research goal.

Our 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.

References

1. Qt – Cross-platform application and UI framework, http://qt.nokia.com, website
2. Qt Jambi – The Qt library for Java, http://qt-jambi.org, website
3. Angelov, K.: 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)
4. Angelov, K., Bringert, B., Ranta, A.: PGF: A Portable Run-Time Format for Type-Theoretical Grammars. Journal of Logic, Language and Information 19(2), 201–228 (2010)
5. Costagliola, G., Deufemia, V., Polese, G.: 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
6. Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., Veldman, J.: The Naproche Project – Controlled Natural Language Proof Checking of Mathematical Texts. In: Fuchs, N.E. (ed.) CNL. LNCS, vol. 5972, pp. 170–186. Springer (2009)
7. Flex Project: flex: The Fast Lexical Analyzer, http://flex.sourceforge.net, website
8. Free Software Foundation: Bison – GNU parser generator, http://www.gnu.org/software/bison, website
9. Free Software Foundation: GNU General Public License (GPL) v2.0 (June 1991), http://www.gnu.org/licenses/old-licenses/gpl-2.0, website
10. Free Software Foundation: GNU General Public License (GPL) v3.0 (June 2007), http://www.gnu.org/licenses/gpl-3.0, website

11. Hinze, R., Paterson, R.: Derivation of a typed functional LR parser (2003)
12. Humayoun, M.: Developing the System MathNat for Automatic Formalization of Mathematical texts. Ph.D. thesis, University of Grenoble (2012), http://www.lama.univ-savoie.fr/~humayoun/phd/mathnat.html
13. Kasami, T.: An efficient recognition and syntax analysis algorithm for context-free languages. Tech. Rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA (1965)
14. Koepke, P., Schröder, B., Buechel, G., et al.: Naproche – Natural language proof checking, http://www.naproche.net, website
15. Kofler, K.: DynGenPar – Dynamic Generalized Parser, http://www.tigen.org/kevin.kofler/fmathl/dyngenpar, website
16. Kofler, K., Neumaier, A.: The DynGenPar Algorithm on an Example, http://www.tigen.org/kevin.kofler/fmathl/dyngenpar-example.pdf, slides
17. Kofler, K., Neumaier, A.: A Dynamic Generalized Parser for Common Mathematical Language. Work in Progress paper at CICM/MKM (2011), http://www.tigen.org/kevin.kofler/fmathl/dyngenpar-wip.pdf
18. Kohlhase, M.: Using LaTeX as a Semantic Markup Format. Mathematics in Computer Science 2.2, 279–304 (2008)
19. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. ACM Computing Surveys (CSUR) 37(4), 316–344 (2005)
20. Neumaier, A.: Analysis und lineare Algebra, http://www.mat.univie.ac.at/~neum/FMathL/ALA.pdf, unpublished lecture notes
21. Neumaier, A.: FMathL – Formal Mathematical Language, http://www.mat.univie.ac.at/~neum/fmathl.html, website
22. Ranta, A.: Grammatical Framework: A Type-Theoretical Grammar Formalism. Journal of Functional Programming 14(2), 145–189 (2004)
23. Ranta, A., Angelov, K., Hallgren, T., et al.: GF – Grammatical Framework, http://www.grammaticalframework.org, website
24. Schodl, P.: Foundations for a Self-Reflective, Context-Aware Semantic Representation of Mathematical Specifications. Ph.D. thesis, University of Vienna (2011), http://www.mat.univie.ac.at/~schodl/pdfs/diss_online.pdf
25. Schodl, P., Neumaier, A.: An experimental grammar for German mathematical text. Tech. rep., University of Vienna (2009), http://www.mat.univie.ac.at/~neum/FMathL/ALA-grammar.pdf
26. Schodl, P., Neumaier, A.: The FMathL type system. Tech. rep., University of Vienna (2011), http://www.mat.univie.ac.at/~neum/FMathL/types.pdf
27. Schodl, P., Neumaier, A., Kofler, K., Domes, F., Schichl, H.: Towards a Self-reflective, Context-aware Semantic Representation of Mathematical Specifications. In: Kallrath, J. (ed.) Algebraic Modeling Systems – Modeling and Solving Real World Optimization Problems, chap. 2. Springer (2012)
28. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science 88(2), 191–229 (1991)
29. Tomita, M.: An Efficient Augmented Context-Free Parsing Algorithm. Computational Linguistics 13(1–2), 31–46 (1987)
30. Tomita, M., Ng, S.: The Generalized LR parsing algorithm. In: Tomita, M. (ed.) Generalized LR parsing, pp. 1–16. Kluwer (1991)
31. Visser, E.: Scannerless generalized-LR parsing. Tech. Rep. P9707, Programming Research Group, University of Amsterdam (1997)
32. Younger, D.: Recognition and parsing of context-free languages in time n^3. Information and control 10(2), 189–208 (1967)