A Formal Description of an Algorithm Suitable for Parsing the Language of Mathematics*

Luka Vrečar[1](✉), Joe Wells[1], and Fairouz Kamareddine[1]

Heriot-Watt University, Edinburgh, UK
{lv21, F.D.Kamareddine}@hw.ac.uk
https://www.macs.hw.ac.uk/~jbw/

Abstract. Kofler's DynGenPar (DGP) generalized dynamic parsing algorithm [6,7] is aimed at parsing the language of mathematics. Notably, DGP uses a notion of *initial graph* instead of parsing tables in the style of GLR and LR parsers. We distinguish the two previously existing definitions of DGP: (1) The high-level "Abstract DGP" (ADGP) is a non-deterministic algorithm which reaches any particular parse tree via a sequence of non-deterministic choices. It is not obvious just from reading ADGP how to implement it well. (2) Kofler's C++ implementation (KDGP) implements its own concurrency engine for finding all possible parse trees simultaneously while synchronizing on each input token. It is challenging to comprehend how ADGP corresponds to KDGP, and it would be hard to formally prove properties of either ADGP or KDGP. We present a new mathematical definition of Core DGP (CDGP), the context-free grammar (CFG) core of DGP, that is both declarative and deterministic. We formalize the concurrency with a notion of *continuation*, and define a way to match continuations to parse trees in proving that our algorithm is sound and complete, i.e., it terminates and returns all valid parse trees excluding trees with *superfluous recursion*. We also make available a Python implementation which corresponds quite directly to the mathematical definition of CDGP.

* Supported by EPSRC [EP/T517999/1 2770826].
A longer version of this paper is freely available at https://github.com/LVrecar/dyngenpar.
This paper uses the sTeX package [8] to typeset its mathematics. Symbols in mathematical formulas contain hyperlinks to the relevant definitions in the paper (or externally). Clicking on these symbols in most PDF readers jumps to the relevant section of the paper, or opens an external link in your default browser if the concept is not defined in the paper. We usually have a good experience using the Evince PDF reader, where hovering over symbols representing concepts defined within the paper produces a popup which shows the relevant section of the paper. Note that clicking on external hyperlinks in Evince might fail due to a security policy, and that failure might not be reported, depending on your OS. Words in *italic* within a definition denote a defining occurrence of the corresponding term.

V. de Paiva and P. Koepke (Eds.): CICM 2025, LNAI 16136, pp. 171–188, 2026.
https://doi.org/10.1007/978-3-032-07021-0_10

1 Introduction

The language of mathematics (LM) is a combination of natural language and symbolic formulas, making it difficult to parse [5]. Parsing LM is desirable for many applications like automated reasoning, theorem proving, proof checking, etc. Grammars for parsing parts of LM can contain left-recursive rules, and rules which produce the empty string (and thus do not always consume additional symbols from the input when expanded). As seen in Example 2.9, these rules arise naturally when trying to design a grammar to parse some portion of LM. While algorithms for transforming grammars to remove left recursion exist, they change the structure and size of the grammar. We wish to avoid that as "cyclic grammars can be the most compact way to describe a language" [11].

Another difficulty with parsing LM is that it "operates" on its own grammar. While parsing a mathematical document, one encounters definitions, which introduce new terminology and notation. The underlying grammar for parsing LM must therefore change with each definition. Traditional context-free grammar (CFG) parsers like GLR [12] rely on pre-computed, grammar-dependent *parsing tables*. The tables are opaque and difficult to read, and must be (partially) re-computed each time the underlying grammar changes. Updating them during parsing is seemingly too computationally expensive for large grammars. It is thus desirable to have a parser which does not rely on parsing tables, but still supports left-recursive rules, and rules which produce the empty string. The parser must also support ambiguous grammars by parsing exhaustively (i.e., finding all valid parse trees for a given input), since LM is full of ambiguities. Our previous work [13] also identified the need for such a parser.

As part of his PhD thesis [6] and the wider FMathL project at the University of Vienna, Kofler developed a dynamic generalized parsing algorithm called DynGenPar (DGP). Work on the parser was also presented at CICM 2011 as a work in progress paper, and at the DML track of CICM 2012 [7]. Kofler designed a data structure called the *initial graph* to replace parsing tables. Kofler states that it is feasible to update the initial graph during parsing to add new grammar rules, obtained e.g., from mathematical definitions encountered in the input. It seems feasible to us, but to our knowledge no one has tried this yet. The algorithm can parse ambiguous CFGs exhaustively like e.g., GLR [12] would, and supports left-recursive rules and rules which produce the empty string. Importantly, DGP accepts any CFG without transformations like e.g., CYK [14] which only accepts grammars in Chomsky Normal Form. This also removes the need for transforming parse trees afterwards into parse trees for the initial grammar. These properties make DGP a suitable candidate for parsing LM.

There are two previously existing definitions of DGP: (1) the high-level "Abstract DGP" (ADGP), a non-deterministic algorithm which produces any particular parse tree via a sequence of non-deterministic choices (e.g., between different rules to use). From reading the definition of ADGP it is not clear how to implement it well. (2) Kofler's C++ implementation (KDGP), which uses a concurrency mechanism to find all possible parse trees simultaneously, synchronizing on each input token. KDGP also implements additional features, like support for Parallel Multiple Context-Free Grammars (PMCFGs) [2] and incremental parsing.

The correspondence between ADGP and KDGP is difficult to establish, making formally proving properties of either challenging. To that end, we extracted what we call Core DGP (CDGP), the context-free grammar (CFG) core of DGP, from ADGP and KDGP. We obtained it by removing all the additional features offered by KDGP from the implementation: support for PMCFGs, predictive parsing, parse actions, and next token constraints. To simplify the algorithm further, we also removed the unification of trees and stacks (we call stacks *continuations* in this paper). KDGP made use of these unifications for efficiency, but we decided to remove them to first formalise the simplest version of the algorithm possible. Formalising unification and the other features we removed to obtain CDGP is left to future work.

Research contributions: In this paper we present a new mathematical definition of CDGP that is both declarative and deterministic (Section 2). We also provide proofs of soundness and completeness, i.e., the algorithm terminates and returns all valid parse trees which are not *superfluously recursive* (Section 4). We also make available a Python implementation[1], which corresponds closely to the mathematical definition of CDGP. Prior to our work, the only available implementation was KDGP.

While we have not done any research into updating grammars during parsing, any future work in this direction would require a formalisation effort similar to the work we present in this paper. The description of the algorithm we have created can serve as a stepping stone for future research about it.

1.1 Other approaches to parsing LM

Here, we will mention the shortcomings of GLR when it comes to parsing with rules that produce the empty string, and cycles in the grammar (i.e., a rule or set of rules from which a nonterminal can derive itself). Tomita's original algorithm [12] could not handle cycles in the grammar. Farshi [10] and Rekkers [11] modified GLR to accommodate them, by allowing the graph-structured stacks and parse forests to contain cycles. Their approach can produce infinitely many parse trees for a single input. In this paper, we call such trees *superfluously recursive* (Definition 2.7). We wish to explicitly avoid building such trees, since a finite number of trees suffices for practical parsing applications.

For our work on semantic annotations in LaTeX [13] we tried several Python libraries for parsing. `parglare` [3], a GLR-based parsing library, seemed the most promising after PyParsing [9] and GLRParser [4] proved inadequate. However, it could not handle left-recursive rules, which led us to reimplementing CDGP in Python. Initially, this was a direct translation of the C++ code, but since it was difficult to reason about, we provided a precise definition in this paper. We also make available an experimental implementation[1]which more closely follows the definition of the algorithm found in Section 2 of this paper.

[1] It is available to download at https://github.com/LVrecar/dyngenpar.

The other approach we will discuss is the use of machine learning (ML), such as large language models (LLMs) for parsing LM. They are unsuitable for parsing LM, because they are largely opaque, probability-based processes. To parse LM reliably, we need to know exactly how a result was obtained, and we want results to be reproducible. Parts of LLMs (such as the weights or training data) are not available to users, and the same prompt can produce a different result each time. We thus want a deterministic approach where all components are available to users for inspection and modification.

2 Algorithm description

In this section we give a formal definition of a new deterministic algorithm based on CDGP. Some of the terminology that we do not define can be found in parsing literature [1]. We first give a framework of definitions (Definitions 2.1 to 2.7) we will use to define the new algorithm. Building blocks of the algorithm follow, such as the *initial graph* (Definition 2.12), *neighborhood* (Definition 2.16), and *continuations* (Definition 2.18). We end with the definitions of the functions which make up the algorithm itself (Procedures 2.20 to 2.24).

If a variable x ranges over a set, x decorated with any combination of subscripts or superscripts also ranges over the same set. For example, if x ranges over $\mathbb{N}$, then so do x', x_1, $x^{\text{foo}}_{\text{bar}}$, etc. If a variable x ranges over a set A, we use $\hat{x}$ to range over subsets of A (including the entirety of A). We use i, j, and k, to range over $\mathbb{N}$. Given $i, j \in \mathbb{N}$, we denote $\{x \in \mathbb{N} \mid i \leq x \leq j\}$ by $i..j$. A k-ary relation is a set of k-tuples. A function is a set of ordered pairs.

Definition 2.1 (Sequences)**.** A *finite sequence* (from this point on, just) is a function whose domain is $0..k$ for some k, and whose range is a set of objects. We can write the sequence $s = \{(0,x_1), (1,x_2), \ldots, (k,x_{k-1})\}$ as $[x_1, x_2, \ldots, x_{k-1}]$. Let $|s|$ denote the cardinality of s. Conceptually, this denotes the length of the sequence. We denote the empty sequence with $[]$.

For a sequence s, we can obtain the i-th element of it with $s(i-1)$. Additionally, given $i..j$, we can define a *subsequence* of s as $s\langle i..j\rangle = \{(k-i,s(k)) \mid k \in i..j\}$. We use $s\langle i..\rangle$ to mean $s\langle i..(|s|-1)\rangle$.

Given two sequences $s = [x_1, x_2, \ldots, x_j]$ and $t = [y_1, y_2, \ldots, y_k]$, we define the *concatenation* of s and t, written $s \cdot t$, as $s \cup \{(i + |s|,t(i)) \mid i \in \mathbf{dom}(t)\}$. This gives us a new sequence $[x_1, x_2, \ldots, x_j, y_1, y_2, \ldots, y_k]$.

Given a set A, we denote the *set of all sequences* s such that $\mathbf{ran}(s) \subseteq A$ by *FSeq*(A). If a variable x ranges over A, we use $\vec{x}$ to range over *FSeq*(A).

For example, $\vec{\eta}$ ranges over all finite sequences of nonterminals. Similarly, $\vec{\alpha}$ and $\vec{\beta}$ ranges over all finite sequences of symbols (see Definition 2.2). Note that these examples are parametrised by their underlying grammar.

Definition 2.2 (Context-Free Grammar)**.** A *context-free grammar* (CFG) is a quadruple $G = (N, T, P, S)$, where: (1) N is a finite set of symbols called *nonterminals*. (2) T is a finite set of symbols called *terminals*. (3) N and T are disjoint, i.e., $N \cap T = \{\}$. (4) $\Gamma(G) = N \cup T$ is the *set of symbols*. (5) P is a finite set of *grammar rules*. Each rule r is a pair of a nonterminal (called the *left-hand side*, denoted by $\mathit{LHS}(r)$) and a sequence of symbols (called the *right-hand side*, denoted by $\mathit{RHS}(r)$). We denote a rule $(\mathit{LHS}(r),\mathit{RHS}(r))$ by $\mathit{LHS}(r) \to \mathit{RHS}(r)$. (6) S is a a special nonterminal called the *start symbol*.

We use G to range over grammars, t to range over T, η to range over N, α and β to range over $\Gamma(G)$, and r to range over P.

Definition 2.3 (String)**.** A *string* is a finite sequence of symbols. We denote the *empty string* by ε (a shorthand for [], using a notation from parsing literature).

Definition 2.4 (Trees)**.** The *set of finite trees* $\mathbb{T}$ (from this point on, just "trees") is the smallest set such that for every symbol α, $\alpha\#[\mathcal{T}_1, \ldots, \mathcal{T}_k]$ is a tree, where $\mathcal{T}_1, \ldots, \mathcal{T}_k \in \mathbb{T}$ are called its *children*. A tree with no children is also called a *leaf*.

We use $\mathcal{T}$ to range over $\mathbb{T}$. We define: (1) $\mathit{rootLabel}(\alpha\#\vec{\mathcal{T}}) = \alpha$, a function which returns the label of the root of the tree. (2) $\mathit{children}(\alpha\#\vec{\mathcal{T}}) = \vec{\mathcal{T}}$, a function which returns the children of the tree. (3) $\mathit{Subtrees}(\alpha\#[\mathcal{T}_1, \ldots, \mathcal{T}_k]) = \{\alpha\#[\mathcal{T}_1, \ldots, \mathcal{T}_k]\} \cup \bigcup_{i \in 1..k} \mathit{Subtrees}(\mathcal{T}_i)$. (4) The set of proper subtrees of $\mathcal{T}$ as $\mathit{Subtrees}(\mathcal{T}) \backslash \{\mathcal{T}\}$. (5) A special symbol $\bot$ denoting "no tree", and a metavariable $\not{\mathcal{T}}$ to range over $\mathbb{T} \cup \{\bot\}$.

Definition 2.5 (String parsing)**.** Let *Parse* be the least (w.r.t. $\subseteq$) relation where:

$$\frac{t \in T}{\mathit{Parse}(G, t, [t], t\#[\,])}$$

$$\frac{(\eta,\vec{\alpha}) \in P \quad k = |\vec{\alpha}| \quad \vec{\beta_1}, \ldots, \vec{\beta_k} \quad \mathcal{T}_1, \ldots, \mathcal{T}_k \quad i \in 1..k \quad \mathit{Parse}(G, \vec{\alpha}(i), \vec{\beta_i}, \mathcal{T}_i)}{\mathit{Parse}(G, \eta, \vec{\beta_1} \cdot \ldots \cdot \vec{\beta_k}, \eta\#[\mathcal{T}_1, \ldots, \mathcal{T}_k])}$$

If $\mathit{Parse}(G, \eta, \vec{\beta}, \mathcal{T})$ holds for some $\eta \in N$, some string $\vec{\beta}$, and some tree $\mathcal{T}$, we say that $\vec{\beta}$ can be derived from η and $\mathcal{T}$ parses/is a *G-parse tree* for $\vec{\beta}$.

Definition 2.6 (Input of a tree)**.** Let $G = (N, T, P, S)$ be a grammar. Then, we define $\mathit{input}(G)$ as the smallest function such that:

- $\mathit{input}(G) \in T \to (\mathbb{T} \to \mathit{FSeq}(T))$
- For all $\mathcal{T} = \alpha\#\vec{\mathcal{T}} \in \mathbb{T}$:
  - if $\vec{\mathcal{T}} = []$ and $\alpha \in T$, $\mathit{input}(G)(\mathcal{T}) = [\alpha]$
  - if $\vec{\mathcal{T}} = []$ and $\alpha \notin T$, $\mathit{input}(G)(\mathcal{T}) = []$
  - otherwise, $\mathit{input}(G)(\mathcal{T}) = \mathit{input}(G) \circ \vec{\mathcal{T}}$.

Note that for a G-parse tree $\mathcal{T}$, $\mathit{input}(G)(\mathcal{T})$ is exactly the string that $\mathcal{T}$ parses. An ε-tree is a tree $\mathcal{T}$ such that $\mathit{input}(G)(\mathcal{T}) = \varepsilon$.

Definition 2.7 (Recursive trees)**.** We say a tree $\mathcal{T}$ is recursive, if there exists a proper subtree $\mathcal{T}'$ of $\mathcal{T}$ such that $\mathit{rootLabel}(\mathcal{T}) = \mathit{rootLabel}(\mathcal{T}')$. Let G be a grammar. If we further have $\mathit{input}(G)(\mathcal{T}) = \mathit{input}(G)(\mathcal{T}')$, we say that $\mathcal{T}$ is superfluously recursive.

Example 2.8 (Superfluously recursive trees)**.** Consider the simple grammar G expressed in Backus-Naur Form as $S \to S \mid a$. Parsing the input sentence $[a]$

yields infinitely many trees, of which only one is not superfluously recursive (and is in fact not recursive at all). The tree $S\#[a\#[\,]]$ is not recursive, and the tree $S\#[S\#a\#[\,]]$ is superfluously recursive.

Example 2.9 (An example grammar for polynomials)**.** Let $T = \{\mathit{+}, \mathit{-}, \times, \mathit{1}, \mathit{2}, x\}$, let $N = \{\mathbf{Poly}, \mathbf{Term}, \mathbf{Coef}, \mathbf{XPow}, \mathbf{Sign}, \mathbf{Num}\}$[2], let **Poly** be the start symbol (and thus $S = \mathbf{Poly}$), and let $P = \{P_+, P_-, P_T, T_{CP}, T_C, T_P, C_{SN}, Si_e, Si_-, X_x, X_X, N_1, N_2\}$ where

$$P_+ = \mathbf{Poly} \to \mathbf{Poly}\ \mathit{+}\ \mathbf{Term} \qquad P_- = \mathbf{Poly} \to \mathbf{Poly}\ \text{-}\ \mathbf{Term}$$
$$P_T = \mathbf{Poly} \to \mathbf{Term} \qquad T_{CP} = \mathbf{Term} \to \mathbf{Coef}\ \mathbf{XPow}$$
$$T_C = \mathbf{Term} \to \mathbf{Coef} \qquad T_P = \mathbf{Term} \to \mathbf{XPow}$$
$$C_{SN} = \mathbf{Coef} \to \mathbf{Sign}\ \mathbf{Num} \qquad Si_e = \mathbf{Sign} \to \varepsilon$$
$$Si_- = \mathbf{Sign} \to \text{-} \qquad X_X = \mathbf{XPow} \to \mathbf{XPow} \times \mathbf{XPow}$$
$$X_x = \mathbf{XPow} \to x \qquad N_1 = \mathbf{Num} \to \mathit{1} \quad N_2 = \mathbf{Num} \to \mathit{2}$$

Alternatively, P can be expressed in Backus-Naur Form [1] as:

$$\mathbf{Poly} \to \mathbf{Poly}\ \mathit{+}\ \mathbf{Term} \mid \mathbf{Poly}\ \text{-}\ \mathbf{Term} \mid \mathbf{Term}$$
$$\mathbf{Term} \to \mathbf{Coef}\ \mathbf{XPow} \mid \mathbf{Coef} \mid \mathbf{XPow}$$
$$\mathbf{Coef} \to \mathbf{Sign}\ \mathbf{Num}$$
$$\mathbf{Sign} \to \varepsilon \mid \text{-}$$
$$\mathbf{XPow} \to x \mid \mathbf{XPow} \times \mathbf{XPow}$$
$$\mathbf{Num} \to \mathit{1} \mid \mathit{2}$$

We will use this grammar G throughout the paper to illustrate various features of the algorithm. Note that in practice, rules for parsing **Num** would be more complex and in our grammar we simplify them for brevity.

Definition 2.10 (Nullable nonterminals)**.** The set $\mathit{null}(G) \subseteq N$, called the *set of nullable nonterminals*, is the set of all $\eta \in N$ from which the empty string ε can be derived (i.e., for which $\mathit{Parse}(G, \eta, \varepsilon, \mathcal{T})$ holds for some tree $\mathcal{T}$). We let θ (chosen for its similarity to 0) range over $\mathit{null}(G)$.

Example 2.11 (Nullable nonterminals)**.** For the grammar given in Example 2.9, we have $\mathit{null}(G) = \{\mathbf{Sign}\}$

Definition 2.12 (Initial graph)**.** Let $G = (N, T, P, S)$ be a grammar. For $r = \eta \to \vec{\theta}\ \alpha\ \vec{\beta} \in P$ and $k = |\vec{\theta}|$, let an *edge label* be a pair (r, k). Then, we can define the corresponding *edge* as the triple $(\alpha, (r, k), \eta)$. Let $\mathit{Edges}(G)$ be the set of all edges for the grammar G. We let e range over $\mathit{Edges}(G)$. For an edge $e = (\alpha, (r, k), \eta)$, we define $\mathit{start}(e) = \alpha$, $\mathit{edgeLabel}(e) = (r, k)$, and $\mathit{end}(e) = \eta$. Note that for every edge $(\alpha, (r, k), \eta)$, we have $\mathit{start}(e) = \mathit{RHS}(r)(k)$ and $\mathit{end}(e) = \mathit{LHS}(r)$. Given $\mathit{Edges}(G)$, we define the *initial graph* of G, a directed labelled graph, as $\mathit{Initial}(G) = (\Gamma(G), \mathit{Edges}(G))$.

[2] Abbreviations for "polynomial", "term", "coefficient", "power of x", "sign", and "number", respectively.

Example 2.13 (Initial graph)**.** We will illustrate the initial graph of the grammar given in Example 2.9 by drawing it. For a given pair $(\alpha, (r, k), \eta) \in \mathit{Edges}(G)$, we draw α and η as nodes with a directed edge between them, which we label with (r, k). Recall that (r, k) is a pair of a grammar rule and a natural number.

$$\begin{tikzcd}
\boxed{\text{-}} \arrow[d, "(Si_{\text{-}},0)"'] & & \mathbf{Poly} \arrow[loop left, "(P_{+},0)"] \arrow[loop right, "(P_{\text{-}},0)"] & & & \\
\mathbf{Sign} \arrow[r, red, "(C_{SN},0)"] & \mathbf{Coef} \arrow[r, bend left, "(T_{CP},0)"] \arrow[r, bend right, "(T_{C},0)"'] & \mathbf{Term} \arrow[u, "(P_{T},0)"'] & \mathbf{XPow} \arrow[l, "(T_{P},0)"] \arrow[loop right, "(X_{X},0)"] & & \\
\boxed{\mathit{1}} \arrow[r, "(N_{\mathit{1}},0)"'] & \mathbf{Num} \arrow[u, red, "(C_{SN},1)"] & \boxed{\mathit{2}} \arrow[l, "(N_{\mathit{2}},0)"] & \boxed{x} \arrow[u, "(X_{x},0)"'] & \boxed{+} & \boxed{\times}
\end{tikzcd}$$

Note that $+$ and $\times$ are drawn as nodes, but do not have any edges connecting them to the rest of the graph, since **Poly** $\notin \mathit{null}(G)$ and **XPow** $\notin \mathit{null}(G)$. Consider the two edges highlighted in red. They are both created from the same grammar rule, namely **Coef** $\rightarrow$ **Sign Num**. The bottom, vertical edge connects **Num** to **Coef** and $k = 1$, since there is a single nullable nonterminal (**Sign**) preceding **Num**. The top, horizontal edge connects **Sign** to **Coef** and $k = 0$, since there are no nullable nonterminals preceding **Sign** on the RHS of the rule. Other edges in the graph are constructed in the same fashion as the latter. For a rule $r \in P$, there is an edge from the first nonterminal in $\mathit{RHS}(r)$, to the nonterminal $\mathit{LHS}(r)$, labelled by $(r, 0)$.

Definition 2.14 (Acyclic path)**.** Let $\vec{e}$ be a non-empty finite sequence of edges. If $\mathit{end}(\vec{e}(i)) = \mathit{start}(\vec{e}(i+1))$ for all $i \in 0..(|e|-2)$, we say that $\vec{e}$ is a *path* from $\mathit{start}(\vec{e}(0))$ to $\mathit{end}(\vec{e}(|\vec{e}|-1))$. A path $\vec{e}$ is said to contain a *cycle*, if for some $i, j \in \mathbf{dom}(\vec{e})$, such that $i \neq j$, we have $\mathit{start}(\vec{e}(i)) = \mathit{start}(\vec{e}(j))$ or $\mathit{end}(\vec{e}(i)) = \mathit{end}(\vec{e}(j))$. If a path contains no cycles, we say that it is *acyclic*.

Example 2.15 (Path)**.** The following sequences are all paths from **XPow** to **Poly**:

1. $\vec{e}_1$: $[(\mathbf{XPow}, (T_P, 0), \mathbf{Term}), (\mathbf{Term}, (P_T, 0), \mathbf{Poly}), (\mathbf{Poly}, (P_+, 0), \mathbf{Poly})]$
2. $\vec{e}_2$: $[(\mathbf{XPow}, (X_X, 0), \mathbf{XPow}), (\mathbf{XPow}, (T_P, 0), \mathbf{Term}), (\mathbf{Term}, (P_T, 0), \mathbf{Poly})]$
3. $\vec{e}_3$: $[(\mathbf{XPow}, (T_P, 0), \mathbf{Term}), (\mathbf{Term}, (P_T, 0), \mathbf{Poly})]$

Of those, only $\vec{e}_3$ is acyclic, since for $\vec{e}_1$ we have $\mathit{end}(\vec{e}_1(1)) = \mathit{end}(\vec{e}_1(2)) = \mathbf{Poly}$, and for $\vec{e}_2$ we have $\mathit{start}(\vec{e}_2(0)) = \mathit{start}(\vec{e}_2(1)) = \mathbf{XPow}$.

Definition 2.16 (Neighborhood)**.** Let G be a grammar and let $\alpha, \beta \in \Gamma(G)$, and let E be the set of all paths in the initial graph from α to β. Then the *neighborhood* of α w.r.t. β, written $\mathit{neighborhood}(\alpha, \beta)$, is $\{\mathit{edgeLabel}(\vec{e}(0)) \mid \vec{e} \in E, \vec{e}\langle 1..\rangle \text{ is acyclic}\}$.

Example 2.17. The following are examples of paths in the initial graph given in Example 2.13:

1. $[(\mathbf{Coef}, (T_{CP}, 0), \mathbf{Term}), (\mathbf{Term}, (P_T, 0), \mathbf{Poly})]$
2. $[(\mathbf{Coef}, (T_C, 0), \mathbf{Term}), (\mathbf{Term}, (P_T, 0), \mathbf{Poly})]$
3. $[(\mathbf{XPow}, (X_X, 0), \mathbf{XPow}), (\mathbf{XPow}, (T_P, 0), \mathbf{Term}), (\mathbf{Term}, (P_T, 0), \mathbf{Poly})]$

4. $[(\mathbf{XPow}, (T_P, 0), \mathbf{Term}), (\mathbf{Term}, (P_T, 0), \mathbf{Poly})]$

Note that paths 1 and 2 are the only acyclic paths in the initial graph from **Coef** to **Poly**. While path 3 is not an acyclic path, it still satisfies the condition for its initial edge label to be in *neighborhood*$(\mathbf{XPow}, \mathbf{Poly})$. Therefore, we have *neighborhood*$(\mathbf{Coef}, \mathbf{Poly}) = \{(T_{CP}, 0), (T_C, 0)\}$ and *neighborhood*$(\mathbf{XPow}, \mathbf{Poly}) = \{(X_X, 0), (T_P, 0)\}$.

Definition 2.18 (Continuations)**.** We define a *continuation* as follows:

$$\mathcal{C} \in \mathbb{C} ::= \mathcal{C}^{\mathbf{m}} \mid \mathcal{C}^{\mathbf{r}} \mid \mathcal{C}^{\mathbf{d}} \qquad \mathcal{C}^{\mathbf{m}} ::= \mathcal{C}^{\mathbf{m}\alpha} \qquad \mathcal{C}^{\mathbf{d}} ::= \mathcal{C}^{\mathbf{d}\alpha} \qquad \mathcal{C}^{\mathbf{r}} ::= \mathcal{C}^{\mathbf{r}\alpha}$$
$$\mathcal{C}^{\mathbf{r}\alpha} ::= \mathbf{reduce}\ \alpha \blacktriangleright \mathcal{C}^{\star\alpha} \qquad \mathcal{C}^{\star\alpha} ::= \mathcal{C}^{\mathbf{d}\alpha} \mid \mathcal{C}^{\mathbf{m}\alpha} \qquad \mathcal{C}^{\mathbf{d}S} ::= \mathbf{done}$$
$$\mathcal{C}^{\mathbf{m}\alpha} ::= \mathbf{match}\ (r, k)\ \vec{\mathcal{T}} \blacktriangleright \mathcal{C}^{\mathbf{r}\alpha'} \text{ s.t. } \forall i \in \mathbf{dom}(\vec{\mathcal{T}}), \mathit{rootLabel}(\vec{\mathcal{T}}(i)) = \mathit{RHS}(r)(i)$$

We call the text in bold the continuation's label. We let $\mathcal{C}$ range over continuations. We call the continuation that appears inside another continuation its *successor*. For example, for the continuation $\mathcal{C}_1 = \mathbf{reduce}\ \alpha \blacktriangleright \mathcal{C}$, we say that $\mathcal{C}$ is the successor of $\mathcal{C}_1$. We denote the *set of continuations* by $\mathbb{C}$.

Continuations store the "progress" of parsing at various points in the algorithm, so incomplete trees can be completed once more symols are shifted from the input mimicking stacks found in parsers like GLR. Each continuation encodes a sequence of "steps" that will lead to a parse tree. By creating a new continuation each time a choice must be made during the algorithm's execution (e.g., between multiple rules that can be reduced), we can find all possible parse trees for a given input by only going through it once.

Definition 2.19 (*mapUnzip*)**.** Let f be a function with k arguments which returns an ordered pair of sets, and let A be a set of k-tuples. Then *mapUnzip*(f, A) is a function which returns an ordered pair of sets computed as follows:

1. **map:** Let $R = \{f(a) \mid a \in A\}$.
2. **unzip:** Return $(\bigcup_{(E_1,E_2)\in R} E_1, \bigcup_{(E_1,E_2)\in R} E_2)$.

Note that if $A = \{a\}$ for some k-tuple a, then *mapUnzip*$(f, A) = f(a)$.

We now define the procedures that make up the algorithm, starting with *parse*, which takes as input a grammar $G = (N, T, P, S)$ and sentence $\vec{\alpha}$ and returns a set of all G-parse trees for $\vec{\alpha}$, i.e., for any $\mathcal{T}$ returned by *parse*, *Parse*$(G, S, \vec{\alpha}, \mathcal{T})$ holds. We then continue with *procCont*[3], *reduce*, *match*, and finally ε*Trees*.

Procedure 2.20 (*parse*)**. Input:** A grammar G, and a string $\vec{\alpha}$ such that $\mathbf{ran}(\vec{\alpha}) \subseteq \Gamma(G)$.
Returns: A pair of a set of continuations and a set of trees.
Summary: *parse* will return the set of all non-superfluously recursive G-parse trees $\mathcal{T}$ such that *Parse*$(G, S, \vec{\alpha}, \mathcal{T})$ holds. *parse* is computed as follows:

1. Let $\hat{\mathcal{C}}_0 = \{\mathbf{reduce}\ S \blacktriangleright \mathbf{done}\}$ and let $\hat{\mathcal{T}}_0 = \varepsilon\mathit{Trees}(G, S)$.
2. For every $k \in \mathbf{dom}(\vec{\alpha})$, let $(\hat{\mathcal{C}}_{k+1}, \hat{\mathcal{T}}_{k+1}) = \mathit{mapUnzip}(\mathit{procCont}, \{(G, \mathcal{C}^{\mathbf{r}}, \{\vec{\alpha}(k)\#[\,]\}) \mid \mathcal{C}^{\mathbf{r}} \in \hat{\mathcal{C}}_k\})$. *Note: each member of* $\hat{\mathcal{C}}_k$ *is of the form* $\mathbf{reduce}\ \alpha \blacktriangleright \mathcal{C}'$.

[3] *procCont* is short for "processContinuation".

3. Return $(\hat{\mathcal{C}}_{|\vec{\alpha}|}, \hat{\mathcal{T}}_{|\vec{\alpha}|})$.

Procedure 2.21 (*procCont*)**.** **Input:** A grammar G, a continuation $\mathcal{C}$, and a set of trees $\hat{\mathcal{T}}$. For *procCont* to not return $(\{\},\{\})$, then if $\mathcal{C} = \textbf{reduce}\ \alpha \blacktriangleright \mathcal{C}'$, all trees $\mathcal{T}' \in \hat{\mathcal{T}}$ must be G-parse trees with the same root label β, such that some G-parse tree with root α contains $\mathcal{T}'$ as a subtree. If $\mathcal{C} = \textbf{match}\ (r,k)\ \vec{\mathcal{T}} \blacktriangleright \mathcal{C}'$, all trees in $\hat{\mathcal{T}}$ are G-parse trees for $\mathit{RHS}(r)(|\vec{\mathcal{T}}|)$. Top-level calls to *procCont* are always made with a continuation of the form $\textbf{reduce}\ \alpha \blacktriangleright \mathcal{C}$, and one leaf tree.
Returns: A pair of a set of continuations and a set of trees.
Summary: *procCont* uses the trees in $\hat{\mathcal{T}}$ to complete the G-parse trees which are described by $\mathcal{C}$. If shifting more symbols from the input can complete the G-parse trees, a new continuation is created for each way in which that can happen. If $\hat{\mathcal{T}} = \{\}$, *procCont* returns $(\{\},\{\})$. *procCont* is computed as follows:

1. If $\hat{\mathcal{T}} = \{\}$, return $(\{\},\{\})$.
2. If $\mathcal{C} = \textbf{done}$, return $(\{\},\hat{\mathcal{T}})$.
3. If $\mathcal{C} = \textbf{reduce}\ \alpha \blacktriangleright \mathcal{C}'$: let $(\hat{\mathcal{C}}_{new},\hat{\mathcal{T}}_{new}) = \mathit{mapUnzip}(\mathit{reduce}, \{(G,\mathcal{T},\{\},\mathcal{C}) \mid \mathcal{T} \in \hat{\mathcal{T}}\})$.
4. If $\mathcal{C} = \textbf{match}\ (r,k)\ \vec{\mathcal{T}} \blacktriangleright \mathcal{C}'$: let $(\hat{\mathcal{C}}_{new},\hat{\mathcal{T}}_{new}) = \mathit{mapUnzip}(\mathit{match}, \{(G,\bot,\textbf{match}\ (r,k)\ \vec{\mathcal{T}}\cdot[\mathcal{T}] \blacktriangleright \mathcal{C}') \mid \mathcal{T} \in \hat{\mathcal{T}}\})$.
5. Let $(\hat{\mathcal{C}}',\hat{\mathcal{T}}') = \mathit{procCont}(G,\mathcal{C}',\hat{\mathcal{T}}_{new})$.
6. Return $(\hat{\mathcal{C}}_{new} \cup \hat{\mathcal{C}}',\hat{\mathcal{T}}')$.

Procedure 2.22 (*reduce*)**.** **Input:** A grammar G, a tree $\mathcal{T} = \beta\#\vec{\mathcal{T}}$, a set of symbols $M \subseteq \Gamma(G)$ (defaults to $\{\}$ for external calls), and a continuation $\mathcal{C} = \textbf{reduce}\ \alpha \blacktriangleright \mathcal{C}'$.
Returns: A pair of a set of continuations and a set of trees.
Summary: *reduce* will attempt to build all G-parse trees $\mathcal{T}'$ such that $\mathit{rootLabel}(\mathcal{T}') = \alpha$ and $\mathcal{T} \in \mathit{Subtrees}(\mathcal{T}')$, including those that are recursive, but which are not superfluously recursive. It does so by considering all edge labels (r,k) in $\mathit{neighborhood}(\beta,\alpha)$ and building G-parse trees for their rules. All trees that have been successfully completed are returned. If a G-parse tree can be completed by shifting more symbols from the input, *reduce* will return a continuation instead. The set M ensures each neighborhood is considered at most once. *reduce* is computed as follows:

1. Let $(\hat{\mathcal{C}}_{start},\hat{\mathcal{T}}_{start}) = \mathit{mapUnzip}(\mathit{match}, \{(G,\mathcal{T},\textbf{match}\ (r,k)\ [] \blacktriangleright \mathcal{C}) \mid (r,k) \in \mathit{neighborhood}(\beta,\alpha)\})$.
2. Let $\hat{\mathcal{T}}_{done} = \{\mathcal{T}' \mid \mathcal{T}' \in (\hat{\mathcal{T}}_{start} \cup \{\mathcal{T}\}), \mathit{rootLabel}(\mathcal{T}') = \alpha\}$.
3. Let $(\hat{\mathcal{C}}_{new},\hat{\mathcal{T}}_{new}) = \mathit{mapUnzip}(\mathit{reduce}, \{(G,\mathcal{T},M \cup \{\beta\},\mathcal{C}) \mid \mathcal{T} \in \hat{\mathcal{T}}_{start}, \mathit{rootLabel}(\mathcal{T}) \notin (M \cup \{\beta\})\})$.
4. Return $(\hat{\mathcal{C}}_{start} \cup \hat{\mathcal{C}}_{new},\hat{\mathcal{T}}_{done} \cup \hat{\mathcal{T}}_{new})$.

Procedure 2.23 (*match*)**.** **Input:** A grammar G, a "maybe tree" $\mathcal{F}$, and a continuation $\mathcal{C} = \textbf{match}\ (r,k)\ \vec{\mathcal{T}} \blacktriangleright \mathcal{C}'$, such that if $\mathcal{F} = \mathcal{T}$, $\mathcal{T}$ is a G-parse tree for $\mathit{RHS}(r)(k)$. $\mathcal{F} = \bot$ if and only if $|\vec{\mathcal{T}}| > k$.
Returns: A pair of a set of continuations and a set of trees.

Summary: *match* will attempt to add an ε-tree for $RHS(r)(|\vec{\mathcal{T}}|)$ to $\vec{\mathcal{T}}$, unless $|\vec{\mathcal{T}}| = k$, in which case $\mathcal{F}$ is attached instead. After $\mathcal{F}$ has been attached, *match* will keep creating ε-trees for the rest of $RHS(r)$, until either (1) no ε-tree exists for the current symbol, or (2) $\vec{\mathcal{T}}$ contains a G-parse tree for each symbol in $RHS(r)$. If (1), *match* will return a continuation which expects to build a G-parse tree for the next unmatched symbol from $RHS(r)$ by shifting more tokens from the input. Such continuations are also created for each symbol *match* attempts to build trees for after attaching $\mathcal{F}$. If (2), *match* will return the tree $LHS(r)\#\vec{\mathcal{T}}$, if it is not superfluously recursive, and no trees otherwise. *match* is computed as follows:

1. If $|\vec{\mathcal{T}}| = |RHS(r)|$:
   - If $LHS(r)\#\vec{\mathcal{T}}$ is not superfluously recursive[4], return $(\{\},\{LHS(r)\#\vec{\mathcal{T}}\})$.
   - Otherwise, return $(\{\},\{\})$.
2. If $|\vec{\mathcal{T}}| = k$ let $\hat{\mathcal{T}} = \{\mathcal{F}\}$ and let $\mathcal{F}' = \bot$.
3. If $|\vec{\mathcal{T}}| \neq k$, let $\hat{\mathcal{T}} = \varepsilon Trees(G, RHS(r)(|\vec{\mathcal{T}}|))$ and let $\mathcal{F}' = \mathcal{F}$.
4. If $|\vec{\mathcal{T}}| > k$, let $\hat{\mathcal{C}} = \{\textbf{reduce } RHS(r)(|\vec{\mathcal{T}}|) \blacktriangleright \mathcal{C}\}$, otherwise let $\hat{\mathcal{C}} = \{\}$.
5. Let $(\hat{\mathcal{C}}_{new},\hat{\mathcal{T}}_{new}) = mapUnzip(match, \{(G,\mathcal{F}', \textbf{match } (r,k)\ \vec{\mathcal{T}}\cdot[\mathcal{T}'] \blacktriangleright \mathcal{C}') \mid \mathcal{T}' \in \hat{\mathcal{T}}\})$.
6. Return $(\hat{\mathcal{C}} \cup \hat{\mathcal{C}}_{new},\hat{\mathcal{T}}_{new})$.

Note that every continuation contained within the first element of an ordered pair returned by any function in this algorithm is created within *match*. While *reduce* also creates continuations in step 1, they are immediately passed to *match*. Given a continuation $\mathcal{C}$, *match* returns either no continuations, or a set containing only **reduce** continuations. It follows that any set $\hat{\mathcal{C}}_k$ created in *parse* contains only **reduce** continuations.

Procedure 2.24 ($\varepsilon Trees$)**. Input:** A grammar G, a nonterminal η, and optionally a set of nonterminals $M \subseteq N$, which defaults to $\{\}$ for external calls.
Returns: A (possibly empty) set of trees.
Summary: $\varepsilon Trees(G, \eta)$ returns the set of all non-recursive trees[5] with root label η that parse ε. Recursive calls to $\varepsilon Trees$ use the additional parameter M to ensure the trees are not recursive. $\varepsilon Trees$ can be computed as follows:

Consider all rules r of the form $\eta \to \vec{\theta} \in P$ such that $\textbf{ran}(\vec{\theta}) \subseteq null(G)$ and $\textbf{ran}(\vec{\theta}) \cap M = \{\}$. For each r, for all $i \in \textbf{dom}(\vec{\theta})$, let $\hat{\mathcal{T}}_i = \varepsilon Trees(G, \vec{\theta}(i), M \cup \{\eta\})$. Let $Trees = \hat{\mathcal{T}}_0 \times \ldots \times \hat{\mathcal{T}}_{|\vec{\theta}|-1}$. [6] Let $\hat{\mathcal{T}}_r = \{\eta\#[\mathcal{T}_0, \ldots, \mathcal{T}_k] \mid (\mathcal{T}_0, \ldots, \mathcal{T}_k) \in Trees\}$. Return the union of all $\hat{\mathcal{T}}_r$.

Lemma 2.25. The algorithm never builds superfluously recursive trees.

[4] Here, we filter out superfluously recursive trees built using rules of the form $\eta \to \vec{\theta}\ \eta\ \vec{\theta}'$ or sequences of rules of the form $\eta \to \vec{\theta}_1\ \eta_1\ \vec{\theta}'_1, \ldots, \eta_k \to \vec{\theta}_{k+1}\ \eta\ \vec{\theta}'_{k+1}$.

[5] Note that a recursive G-parse tree for ε is necessarily superfluously recursive, so requiring them to be non-recursive or to not be superfluously recursive is equivalent.

[6] This creates tuples of all possible combinations of G-parse trees for symbols in $\vec{\theta}$. Note that if $\varepsilon Trees$ returned an empty set for any $\vec{\theta}(i)$, then $Trees = \{\}$.

Proof. This is clear from the definitions of ε*Trees* and *match*. ε*Trees* returns only non-recursive G-parse trees. If a tree parses the empty string and is recursive, it is necessarily superfluously recursive. *match* returns a tree only if it is not superfluously recursive. $\square$

3 Parsing a sample input: $2x \times x + 1$

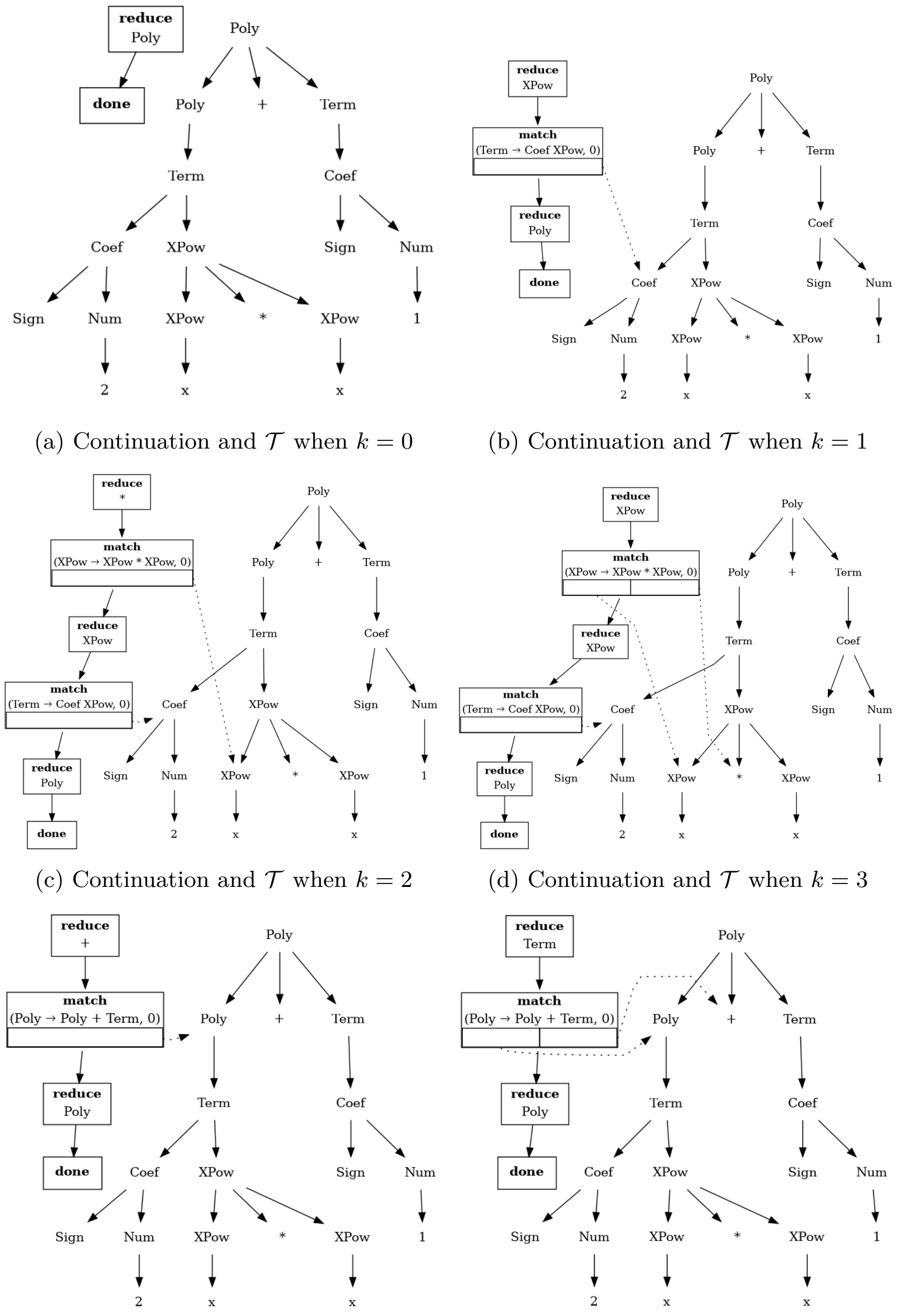
Recall the grammar G from Example 2.9. Let $\vec{\alpha} = [\mathit{2}, x, \times, x, +, x]$ and let $(\hat{\mathcal{C}},\hat{\mathcal{T}}) = \mathit{parse}(G, \vec{\alpha})$. Let $\mathcal{T}$ be the tree shown in Figure 1, which is clearly a parse tree for $\vec{\alpha}$ with root **Poly**, and thus would be in $\hat{\mathcal{T}}$. A specific sequence of continuations, starting with **reduce Poly ▶ done**, yields $\mathcal{T}$ after the last symbol is shifted. Each continuation $\mathcal{C}_{k+1}$ in the sequence is obtained by computing $\mathit{procCont}(G, \{\vec{\alpha}(k)\#[\,]\}, \mathcal{C}_k)$, for all $k \in 0..|\vec{\alpha}| - 1$. Figure 1 shows the sequence of continuations that yields $\mathcal{T}$. For this example, we have omitted all other continuations and trees that are returned by calls to *procCont*.

We will now briefly discuss any notable steps that occur while parsing $\vec{\alpha}$. After $\mathit{2}$ is shifted, $\mathit{reduce}(G, \mathit{2}\#[\,], \{\}, \textbf{reduce Poly} \blacktriangleright \textbf{done})$ is called. The algorithm considers $\mathit{neighborhood}(\mathit{2}, \textbf{Poly}) = \{(\textbf{Num} \to \mathit{2}, 0)\}$ and builds the tree $\mathcal{T}_{\textbf{Num}} = \textbf{Num}\#[\mathit{2}\#[\,]]$. Then *reduce* is recursively called again and the edge label $(\textbf{Coef} \to \textbf{Sign Num}, 1)$ is considered. **Sign** is matched to ε, and the resulting tree, alongside $\mathcal{T}_{\textbf{Num}}$, is used to complete the parse tree for **Coef**. Then, *reduce* is called recursively again and the edge label $(\textbf{Term} \to \textbf{Coef XPow}, 0)$ is considered. Since **XPow** is not nullable and the rule cannot be completed without shifting additional symbols from the input, the resulting continuation, shown in the second image in Figure 1, is returned.

After the first x is shifted from the input, the tree $\textbf{XPow}\#[x\#[\,]]$ is constructed. While this does complete the tree described by the continuation being processed (and the algorithm in fact returns it), we must also consider the recursive rule $\textbf{XPow} \to \textbf{XPow} \times \textbf{XPow}$. This results in the continuation shown in the third image in Figure 1.

After shifting $\times$ and the second x from the input, the algorithm completes and returns a parse tree with root **Poly** (the first child of the tree in Figure 1). If the end of the input had been reached, this would have resulted in a complete parse tree for our input sentence. However, we must allow for the possibility that we have not reached the end of the input, so the algorithm must consider the recursive rule $\textbf{Poly} \to \textbf{Poly} + \textbf{Term}$. The resulting continuation is shown in the fifth image in Figure 1.

After the last symbol ($\mathit{1}$) has been shifted from the input and the continuation is processed, we have completed the parse tree shown in Figure 1. In order to complete the tree, the rule $\textbf{Term} \to \textbf{Coef}$ was considered. We should note that three continuations are also returned. One expects an x to be shifted from the input, created by considering the rule $\textbf{Term} \to \textbf{Coef XPow}$ instead of $\textbf{Term} \to \textbf{Coef}$. The other two expect a $+$ or a - to be shifted, respectively, to account for the left-recursive rules $\textbf{Poly} \to \textbf{Poly} + \textbf{Term}$ and $\textbf{Poly} \to \textbf{Poly}$ - $\textbf{Term}$, after the tree has been completed.


reduce Poly
done
reduce XPow
match (Term → Coef XPow, 0)
reduce *
match (XPow → XPow * XPow, 0)
reduce +
match (Poly → Poly + Term, 0)
reduce Term
Poly
+
Term
Coef
XPow
Sign
Num
*
2
x
1

(a) Continuation and $\mathcal{T}$ when $k = 0$ (b) Continuation and $\mathcal{T}$ when $k = 1$

(c) Continuation and $\mathcal{T}$ when $k = 2$ (d) Continuation and $\mathcal{T}$ when $k = 3$

(e) Continuation and $\mathcal{T}$ when $k = 4$ (f) Continuation and $\mathcal{T}$ when $k = 5$

Fig. 1: The stages of parsing $2x \times x + 1$

4 Soundness and completeness of the algorithm

In this section, we prove that the algorithm we described in Section 2 is sound and complete. For these proofs, we define how trees can be "matched" to continuations (Definition 4.31). We then prove the algorithm is sound (Corollary 4.28), i.e., that given a grammar G and an input sentence which is in $\mathbf{L}(G)$, the algorithm always produces a valid G-parse tree for that sentence. We then show that it is complete (Theorem 4.37), i.e., that for any G-parse tree $\mathcal{T}$ for a given input sentence, the algorithm reproduces $\mathcal{T}$.

Theorem 4.26 (Termination of the algorithm)**.** Given a grammar G and an input sentence $\vec{\alpha}$, the algorithm always terminates.

Proof. The full proof can be found in the long version of the paper[1]. We first show that all the various components of the algorithm (i.e., Procedures 2.20 to 2.24) always terminate and return finite results. From this it follows that the algorithm as a whole always terminates and returns finite results. □

Theorem 4.27 (The algorithm only returns parse trees)**.** Given a grammar $G = (N, T, P, S)$ and an input sentence $\vec{\alpha} \in \mathbf{L}(G)$, let $(\hat{\mathcal{C}}, \hat{\mathcal{T}}) = \mathit{parse}(G, \vec{\alpha})$. Then if $\mathcal{T} \in \hat{\mathcal{T}}$, $\mathcal{T}$ is not superfluously recursive and $\mathit{Parse}(G, S, \vec{\alpha}, \mathcal{T})$ holds.

Proof. To construct a new tree, the algorithm considers rules in the underlying grammar G, and finding a G-parse tree for each symbol on the right-hand side of a given rule. If a tree $\mathcal{T}$ is completed, then $\mathit{Parse}(G, \mathit{rootLabel}(\mathcal{T}), \vec{\beta}, \mathcal{T})$ must hold for some $\vec{\beta}$. Any $\hat{\mathcal{T}}_k$ therefore contains only G-parse trees for some k-prefix of $\vec{\alpha}$, so any G-parse tree returned after the algorithm has processed $\vec{\alpha}$ must be a valid G-parse tree for $\vec{\alpha}$. By Lemma 2.25, the trees are not superfluously recursive. □

Corollary 4.28 (The algorithm is sound)**.** Let $G = (N, T, P, S)$ be a grammar, and let $\vec{\alpha}$ be a sentence. By Theorems 4.26 and 4.27, the algorithm is sound, i.e., for $(\hat{\mathcal{C}}, \hat{\mathcal{T}}) = \mathit{parse}(G, \vec{\alpha})$, the algorithm terminates and if $\mathcal{T} \in \hat{\mathcal{T}}$, then $\mathcal{T}$ is not superfluously recursive and $\mathit{Parse}(G, S, \vec{\alpha}, \mathcal{T})$ holds.

Definition 4.29 (Tree path)**.** Let $\alpha \# \vec{\mathcal{T}}$ be a tree. Let π be a finite sequence of natural numbers called a *tree path*. Let $\mathit{\Pi}$ be the set of all tree paths.

Definition 4.30 (Subtree at path)**.** We define *SubtreeAt* as the smallest function such that:

- $\mathit{SubtreeAt} \in \mathbb{T} \to (\mathit{\Pi} \to \mathbb{T})$
- For all $\mathcal{T} \in \mathbb{T}$, $\mathit{SubtreeAt}(\mathcal{T})([]) = \mathcal{T}$.
- For all α and for all $\vec{\mathcal{T}} \in \vec{\mathbb{T}}$ and for all $\pi \in \mathit{\Pi}$, $\mathit{SubtreeAt}(\alpha \# \vec{\mathcal{T}})([i] \cdot \pi) = \mathit{SubtreeAt}(\vec{\mathcal{T}}(i))(\pi)$ if $i \in \mathbf{dom}(\vec{\mathcal{T}})$.

Definition 4.31 (Pattern matching on parse trees)**.** Let $G = (N, T, P, S)$ be a grammar, and let $\mathcal{T}$ be a tree such that $\mathit{Parse}(G, S, \mathcal{T}, \vec{\alpha})$ holds for some $\vec{\alpha}$. Let $\mathcal{C}$ be a continuation, and let π be a tree path. We define $\mathit{PM}(\mathcal{T}, \mathcal{C}, \pi)$ (short for "Pattern Matching") as the smallest (w.r.t. $\subseteq$) relation such that:

- $\mathit{PM}(\mathcal{T}, \textbf{reduce } S \blacktriangleright \textbf{done}, [])$ holds and $\mathit{PM}(\mathcal{T}, \textbf{done}, [])$ holds.
- Suppose $\mathit{PM}(\mathcal{T}, \textbf{reduce } \alpha \blacktriangleright \mathcal{C}', \pi)$ holds and there exists π' such that $\eta\#\vec{\mathcal{T}}\cdot\vec{\mathcal{T}}' = \mathit{SubtreeAt}(\mathcal{T})\pi\cdot\pi'$. Further suppose that there exists $k < |\vec{\mathcal{T}}|$ such that $\forall j < k.\mathit{input}(G)\vec{\mathcal{T}}(j) = \varepsilon$, and let $\vec{\beta} = \mathit{rootLabel} \circ (\vec{\mathcal{T}}\cdot\vec{\mathcal{T}}')$. Then if $\eta \to \vec{\beta} \in P$, $\mathit{PM}(\mathcal{T}, \textbf{match } (\eta \to \vec{\beta}, k)\ \vec{\mathcal{T}} \blacktriangleright \textbf{reduce } \alpha \blacktriangleright \mathcal{C}', \pi\cdot\pi')$ holds.
- Suppose $\mathit{PM}(\mathcal{T}, \textbf{match } (r, k)\ \vec{\mathcal{T}} \blacktriangleright \mathcal{C}', \pi)$ holds. Let $l = |\vec{\mathcal{T}}|$, let $\mathcal{T}' = \mathit{SubtreeAt}(\mathcal{T})\pi\cdot[l]$. Further suppose that $\mathit{rootLabel}(\mathcal{T}') = \alpha$. Then $\mathit{PM}(\mathcal{T}, \textbf{reduce } \alpha \blacktriangleright \textbf{match } (r, k)\ \vec{\mathcal{T}} \blacktriangleright \mathcal{C}', \pi\cdot[l])$ holds.

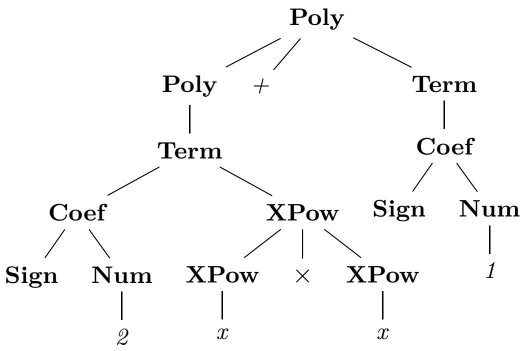


Fig. 2: G-parse tree for $\vec{\alpha}$ from Example 4.32.

Example 4.32 (Pattern matching)**.** Let G be the grammar defined in Example 2.9, and let $\vec{\alpha} = [\mathit{2}, x, \times, x, +, \mathit{1}]$. Let $\mathcal{T}$ be the G-parse tree shown in Figure 2, let $\mathcal{T}_{\textbf{Coef}} = \textbf{Coef}\#[\textbf{Sign}\#[\,], \textbf{Num}\#[\mathit{2}\#[\,]]]$, and let $\mathcal{T}_{\textbf{XPow}} = \textbf{XPow}\#[x\#[\,]]$. Let $\mathcal{C} = \textbf{reduce } \times \blacktriangleright \textbf{match } (X_X, 0)\ [\mathcal{T}_{\textbf{XPow}}] \blacktriangleright \textbf{reduce XPow} \blacktriangleright \textbf{match } (T_{CP}, 0)\ [\mathcal{T}_{\textbf{Coef}}] \blacktriangleright \textbf{reduce Poly} \blacktriangleright \textbf{done}$. Let $\pi = []\cdot[0,0]\cdot[1]\cdot[]\cdot[1]$. We have written it this way to illustrate which part of the continuation relates to each part of the path. To show that $\mathit{PM}(\mathcal{T}, \mathcal{C}, \pi)$ holds, we will work "backwards" through the continuation, showing that the conditions for PM are satisfied at every stage.

1. First, $\mathit{PM}(\mathcal{T}, \textbf{reduce Poly} \blacktriangleright \textbf{done}, [])$ always holds.
2. We get $\mathit{SubtreeAt}(\mathcal{T})[0,0] = \textbf{Term}\#[\mathcal{T}_{\textbf{Coef}}, \textbf{XPow}\#[\mathcal{T}_{\textbf{XPow}}, \times\#[\,], \mathcal{T}_{\textbf{XPow}}]]$. We have $\vec{\mathcal{T}} = [\mathcal{T}_{\textbf{Coef}}]$, and since $\mathit{input}(G)\mathcal{T}_{\textbf{Coef}} \neq \varepsilon$, $k = 0$. Therefore, $\mathit{PM}(\mathcal{T}, \textbf{match } (T_{CP}, 0)\ [\mathcal{T}_{\textbf{Coef}}] \blacktriangleright \textbf{reduce } \alpha \blacktriangleright \textbf{done}, [0,0])$.
3. We then need the path $[0,0]\cdot[|\vec{\mathcal{T}}|] = [0,0]\cdot[1]$ and check its corresponding subtree. We get $\textbf{XPow}\#[\mathcal{T}_{\textbf{XPow}}, \times\#[\,], \mathcal{T}_{\textbf{XPow}}] = \mathit{SubtreeAt}(\mathcal{T})[0,0,1] = \mathcal{T}'$. Since $\mathit{rootLabel}(\mathcal{T}') = \textbf{XPow}$, we have $\mathit{PM}(\mathcal{T}, \textbf{reduce XPow} \blacktriangleright \textbf{match } (T_{CP}, 0)\ [\mathcal{T}_{\textbf{Coef}}] \blacktriangleright \textbf{reduce } \alpha \blacktriangleright \textbf{done}, [0,0,1])$.
4. We then have $\mathit{SubtreeAt}(\mathcal{T})[0,0,1] = \textbf{XPow}\#[\mathcal{T}_{\textbf{XPow}}, \times\#[\,], \mathcal{T}_{\textbf{XPow}}]$. This gives us $\vec{\mathcal{T}} = [\mathcal{T}_{\textbf{XPow}}]$ and $k = 0$, since $\mathit{input}(G)\mathcal{T}_{\textbf{XPow}} \neq \varepsilon$. We therefore have $\mathit{PM}(\mathcal{T}, \textbf{match } (X_X, 0)\ [\mathcal{T}_{\textbf{XPow}}] \blacktriangleright \textbf{reduce XPow} \blacktriangleright \textbf{match } (T_{CP}, 0)\ [\mathcal{T}_{\textbf{Coef}}] \blacktriangleright \textbf{reduce Poly} \blacktriangleright \textbf{done}, [0,0,1])$.

5. Finally, we get $[0,0,1]\cdot[|\vec{\mathcal{T}}|] = [0,0,1,1]$. We have $\mathit{SubtreeAt}(\mathcal{T})[0,0,1,1] = \times\#[\,]$. Since $\mathit{rootLabel}(\times\#[\,])$, we get $\mathit{PM}(\mathcal{T},\mathcal{C},[0,0,1,1])$.

We will use PM to prove that given a grammar $G = (N,T,P,S)$, an input sentence $\vec{\alpha}$ and a G-parse tree $\mathcal{T}$ for that input sentence, all the functions that make up the algorithm return (1) a continuation $\mathcal{C}$ such that $\mathit{PM}(\mathcal{T},\mathcal{C},\pi)$ holds for some π, and (2) possibly a subtree of $\mathcal{T}$. Lemmas 4.33 and 4.34 prove this for *match*, Lemma 4.35 proves this for *reduce*, and Lemma 4.36 proves this for *procCont*. We then use these lemmas and an induction on the length of the input sentence to prove that the above assertions hold for *parse*, and thus the algorithm as a whole is complete (Theorem 4.37).

Lemma 4.33. Let $G = (N,T,P,S)$ be a grammar and let $\mathcal{T}$ be a tree such that $\mathit{Parse}(G,S,\mathcal{T},\vec{\alpha})$ holds for some $\vec{\alpha}$. Let $\mathcal{C} = \textbf{match}\ (r,k)\ \vec{\mathcal{T}} \blacktriangleright \mathcal{C}'$ be a continuation such that $k > |\vec{\mathcal{T}}|$ and for some π, $\mathit{PM}(\mathcal{T},\mathcal{C},\pi)$ holds. Let $(\hat{\mathcal{C}},\hat{\mathcal{T}}) = \mathit{match}(G,\bot,\mathcal{C})$. Then the following hypothesis holds:

– If $\hat{\mathcal{T}} \neq \{\}$, there exists $\mathcal{T}' \in \hat{\mathcal{T}}$ such that $\mathcal{T}' = \mathit{SubtreeAt}(\mathcal{T})(\pi)$.
– Otherwise, there exists $\mathcal{C}'' \in \hat{\mathcal{C}}$ such that for some π', $\mathit{PM}(\mathcal{T},\mathcal{C}'',\pi\cdot\pi')$ holds.

Proof. The full proof can be found in the long version of the paper[1]. The proof is by induction on $d = |\mathit{RHS}(r)| - |\vec{\mathcal{T}}|$. When $d = 0$, a non-empty set of trees is returned, for which we show the lemma holds. In the inductive step we show that the lemma holds for the continuation that is created as part of each recursive call to *match*. □

Lemma 4.34. Let $G = (N,T,P,S)$ be a grammar and let $\mathcal{T}$ be a tree such that $\mathit{Parse}(G,S,\mathcal{T},\vec{\alpha})$ holds for some $\vec{\alpha}$. Let $\mathcal{C} = \textbf{reduce}\ \alpha \blacktriangleright \mathcal{C}'$ be a continuation such that for some π, $\mathit{PM}(\mathcal{T},\mathcal{C},\pi)$ holds. Further, let π' be a path such that $\eta\#\vec{\mathcal{T}} = \mathit{SubtreeAt}(\mathcal{T})(\pi\cdot\pi')$, and let $\vec{\beta} = \mathit{rootLabel} \circ \vec{\mathcal{T}}$. Let $j \in \mathbb{N}$ such that for all $i < j$, $\mathit{input}(G)(\vec{\mathcal{T}}(i)) = \varepsilon$ and $\mathcal{T}' = \mathit{SubtreeAt}(\mathcal{T})(\pi\cdot\pi'\cdot[j])$. Finally, for $\vec{\mathcal{T}}'$ such that $j < |\vec{\mathcal{T}}'|$, let $(\hat{\mathcal{C}},\hat{\mathcal{T}}) = \mathit{match}(G,\mathcal{T}',\textbf{match}\ (\eta \to \vec{\beta}, j)\ \vec{\mathcal{T}}' \blacktriangleright \mathcal{C})$. Then the following hypothesis holds:

– If $\hat{\mathcal{T}} \neq \{\}$, we have $\eta\#\vec{\mathcal{T}} \in \hat{\mathcal{T}}$.
– Otherwise, there exists $\mathcal{C}' \in \hat{\mathcal{C}}$ such that $\mathit{PM}(\mathcal{T},\mathcal{C}',\pi\cdot\pi')$ holds.

Proof. The full proof can be found in the long version of the paper[1]. The proof is by induction on $l = k - |\vec{\mathcal{T}}|$. When $l = 0$, Lemma 4.33 applies. In the inductive step we show that the lemma holds for the continuation that is created as part of each recursive call to *match*. □

Lemma 4.35. Let $G = (N,T,P,S)$ be a grammar and let $\mathcal{T}$ be a tree such that $\mathit{Parse}(G,S,\mathcal{T},\vec{\alpha})$ holds for some $\vec{\alpha}$. Let $\mathcal{C} = \textbf{reduce}\ \alpha \blacktriangleright \mathcal{C}'$ be a continuation such that for some π, $\mathit{PM}(\mathcal{T},\mathcal{C},\pi)$ holds. Further, let $\mathcal{T}' = \beta\#\vec{\mathcal{T}} = \mathit{SubtreeAt}(\mathcal{T})(\pi\cdot\pi')$ for some π'. Let $(\hat{\mathcal{C}},\hat{\mathcal{T}}) = \mathit{reduce}(G,\mathcal{T}',M,\mathcal{C})$ where $M \subseteq \Gamma(G)$ such that $\beta \notin M$. Then the following hypothesis holds:

- If $\hat{\mathcal{T}} \neq \{\}$, then we have $\mathit{SubtreeAt}(\mathcal{T})(\pi) = \alpha\#\mathcal{T}' \in \hat{\mathcal{T}}$.
- Otherwise, there exists $\mathcal{C}' \in \hat{\mathcal{C}}$ such that $\mathit{PM}(\mathcal{T}, \mathcal{C}', \pi \cdot \pi'')$ holds for some π''.

Proof. The full proof can be found in the long version of the paper[1]. The proof is by induction on $|\pi'|$. If $|\pi'| = 0$, then we have $\alpha = \beta$ and *reduce* returns $\mathcal{T}'$. Otherwise, *reduce* must recursively consider a new tree $\mathcal{T}''$, which contains $\mathcal{T}'$ as a subtree, where $\mathcal{T}'' = \mathit{SubtreeAt}(\mathcal{T})(\pi \cdot \pi' \langle 0..(|\pi'| - 2) \rangle)$. □

Lemma 4.36. Let $G = (N, T, P, S)$ be a grammar and let $\mathcal{T}$ be a tree such that $\mathit{Parse}(G, S, \mathcal{T}, \vec{\alpha})$ holds for some $\vec{\alpha}$. Let $\mathcal{C} = \textbf{reduce}\ \alpha \blacktriangleright \mathcal{C}'$ be a continuation such that for some π, $\mathit{PM}(\mathcal{T}, \mathcal{C}, \pi)$ holds. Let $\mathcal{T}' = \mathit{SubtreeAt}(\mathcal{T})(\pi)$ and let $\hat{\mathcal{T}}' = \{\mathcal{T}'\}$. Let $(\hat{\mathcal{C}}, \hat{\mathcal{T}}) = \mathit{procCont}(G, \mathcal{C}, \hat{\mathcal{T}}')$. Then, the following hypothesis holds:

- If $\hat{\mathcal{T}} \neq \{\}$, then $\mathcal{T}' \in \hat{\mathcal{T}}$ such that for some π', $\mathcal{T}' = \mathit{SubtreeAt}(\mathcal{T})(\pi')$.
- Otherwise, there exists $\mathcal{C}' \in \hat{\mathcal{C}}$ such that for some π', $\mathit{PM}(\mathcal{T}, \mathcal{C}', \pi')$ holds.

Proof. The proof is by induction on the structure of $\mathcal{C}$.
If $\mathcal{C} = \textbf{done}$, then $\mathit{PM}(\mathcal{T}, \mathcal{C}, [])$ holds and $\mathit{SubtreeAt}(\mathcal{T})([]) = \mathcal{T}$. $\mathit{procCont}(G, \mathcal{C}, \hat{\mathcal{T}}) = (\{\}, \{\mathcal{T}\})$, which satisfies the hypothesis.
If $\mathcal{C} = \textbf{reduce}\ \alpha \blacktriangleright \mathcal{C}'$, we have $(\hat{\mathcal{C}}_{new}, \hat{\mathcal{T}}_{new}) = \mathit{mapUnzip}(\mathit{reduce}, \{(G, \mathcal{T}', \{\}, \mathcal{C}) \mid \mathcal{T}' \in \hat{\mathcal{T}}'\})$. By Lemma 4.35, if $\hat{\mathcal{T}}_{new} \neq \{\}$, then there exists $\mathcal{T}' \in \hat{\mathcal{T}}_{new}$ such that $\mathcal{T}' = \mathit{SubtreeAt}(\mathcal{T})(\pi)$. We proceed by computing $\mathit{procCont}(G, \mathcal{C}', \hat{\mathcal{T}}_{new})$. Otherwise $\hat{\mathcal{T}}_{new} = \{\}$ and there exists $\mathcal{C}'' \in \hat{\mathcal{C}}_{new}$ such that $\mathit{PM}(\mathcal{T}, \mathcal{C}'', \pi \cdot \pi'')$ holds for some π''. Then $\mathit{procCont}(G, \mathcal{C}', \hat{\mathcal{T}}_{new}) = (\{\}, \{\})$ and so $\mathit{procCont}(G, \mathcal{C}, \hat{\mathcal{T}}) = (\hat{\mathcal{C}}_{new}, \{\})$, which satisfies the hypothesis.
If $\mathcal{C} = \textbf{match}\ (r, k)\ \vec{\mathcal{T}} \blacktriangleright \mathcal{C}$, we have $(\hat{\mathcal{C}}_{new}, \hat{\mathcal{T}}_{new}) = \mathit{mapUnzip}(\mathit{match}, \{(G, \bot, \textbf{match}\ (r, k)\ \vec{\mathcal{T}} \cdot [\mathcal{T}'] \blacktriangleright \mathcal{C}) \mid \mathcal{T}' \in \hat{\mathcal{T}}'\})$. By Lemma 4.33, if $\hat{\mathcal{T}}_{new} \neq \{\}$, then there exists $\mathcal{T}' \in \hat{\mathcal{T}}_{new}$ such that $\mathcal{T}' = \mathit{SubtreeAt}(\mathcal{T})(\pi)$. We proceed by computing $\mathit{procCont}(G, \mathcal{C}', \hat{\mathcal{T}}_{new})$. Otherwise $\hat{\mathcal{T}}_{new} = \{\}$ and there exists $\mathcal{C}'' \in \hat{\mathcal{C}}_{new}$ such that $\mathit{PM}(\mathcal{T}, \mathcal{C}'', \pi \cdot \pi'')$ holds for some π''. Then $\mathit{procCont}(G, \mathcal{C}', \hat{\mathcal{T}}_{new}) = (\{\}, \{\})$ and so $\mathit{procCont}(G, \mathcal{C}, \hat{\mathcal{T}}) = (\hat{\mathcal{C}}_{new}, \{\})$, which satisfies the hypothesis. □

Theorem 4.37 (Completeness of the algorithm)**.** The algorithm is complete, i.e., given a grammar $G = (N, T, P, S)$ and an input sentence $\vec{\alpha} \in \mathbf{L}(G)$, then for any G-parse tree $\mathcal{T}_{goal}$, which is not superfluously recursive such that $\mathit{Parse}(G, S, \vec{\alpha}, \mathcal{T}_{goal})$ holds, the algorithm will find it.

Proof. For all $k \in 0..|\vec{\alpha}|$, let $(\hat{\mathcal{C}}_k, \hat{\mathcal{T}}_k) = \mathit{parse}(G, \vec{\alpha}\langle 0..k \rangle)$. We now prove the following by induction on k:

IH: If $k = |\vec{\alpha}|$, then $\mathcal{T}_{goal} \in \hat{\mathcal{T}}_k$. Otherwise, there exists $\mathcal{C} \in \hat{\mathcal{C}}_k$ such that $\mathit{PM}(\mathcal{T}_{goal}, \mathcal{C}, \pi)$ holds for some π.

For $k = 0$, this is clear. We have $\hat{\mathcal{C}}_0 = \{\textbf{reduce}\ S \blacktriangleright \textbf{done}\}$ and $\hat{\mathcal{T}}_0 = \varepsilon\mathit{Trees}(G, S)$. If $\vec{\alpha} = []$, then $\mathcal{T}_{goal} \in \hat{\mathcal{T}}_0$, by Procedure 2.24. Otherwise, $\mathit{PM}(\mathcal{T}_{goal}, \textbf{reduce}\ S \blacktriangleright \textbf{done}, [])$ holds by Definition 4.31.

Now assume that IH holds for some $k < |\vec{\alpha}|$. Note that $(\hat{\mathcal{C}}_{k+1},\hat{\mathcal{T}}_{k+1}) = \mathit{parse}(G, \vec{\alpha}\langle 0..k+1\rangle)$. Let $t_k = \vec{\alpha}(k)$. By assumption, we have $\mathcal{C} \in \hat{\mathcal{C}}_k$ and π such that $\mathit{PM}(\mathcal{T}_{goal}, \mathcal{C}, \pi)$ holds (note that $\mathcal{T}_{goal} \notin \hat{\mathcal{T}}_k$).
Let $(\hat{\mathcal{C}},\hat{\mathcal{T}}) = \mathit{procCont}(G, \mathcal{C}, \{t_k\#[\,]\})$. Note that $\hat{\mathcal{C}} \subseteq \hat{\mathcal{C}}_{k+1}$ and $\hat{\mathcal{T}} \subseteq \hat{\mathcal{T}}_{k+1}$. By Lemma 4.36, if $\hat{\mathcal{T}} \neq \{\}$, then $\mathcal{T} \in \hat{\mathcal{T}}$ such that for some π, $\mathcal{T} = \mathit{SubtreeAt}(\mathcal{T}_{goal})(\pi)$. Note that if $k+1 = |\vec{\alpha}|$, we have $\mathcal{T}_{goal} \in \hat{\mathcal{T}}_{k+1}$. If $\hat{\mathcal{T}} = \{\}$, then we have $\mathcal{C} \in \hat{\mathcal{C}}$ such that $\mathit{PM}(\mathcal{T}_{goal}, \mathcal{C}, \pi')$ holds for some π', satisfying IH.
We have therefore shown that if IH holds for k, it holds for $k+1$, concluding the proof. □

5 Conclusion

In this paper, we identified DynGenPar as an algorithm which can potentially be used to parse the language of mathematics. From its available descriptions (ADGP) and C++ implementation (KDGP), we extracted CDGP, a CFG parsing core of KDGP. We presented a formalisation of a table-less parsing algorithm, capable of supporting left-recursive rules, rules which do not consume additional tokens when expanded. We have proven its soundness and completeness and made available an experimental implementation, which closely follows the formal description. This work provides a foundation for further reasoning about the algorithm.

In the future, we would like to:

- Formalise and verify the soundness of KDGP features we omitted, such as unification of parse trees and continuations, and parse actions, as we believe this could lead to improvements in efficiency.
- Investigate and document implementation choices that can lead to efficiency improvements.
- Formalise the effects of modifying the initial graph during parsing.
- Prove some properties about the initial graph, especially regarding its time and space complexity.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. 1st ed. Addison-Wesley, 1986. 796 pp. ISBN: 0-201-10088-6.
2. Krasimir Angelov. "Incremental Parsing with Parallel Multiple Context-Free Grammars". In: *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*. The 12th Conference of the European Chapter of the Association for Computational Linguistics. Athens, Greece: Association for Computational Linguistics, 2009, pp. 69–76. DOI: 10.3115/1609067.1609074. URL: http://portal.acm.org/citation.cfm?doid=1609067.1609074 (visited on 04/18/2022).
3. Igor Dejanović. "Parglare: A LR/GLR Parser for Python". In: *Science of Computer Programming* 214 (2021). ISSN: 0167-6423. DOI: 10.1016/j.scico.2021.102734. URL: https://www.sciencedirect.com/science/article/pii/S0167642321001271.

4. Mehmet Dolgun. GLRParser: *A GLR Parser for Natural Language Processing and Translation*. Version 0.3.25. URL: https://github.com/mdolgun/GLRParser (visited on 05/02/2023).
5. Mohan Ganesalingam. *The Language of Mathematics*. Vol. 7805. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013. ISBN: 978-3-642-37011-3. DOI: 10.1007/978-3-642-37012-0. URL: http://link.springer.com/10.1007/978-3-642-37012-0 (visited on 10/28/2021).
6. Kevin Kofler. "Dynamic Generalized Parsing and Natural Mathematical Language". PhD thesis. Vienna, Austria: University of Vienna, 2017. URL: https://www.tigen.org/kevin.kofler/fmathl/dyngenpar/diss.pdf.
7. Kevin Kofler and Arnold Neumaier. "DynGenPar – A Dynamic Generalized Parser for Common Mathematical Language". In: *Intelligent Computer Mathematics*. Ed. by Johan Jeuring et al. Berlin, Heidelberg: Springer, 2012, pp. 386–401. ISBN: 978-3-642-31374-5. DOI: 10.1007/978-3-642-31374-5_26.
8. Michael Kohlhase and Dennis Müller. *The sTeX3 Manual*. Oct. 13, 2023. URL: https://github.com/slatex/sTeX/blob/main/doc/stex-manual.pdf (visited on 10/17/2023).
9. Paul T. McGuire. *Welcome to PyParsing's Documentation!* — PyParsing 3.1.1 Documentation. URL: https://pyparsing-docs.readthedocs.io/en/latest/index.html (visited on 04/04/2024).
10. Rahman Nozohoor-Farshi. "GLR Parsing for "-Grammers". In: *Generalized LR Parsing*. Ed. by Masaru Tomita. Boston, MA: Springer US, 1991, pp. 61–75. ISBN: 978-1-4615-4034-2. DOI: 10.1007/978-1-4615-4034-2_5. URL: https://doi.org/10.1007/978-1-4615-4034-2_5 (visited on 05/01/2025).
11. Jan Rekers. "Parser Generation for Interactive Environments". University of Amsterdam, Jan. 1992. 179 pp. URL: https://ir.cwi.nl/pub/29930/29930D.pdf (visited on 02/21/2024).
12. Masaru Tomita. "The Generalized LR Parsing Algorithm". In: *Generalized LR Parsing*. Springer Science & Business Media, Aug. 31, 1991, pp. 1–16. ISBN: 978-0-7923-9201-9. Google Books: PvZiZiVqwHcC.
13. Luka Vrečar, Joe Wells, and Fairouz Kamareddine. "Towards Semantic Markup of Mathematical Documents via User Interaction". In: *Intelligent Computer Mathematics*. Ed. by Andrea Kohlhase and Laura Kovács. Vol. 14960. Cham: Springer Nature Switzerland, 2024, pp. 223–240. ISBN: 978-3-031-66996-5. DOI: 10.1007/978-3-031-66997-2_13. URL: https://link.springer.com/10.1007/978-3-031-66997-2_13 (visited on 10/25/2024).
14. Daniel H. Younger. "Recognition and parsing of context-free languages in time n3". In: *Information and Control* 10.2 (1967), pp. 189–208. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(67)80007-X. URL: https://www.sciencedirect.com/science/article/pii/S001999586780007X.