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 DynGenPar, a dynamic generalized parser under development, aimed primarily at natural mathematical language. Our algorithm aims at uniting the efficiency of LR or 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, which allows for grammatically correct English indefinite articles when working with word tokens and which can 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. This paper presents the algorithmic ideas behind our approach.

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

1 Introduction

The primary target application for our algorithm is the FMathL (Formal Mathematical Language) project [7]. 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 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,

* Support by the Austrian Science Fund (FWF) under contract number P20631 is gratefully acknowledged.

editing, and checking mathematics (but also computer science and theoretical physics) in both informal, partially formalized, and completely formalized mathematical form.

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

- 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) or LALR(1) ones – natural language is usually not LR(1), and being able to parse so-called parallel multiple context-free grammars (PMCFGs) [10] is also a necessity for reusing the natural language processing facilities of the Grammatical Framework (GF) [8, 9];
- 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 approach this with an algorithm loosely modeled on Generalized LR [11, 12], but with an important difference: to decide when to shift a new token and which rule to reduce when, instead of 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, we use a graph, 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 have made significant progress towards the implementation of a prototype parser along the lines described. We expect a fully developed version of

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 [6]. These are currently mainly handled by scannerless parsing using GLR [13] for context-free grammars (CFGs), but would benefit a lot from our incremental approach.

2 The DynGenPar Algorithm

In this section, we describe the basics of our algorithm. (Details about the requirements for the implementation of some features will be presented in the next section.) 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. Finally, we conclude the section by analyzing the algorithm as a whole. A step-by-step example for the algorithm can be found in [5].

2.1 Design Considerations

Our design is 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.

2.2 The Initial Graph

To fulfill the above requirements, we designed a data structure we call the *initial graph*. Given a context-free grammar $G = (N, T, P, S)$, the corresponding initial graph is a directed labeled multigraph on the set of symbols $\Gamma = N \cup T$, where N is the set of nonterminals and T the set of terminals (tokens) 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 P (the set of productions of G) contains a rule p: $n \to n_1\ n_2\ \ldots\ n_k\ s\ \ldots$ 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.

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.

2.3 Operations

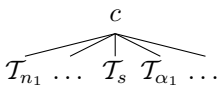
Given these concepts, we define four elementary operations:

- $\mathit{match}_\varepsilon(n), n \in N_0$: 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 the next section.

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

2.4 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.

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 are designing 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.

3 Implementation Considerations

This section documents some tweaks we are making 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 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).

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, the backtracking must be replaced 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 have to remember complete matches (where the entire starting symbol S was matched) and return them if the end of input was reached, or otherwise 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 can be 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.

3.2 Efficient Exhaustive Parsing

In order to achieve efficiency in the presence of ambiguities, the parse stacks must be organized in a DAG structure similar to the GLR algorithm's graph-structured stacks. [11, 12] In particular, a *match* operation can have multiple parents, and the algorithm must produce 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 must be represented as packed forests. Top-down sharing must be 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 must be created by explicit unification steps. This sharing will also ensure 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, should be 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.

3.3 Natural Language

Natural language, even the subset used for mathematics, poses some additional challenges to our approach. 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 [10].

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 [10], 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 will not use the naive and inefficient approach of first parsing the context-free approximation and then filtering the result, but we will enforce the constraints directly during parsing, leading to maximum efficiency and avoiding the need for subsequent filtering. This can be 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 can be transformed to PMCFG syntax trees by a subsequent unification algorithm, but the parsing algorithm will ensure that only CFG parse trees which can be successfully unified are produced, saving time both during parsing and during unification.

3.4 Next Token Constraints

Our approach 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 idea

can be used to implement scannerless parsing features, in particular, maximally-matched character sequences, but also to restrict the words following e.g., "a" or "an" in word-oriented grammars. This can be achieved 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, one needs to check whether the shifted token fulfills the pending constraints and reject the stacks whose constraints aren't satisfied.

3.5 Token Sources

Our application will require to interface the implementation with several different types of token sources, e.g., a Flex 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.

We also plan to allow the token source to return a whole parse tree instead of the usual leaf node. That parse tree will be attached in place of the leaf. This idea makes hierarchical parsing possible: Using this approach, the token source will be able to run another instance of the parser or a different parser (e.g., a formula parser) on a token and return the resulting parse tree.

3.6 Interoperability with GF

We plan to be able to import PGF [2] grammar files produced by the Grammatical Framework (GF) [8, 9], a binary format based on PMCFGs. This will be achieved by converting them to PMCFG standard form, with a few required extensions trivially supportable by our algorithm:

- 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.

A GF-compatible lexer will also be required.

4 Related Work

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

Ambiguous grammars are usually handled using **Generalized LR (GLR)** [11, 12], 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, GLR as formulated only works for context-free grammars, our approach also supports PMCFGs (but it may be possible to extend GLR to PMCFGs using our techniques). Costagliola et al. [3] 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 needs no tables, but handles only CFGs and is very inefficient. Hinze & Paterson [4] propose a more efficient tableless parser; their idea hasn't been followed up by others.

The most serious competitor to our method is Krasimir Angelov's PMCFG parser [1] as found in the code of the **Grammatical Framework** (GF) [8, 9], which has some support for natural language and predictive parsing. Compared to Angelov's parser, our approach promises similar features while being radically different, and we hope it will prove better in the long run. In addition, our algorithm is designed to easily support 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. Finally, the planned support for importing the compiled PGF [2] files from GF will allow our parser to reuse the rest of the framework.

5 Conclusion

We introduced DynGenPar, a dynamic generalized parser under development 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 to be made.

But DynGenPar is still in an early stage of development and planning, and we are envisioning even more features, which will bring us further towards our goal of computerizing a library of mathematical knowledge:

- context-sensitive constraints on rules: In this paper, we only discussed some very specific types of context-sensitive constraints, i.e., PMCFG constraints 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.
- custom parse actions: The algorithm as written in this paper generates only a plain parse tree and cannot execute any other actions according to the grammar rules. In order to efficiently support things such as mathematical definitions, we would like 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. This should not require major changes to the algorithm, but the implementation will have to accomodate it.

– a runtime parser for rules: We are working on ways to read rules into the parser from a user-writable format at runtime, rather than from precompiled formats such as machine code or PGF grammars. This will require a user-friendly mechanism for specifying the rules and an easy way to convert them to our internal representation.
– error correction: The methods in this paper can be used only for 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. Angelov, K.: Incremental parsing with parallel multiple context-free grammars. Proceedings of the 12$^{\text{th}}$ Conference of the European Chapter of the Association for Computational Linguistics pp. 69–76 (2009)
2. 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)
3. 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 (VLC '05)
4. Hinze, R., Paterson, R.: Derivation of a typed functional LR parser (2003)
5. Kofler, K., Neumaier, A.: The DynGenPar algorithm on an example, http://www.tigen.org/kevin.kofler/fmathl/dyngenpar-example.pdf, slides
6. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. ACM Computing Surveys (CSUR) 37(4), 316–344 (2005)
7. Neumaier, A.: FMathL – formal mathematical language, http://www.mat.univie.ac.at/~neum/FMathL.html, website
8. Ranta, A.: Grammatical Framework: A type-theoretical grammar formalism. Journal of Functional Programming 14(2), 145–189 (2004)
9. Ranta, A., Angelov, K., Hallgren, T., et al.: GF – grammatical framework, http://www.grammaticalframework.org/, website
10. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science 88(2), 191–229 (1991)
11. Tomita, M.: An efficient augmented context-free parsing algorithm. Computational Linguistics 13(1–2), 31–46 (1987)
12. Tomita, M., Ng, S.: The Generalized LR parsing algorithm, vol. Generalized LR parsing, pp. 1–16. Kluwer (1991)
13. Visser, E.: Scannerless generalized-LR parsing. Tech. Rep. P9707, Programming Research Group, University of Amsterdam (1997)