# Generation and Polynomial Parsing of Graph Languages with Non-Structural Reentrancies

Johanna Björklund
Department of Computing Science
Umeå University
johanna@cs.umu.se

Frank Drewes
Department of Computing Science
Umeå University
drewes@cs.umu.se

Anna Jonsson
Department of Computing Science
Umeå University
aj@cs.umu.se



*Graph-based semantic representations are popular in natural language processing, where it is often convenient to model linguistic concepts as nodes and relations as edges between them. Several attempts have been made to find a generative device that is sufficiently powerful to describe languages of semantic graphs, while at the same allowing efficient parsing. We contribute to this line of work by introducing graph extension grammar, a variant of the contextual hyperedge replacement grammars proposed by Hoffmann et al. Contextual hyperedge replacement can generate graphs with* non-structural reentrancies*, a type of node-sharing that is very common in formalisms such as abstract meaning representation, but that context-free types of graph grammars cannot model. To provide our formalism with a way to place reentrancies in a linguistically meaningful way, we endow rules with logical formulas in counting monadic second-order logic. We then present a parsing algorithm and show as our main result that this algorithm runs in polynomial time on graph languages generated by a subclass of our grammars, the so-called local graph extension grammars.*

## 1. Introduction

Formal graph languages are commonly used to represent the semantics of natural and artificial languages. They are exceptionally versatile, lend themselves to human interpretation (in contrast to, for example, vector-based semantic representations), and have

Action Editor: Carlos Gómez-Rodríguez. Submission received: 8 December 2022; revised version received: 7 April 2023; accepted for publication: 20 May 2023.

https://doi.org/10.1162/coli_a_00488

a comprehensive mathematical theory. Recent applications are the abstract meaning representations (AMRs) (Langkilde and Knight 1998; Banarescu et al. 2013) that capture the semantics of natural language sentences, and the work of Allamanis, Brockschmidt, and Khademi (2018), who use graphs to represent both the syntactic and semantic structure of source code. In these cases, it is common to represent objects as nodes, and relations as directed edges. Probabilities and other weights are sometimes added to reflect quantitative aspects such as likelihoods and uncertainties.

In this work, we propose the graph extension grammar for modeling languages of graph-based semantic representations. We formalize these grammars in a tree-based fashion in the sense of Drewes (2006). Thus, a grammar consists of a graph algebra $\mathcal{A}$ and a tree grammar $g$. The trees generated by $g$ are well-formed expressions over the operations of $\mathcal{A}$. Each generated tree thus evaluates into a graph, meaning that the tree language generated by $g$ evaluates to a graph language. If we are careful about how we construct and combine $g$ and $\mathcal{A}$, we can make parsing efficient. In other words, given a graph $G$, we can find a tree in the language of $g$ that evaluates to $G$ under $\mathcal{A}$—or decide that no such tree exists, meaning that $G$ is not in the graph language specified by $g$ and $\mathcal{A}$—in polynomial time. The main contribution of this work is the design of the algebra.

As a guiding example, we take the aforementioned AMR. (Note that since this article focuses on parsing in terms of membership problem solving, we do not go into the extensive string-to-AMR parsing literature.) AMR is characterized by its graphs being directed, acyclic, and having unbounded node degree. The concept was first introduced by Langkilde and Knight (1998) based on a semantic abstraction language by Kasper (1989). The notion was refined and popularized by Banarescu et al. (2013) and instantiated for a limited domain by Braune, Bauer, and Knight (2014). To ground AMR in formal language theory, Chiang et al. (2018) analyze the AMR corpus of Banarescu et al. (2013). They note that even though the node degree is generally low in practice, this is not always the case, which speaks in favor of models that allow an unbounded node degree. Regarding the treewidth of the graphs in the corpus, they find that it never exceeds 4 and conclude that an algorithm can depend exponentially on this parameter and still be feasible in practice.

In the context of semantic graphs, it is common to talk about **reentrancies**. Figure 1 illustrates this concept with a pair of AMR graphs, both of which require node sharing,

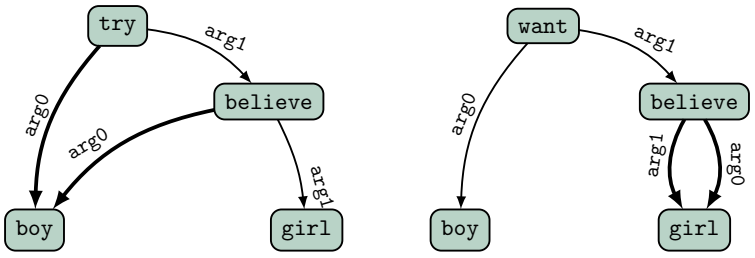



**Figure 1**
Semantic graphs for the pair of sentences "The boy tries to believe the girl" and "The boy wants the girl to believe herself". The nodes represent the concepts "boy", "girl", "try", "want", and "believe". The edges labeled "arg0" give the agent for each verb, and the edges labeled "arg0" the patient. The bold edges indicate reentrancies: on the left-hand side the structural reentrancy needed for the control verb "try", and on the right-hand side a non-structural one used for co-referencing. In the latter case, the "arg1" edge could equally well have targeted the "boy" node, in which case the semantics would be different but still sound.

or **reentrant edges**, to express the correct semantics. We propose to distinguish between two types of reentrancies: In the first graph, the reentrancy is **structural**, meaning that the boy must be the agent of whatever he is trying, so the "arg0" edge can only point to him. In the second graph, the reentrancy is **non-structural** in the sense that although in this particular case the girl believes herself, there is nothing to prevent her from believing someone else, or that someone else believes her. In general, we speak of (1) structural reentrancies when they are syntactically governed, for example, by control or coordination, and of (2) non-structural reentrancies when they can in principle refer to any antecedent, in some sense disregarding the structure of the graph in favor of the roles of the concepts. Structural reentrancy can, for example, also take the form of subject control as in "They persuaded him to talk to her", where the person who talks and the person who was persuaded must be one and the same. In contrast, "[...], but she liked them" is an example of (2) since the antecedent of "them" may be picked from anywhere in "[...]" for the semantic representation to be valid.

A second important characteristic of AMR and other notions of semantic graphs in natural language processing is that certain types of edges, like those in Figure 1, point to *arguments* of which there should be at most one. For example, the concept "try" in Figure 1 must only have one outgoing "arg0" edge. Other edges, of types not present in Figure 1, can for example represent modifiers, such as believing *strongly* and *with a passion*. Outgoing edges of these types are usually not limited in number.

It is known that contextual hyperedge replacement grammars (CHRGs), an extension of the well-known context-free hyperedge replacement grammars (HRGs), can model both structural and non-structural reentrancies (Drewes and Jonsson 2017). To achieve this, contextual hyperedge replacement rules may contain so-called context nodes which, during the application of a rule, are identified with *any* appropriately labeled node of the graph to which the rule is applied. In particular, this allows for the generation of non-structural reentrancies, using rules that contain edges pointing to context nodes. Despite this added generative power compared to HRGs, previous research on CHRGs has resulted in a pumping lemma (Berglund 2019) for the generated graph languages and a parser generator (Drewes, Hoffmann, and Minas 2017, 2021, 2022) that, for a certain subclass of CHRGs, yields a parser that runs in quadratic (and in the common case, linear) time in the size of its input graph.[1] However, similarly to LL- and LR-parsers for ordinary context-free languages, and in contrast to our algorithm, the parser generator may discover parsing conflicts. In this case it is unable to construct a parser. Compared to the string case, the possible reasons for conflicts are much subtler, which makes grammar construction a complex and error-prone manual endeavor.

For this reason, we are now proposing graph extension grammars, a type of graph grammar that makes use of the idea of context nodes in a different way. It enables polynomial parsing based on an entirely syntactic condition on the rules (our main result Theorem 5) while retaining the ability to specify graph languages with reentrancies of both type (1) and (2). One key property of these grammars that allows for polynomial parsing is that, intuitively, nodes are provided with all of their outgoing edges the moment they are created. Using ordinary contextual hyperedge replacement, this would thus result in graph languages of bounded out-degree, the bound being given by the maximal number of outgoing edges of nodes in the right-hand sides of productions. To generate semantic graphs in which a concept can have an arbitrary number of dependents, we use a technique from Drewes et al. (2010) known as **cloning**. Here,

1 The graph parsing tool Grappa is available at: `https://www.unibw.de/inf2/grappa`.

tree-based graph generation is helpful because it allows us to incorporate both contextuality and cloning in a natural way without sacrificing efficient parsability. While tree-based hyperedge replacement grammars are well-known to be equivalent to ordinary ones (see below), the tree-based formulation does make a difference in the contextual case as it avoids the problem of cyclic dependencies that has not yet been fully characterized and can make parsing intractable (Drewes, Hoffmann, and Minas 2019).

The creation of reentrant edges via context nodes requires a control mechanism that ensures that edges are placed in a meaningful way. One must, for example, make certain that co-references refer to entities of the correct type and plurality. For such conditions, we use monadic second-order (mso) logic since it has many well-known ties to the theory of hyperedge replacement. In particular, the theorem commonly known as Courcelle's theorem (see, e.g., Courcelle and Engelfriet 2012) states that mso formulas can be evaluated in linear time on graphs of bounded treewidth. The theorem is thus applicable to graphs generated by HRGs, because they generate graph languages of bounded treewidth. Unfortunately, the addition of non-structural reentrancies destroys this property and can cause a generated graph to have a treewidth proportional to the size of the graph. Despite this, we show in our main result that, if the mso formulas involved are *local* in a sense that restricts their ability to make general statements about the structure of the graph, then Courcelle's theorem can be exploited to solve the membership problem in polynomial time. We also show that the membership problem is NP-complete if no restriction is imposed on the formulas.

We first prove our main result for graph extension grammars that are **edge agnostic**, meaning that their mso formulas are not allowed to make use of the edge predicate. Since this is a rather severe limitation that does not allow the placement of reentrant edges to be controlled by structural conditions at all, we show afterwards how it can be relaxed. We do so by allowing the logic to make use of predicates which, for a given node, state that this node belongs to a part of the graph having a certain form. The resulting *local* graph extension grammars are much more general than the edge-agnostic ones, but they still allow for polynomial parsing. We note here that, in fact, our algorithm works correctly even without any such restriction, the only downside being a lack of efficiency. With this in mind, it may be worth noting that neither edge agnosticism nor locality are needed to be able to apply Courcelle's theorem if we are only interested in parsing graphs of bounded treewidth. For AMR, by checking the AMR bank, Chiang et al. (2018) observed that, although there is no theoretical upper limit on the treewidth of AMR graphs, in practice AMR graphs of treewidth larger than 5 are extremely rare. Hence, if one is willing to place such a bound on the treewidth of acceptable AMR graphs, then Courcelle's theorem and thus our result can be used even without the locality assumption.

Regardless, it is our belief that locality, from a practical point of view, is not a particularly severe restriction. In fact, even edge agnosticism may be tolerable in many practical cases. This is because, as long as we are interested in the generation of AMR-like structures, the primary use of context nodes is to be able to describe reentrancies caused by language elements such as pronouns. Simplifying the situation only a little, one may say that a pronoun may refer to any suitable antecedent in the sentence, wherever and in whichever role it occurs. This is essentially saying that their placement is edge agnostic even from a linguistic point of view. For example, if the pronoun "he" occurs in a sentence (or its AMR graph) that also has an occurrence of "Bob" then the former may refer to the latter, regardless of which other edges point to Bob. The only thing that (usually) matters is whether the label "Bob" refers to a person that uses the personal pronoun "he".

### 1.1 Related Work

Tree-based generation dates back to the seminal article by Mezei and Wright (1967), which generalizes context-free languages to languages over arbitrary domains, by evaluating the trees of a regular tree language with respect to an algebra.[2] Operations on graphs were used in this way for the first time by Bauderon and Courcelle (1987). See the textbook by Courcelle and Engelfriet (2012) for the eventual culmination of this line of work. Graph operations similar to those used here (though without the contextual extension) appeared first in Courcelle's work on the mso logic of graphs (Courcelle 1991, Definition 1.7). Essentially, if the right-hand side of a production contains $k$ nonterminal hyperedges, it is viewed as an operation that takes $k$ hypergraphs as arguments (corresponding to the hypergraphs generated by the $k$ subderivations) and returns the hypergraph obtained by replacing the nonterminals in the right-hand side by those $k$ argument hypergraphs. By context-freeness, evaluating the tree language that corresponds to the set of derivation trees of a grammar of such productions (which is a regular tree language) yields the same set of hypergraphs as is generated by the grammar itself.

This two-step approach is somewhat similar to that of lexical-functional grammar (LFG) by Kaplan and Bresnan (1982): In LFG, a c-structure containing syntactical information about a sentence and an f-structure that provides its semantic information are combined to create a representation for a language user's syntactic knowledge. The c-structure is comparable to a derivation tree, while the semantic f-structure that applies semantic knowledge on top of that can be compared to the algebraic operations on graphs. Using tree formalisms for capturing linguistic aspects such as co-occurrence is not new: See, for example, the work by Joshi and Levy (1982) for the usage of trees to impose local constraints on sentences. Similarly, Carroll et al. (1999) investigate the practical implications of the extended domain of locality of tree-adjoining grammars.

Several formalisms have been put forth in the literature to describe graph-based semantic representations in general and AMR in particular. Most of these can be seen as variations of HRGs (see Habel and Kreowski 1987; Bauderon and Courcelle 1987; Drewes, Kreowski, and Habel 1997). It was established early that unrestricted HRGs can generate NP-complete graph languages (Aalbersberg, Rozenberg, and Ehrenfeucht 1986; Lange and Welzl 1987), so restrictions are needed to ensure efficient parsing. To this end, Lautemann (1990) proposes a CYK-like membership algorithm and proved that it runs in polynomial time provided that the language satisfies the following condition: For every graph $G$ in the language, the number of connected components obtained by removing $s$ nodes from $G$ is in $O(\log n)$, where $n$ is the number of nodes of $G$ and the constant $s$ depends on the grammar. Lautemann's algorithm is refined by Chiang et al. (2013) to make it more suitable for natural language processing (NLP) tasks, but the algorithm is exponential in the node degree of the input graph.

In a parallel line of work, Quernheim and Knight (2012) propose automata on directed acyclic graphs (DAGs) for processing feature structures in machine translation. Chiang et al. (2018) invent an extended model of these DAG automata, focusing on semantic representations such as AMR. For this, the left- and right-hand sides in their DAG automata may take the form of restricted regular expressions. Blum and Drewes (2019) complement this work by studying language-theoretic properties of the DAG

2 Mezei and Wright formulate this in terms of systems of equations, but the essential ideas are the same.

automata, establishing, among other things, that equivalence and emptiness are decidable in polynomial time.

Various types of graph algebras for AMR parsing are described in the work by Groschwitz et al. (see, e.g., Groschwitz, Koller, and Teichmann 2015; Groschwitz et al. 2017, 2018; Lindemann, Groschwitz, and Koller 2019, 2020). A central objective is to find linguistically motivated restrictions that can efficiently be trained from data. An algorithm based on such an algebra for translating strings into semantic graphs is presented by Groschwitz et al. (2018): operations of arity zero denote graph fragments, and operations of arity two denote binary combinations of graph fragments into larger graphs. The trees over the operations of this algebra mirror the compositional structure of semantic graphs. The approach differs from ours in that their graph operations are entirely deterministic, and that neither context nodes nor cloning are used. Moreover, as is common in computational linguistics, evaluation is primarily empirical.

Through another set of syntactic restrictions on HRGs, Björklund et al. (2021) and Björklund, Drewes, and Ericson (2019) arrive at order-preserving DAG grammars (OPDGs)—with or without weights—which can be parsed in linear time. Intuitively, the restrictions ensure that each generated graph can be uniquely represented by a tree interpreted by a particular graph algebra. Despite their restrictions, OPDGs can describe central structural properties of AMR, but their limitation lies in the modeling of reentrancies. Of the previously discussed types of reentrancies, type (1) can to a large extent be modeled using OPDGs. Modeling type (2) cannot be done (except in very limited cases) since it requires attaching edges to the non-local context in a stochastic way, which cannot be achieved using hyperedge replacement alone.

The CHRGs by Drewes, Hoffmann, and Minas (2012) extend the ordinary HRGs with so-called *contextual* rules, which allow for isolated nodes in their left-hand sides. Contextual rules can reference previously generated nodes that are not attached to the replaced nonterminal hyperedge and add structure to them. Even though this formalism is strictly more powerful than HRG, it inherits several of the nice properties of HRG. In particular, there are useful normal forms and the membership problem is in NP (Drewes and Hoffmann 2015) for certain subclasses. However, as indicated above, the conditions defining these subclasses are semantic ones and thus difficult to handle. Here, we set out to use the idea of context nodes, enriched by a benign form of cloning, in a different way to develop a type of graph grammar that is sufficiently descriptive to model structural and non-structural phenomena in semantic graphs such as AMR, while allowing for easily verifiable syntactic conditions that result in polynomial time membership tests.

A very preliminary version of this work was published as Björklund, Drewes, and Jonsson (2021), the main result being a parsing algorithm that runs in time $O(n^{2\tau+1})$ where $\tau$ is the maximal type of any extension operation in the grammar (see Sections 2 and 3 for definitions). The graph extension grammars in that version restrict the matching of context nodes to nodes in the argument graph solely via node labels (that is, their node labels have to be identical). The use of mso formulas for that purpose is more general, simplifies the formalism, and allows us to take advantage of Courcelle's theorem in the main proof of the article while preserving the upper bound on the running time of $O(n^{2\tau+1})$. Furthermore, it enables us to extend polynomial parsing to so-called local graph extension grammars. Their extension operations use a logical vocabulary enriched by so-called local node predicates, thus relaxing the assumption of edge agnosticism.

The remainder of this article is structured as follows. In the next section, we gather the basic definitions regarding graphs and logic on graphs. In Section 3 we introduce

graph extension grammars and discuss an example. Section 4 shows that graph extension grammars can generate NP-complete graph languages. Readers who are less interested in NP-completeness results may safely skip that section, or consider it as another illustration of the concept of graph extension grammars. The major technical section is Section 5, in which we present the parsing algorithm and show that it can be implemented to run in polynomial time for edge-agnostic graph extension grammars. Subsequently, in Section 6, we show how the edge agnosticism requirement can be relaxed without sacrificing polynomial parsing, by turning to local graph extension grammars. Finally, Section 7 concludes.

## 2. Preliminaries

We first recall standard definitions from discrete mathematics, automata theory, and logic. The set of natural numbers (including 0) is denoted by $\mathbb{N}$, and for $n \in \mathbb{N}$, we let $[n]$ abbreviate $\{1, \ldots, n\}$. In particular, $[0] = \emptyset$. For a set $S$, we denote by $\wp(S)$ the powerset of $S$. The set of all finite sequences over $S$ is denoted by $S^*$; the subset of $S^*$ containing only those sequences in which no element of $S$ occurs twice is written $S^{\circledast}$. Both contain the empty sequence $\varepsilon$. For a string $w \in S^*$, we let $[w]$ denote the set of all elements of $S$ occurring in $w$.[3] The canonical extensions of a function $f\colon S \to T$ to $S^*$ and to $\wp(S)$ are denoted by $f$ as well, that is, $f(s_1 \cdots s_n) = f(s_1) \cdots f(s_n)$ for $s_1, \ldots, s_n \in S$, and $f(S') = \{f(s) \mid s \in S'\}$ for $S' \in \wp(S)$. The composition of $f$ and a function $g\colon R \to S$ is the function $f \circ g\colon R \to T$ defined as $(f \circ g)(r) = f(g(r))$ for all $r \in R$. The restriction of $f$ to $S' \subseteq S$ is denoted by $f|_{S'}$. We let the **priority union** of two functions $f_1\colon S_1 \to T_1$ and $f_2\colon S_2 \to T_2$ be the function $f_1 \sqcup f_2\colon S_1 \cup S_2 \to T_1 \cup T_2$ given by

$$(f_1 \sqcup f_2)(s) = \begin{cases} f_1(s) & \text{if } s \in S_1 \\ f_2(s) & \text{if } s \in S_2 \setminus S_1 \end{cases}$$

Hence, in case of conflicts the priority union gives priority to the first operator and is thus not commutative.

We sometimes use boldface letters $x$ to denote (finite) indexed sequences, whose individual elements are then denoted as $x_i$, that is, $x = x_1 \cdots x_k$ where $k = |x|$. Alternatively, we may sometimes denote $x_i$ by $x(i)$.

### Trees and Algebras

A **ranked alphabet** is a pair $A = (\Sigma, rk)$ such that $\Sigma$ is a finite set of symbols and $rk\colon \Sigma \to \mathbb{N}$ is a function that assigns to every $f \in \Sigma$ a rank. We usually keep $rk$ implicit, thus notationally identifying $A$ with $\Sigma$, and write $f^{(k)}$ to indicate that $rk(f) = k$.

The set $\mathrm{T}_\Sigma$ of all well-formed trees over a ranked alphabet $\Sigma$ is defined inductively: It is the smallest set of expressions such that, for every $f^{(k)} \in \Sigma$ and all trees $t_1, \ldots, t_k \in \mathrm{T}_\Sigma$, we have $f[t_1, \ldots, t_k] \in \mathrm{T}_\Sigma$. In particular $f[]$ for $k = 0$, which we henceforth abbreviate by $f$, is in $\mathrm{T}_\Sigma$.

We refer to the subtrees of a tree by their Gorn addresses, defining the set $addr(t) \subseteq \mathbb{N}^*$ of *addresses* in a tree $t$, and the subtrees they designate, inductively in the

3 The similarity to the notation $[n]$, $n \in \mathbb{N}$, is intentional, since both operations generate sets.

usual way: An address is a string of numbers which, when read from left to right, shows how to descend from the root of the tree down to the node in question. Every number in the string corresponds to the child number of the current node to be descended to. Thus, formally, consider a tree $t = f[t_1, \ldots, t_k]$. Then $addr(t) = \{\varepsilon\} \cup \bigcup_{i \in [k]} \{i\alpha \mid \alpha \in addr(t_i)\}$ (where the basis of the induction is the case $k = 0$). For all $\alpha \in addr(t)$, we let

$$t/\alpha = \begin{cases} t & \text{if } \alpha = \varepsilon \\ t_i/\alpha' & \text{if } \alpha = i\alpha' \text{ for some } i \in [k] \text{ and } \alpha' \in addr(t_i) \end{cases}$$

Let $\Sigma$ be a ranked alphabet. A $\Sigma$-*algebra* is a pair $\mathcal{A} = (\mathbb{A}, (f_{\mathcal{A}})_{f \in \Sigma})$ where $\mathbb{A}$ is a set called the **domain** of $\mathcal{A}$, and $f_{\mathcal{A}}$ is a function $f_{\mathcal{A}} \colon \mathbb{A}^k \to \mathbb{A}$ for every $f^{(k)} \in \Sigma$. Given a tree $t \in T_\Sigma$, the result of evaluating $t$ with respect to $\mathcal{A}$ is denoted by $val_{\mathcal{A}}(t)$: for $t = f[t_1, \ldots, t_k]$ we let $val_{\mathcal{A}}(t) = f_{\mathcal{A}}(val_{\mathcal{A}}(t_1), \ldots, val_{\mathcal{A}}(t_k))$.

To generate trees over the operations of an algebra, we use *regular tree grammars*.

**Definition 1** (Regular Tree Grammar)
A **regular tree grammar** is a tuple $g = (N, \Sigma, P, S)$ consisting of the following components:

- $N$ is a ranked alphabet of symbols, all of rank 0, called *nonterminals*.
- $\Sigma$ is a ranked alphabet of *terminals* which is disjoint with $N$.
- $P$ is a finite set of *productions* of the form $A \to f[A_1, \ldots, A_k]$ where, for $k \in \mathbb{N}, f^{(k)} \in \Sigma$, and $A, A_1, \ldots, A_k \in N$.
- $S \in N$ is the *initial nonterminal*.

We also say that $g$ is a regular tree grammar *over* $\Sigma$.

While we are going to work with algebras over graph domains, let us illustrate the general idea by considering an algebra over the rational numbers $\mathbb{Q}$. In this setting, the operations could be the standard arithmetic ones, such as $+, -, /, \times$, all binary, together with constant-valued operations in $\mathbb{Q}$, all of arity zero. It is hopefully easy to see how the trees $+[\times[2, \frac{1}{2}], 2]$ and $/[10, -[10, 4]]$ evaluate to 3 and $\frac{5}{3}$, as they are mere syntactic variants of the common arithmetic expressions $(2 \times \frac{1}{2}) + 2$ and $10/(10 - 4)$, respectively.

**Definition 2** (Regular Tree Language)
Let $g = (N, \Sigma, P, S)$ be a regular tree grammar. Then $(L_A(g))_{A \in N}$ is the smallest family of sets of trees such that, for every $A \in N$, a tree $f[t_1, \ldots, t_k]$ is in $L_A(g)$ if there is a production $A \to f[A_1, \ldots, A_k]$ in $P$ for some $k \in \mathbb{N}$ and $A, A_1, \ldots, A_k \in N$, such that $t_i \in L_{A_i}(g)$ for all $i \in [k]$. The **regular tree language** generated by $g$ is $L(g) = L_S(g)$.

### Graphs and Counting Monadic Second-Order Logic

We now define the type of graphs considered in this article, and the operations used to construct them. In short, we work with node- and edge-labeled directed graphs, each equipped with a sequence of so-called ports. From a graph operation point of view, the

sequence of ports is the "interface" of the graph; its nodes are the only ones that can individually be accessed. The number of ports determines the *type* of the graph.

**Definition 3** (Graph)
A **labeling alphabet** (or simply *alphabet*) is a pair $\mathbb{L} = (\dot{\mathbb{L}}, \bar{\mathbb{L}})$ of finite sets $\dot{\mathbb{L}}$ and $\bar{\mathbb{L}}$ of labels. A **graph** (over $\mathbb{L}$) is a system $G = (V, E, lab, port)$ where

- $V$ is a finite set of *nodes*,
- $E \subseteq V \times \bar{\mathbb{L}} \times V$ is a (necessarily finite) set of *edges*,
- $lab\colon V \to \dot{\mathbb{L}}$ assigns a node label to every node, and
- $port \in V^{\circledast}$ is a sequence of nodes called *ports*.

The *type* of $G$ is $type(G) = |port|$. The set of all graphs (over an implicitly understood labeling alphabet) of type $k$ is denoted by $\mathbb{G}_k$. For an edge $e = (u, z, v) \in E$, we let $src(e) = u$ (short for *source*) and $tar(e) = v$ (short for *target*).

In the following, if the components of a graph $G$ are not explicitly named, they will be denoted by $V_G$, $E_G$, $lab_G$, and $port_G$, respectively.

The *restriction* of a graph $G$ to $V \subseteq V_G$ and $E \subseteq E_G$ is defined if $[port_G] \subseteq V$ and $E \subseteq V \times \bar{\mathbb{L}} \times V$, and is in this case given by $(V, E, lab|_V, port_G)$.

A **morphism** from a graph $G = (V, E, lab, port)$ to a graph $G' = (V', E', lab', port')$ is a structure and label preserving function $\mu\colon V \to V'$. More precisely, we require that $lab'(\mu(v)) = lab(v)$ for all $v \in V$, $(\mu(u), z, \mu(v)) \in E'$ for all $(u, z, v) \in E$, and $\mu(port)$ is a prefix of $port'$ (that is, every port of $G$ is mapped to the respective port of $G'$, but $G'$ may contain additional ports). The fact that $\mu$ is such a morphism is also denoted by writing $\mu\colon G \to G'$. $G$ is *included in* $G'$, denoted by $G \subseteq G'$, if the identity on $V$ is a morphism from $G$ to $G'$.

A *morphing* of a graph $G = (V, E, lab, port)$ into a graph $G' = (V', E', lab', port')$ is a surjective morphism $\mu\colon G \to G'$ which is also surjective on edges and ports, that is, $E' = \{(\mu(u), z, \mu(v)) \mid (u, z, v) \in E\}$ and $port' = \mu(port)$. Given such a morphing, we denote $G'$ by $\mu(G)$, and call it a *morph* of $G$. A morphing is an **isomorphism** if it is bijective; if an isomorphism between $G$ and $G'$ exists, then these graphs are *isomorphic*.

As described in Courcelle and Engelfriet (2012), graphs are examples of (finite) relational structures. Such a structure consists of a finite set of objects (in the case of graphs these are the nodes of the graph[4]) and a finite set of relations on these objects. Selecting the set of relations to be considered and their arities determines which type of structures we talk about. To be able to view (our type of) graphs as relational structures we need unary relations $\mathbf{lab}_a$ for every node label $a \in \dot{\mathbb{L}}$, so that $\mathbf{lab}_a(v)$ is true if the node $v$ carries the label $a$. We also need unary relations $\mathbf{port}_i$ for all port numbers $i$ to express that a given node is the $i$-th port. Finally, we need binary relations $\mathbf{edg}_a$ for all $z \in \bar{\mathbb{L}}$ in order to express that there is an edge labeled $z$ between two nodes, that is, $\mathbf{edg}_z(u, v)$ is true if there is an edge labeled $z$ from $u$ to $v$.

4 We get an alternative definition of graphs as relational structures if we additionally consider edges as objects, rather than as relations as we will do here; see Courcelle and Engelfriet (2012) for a discussion of the difference.

The formal definition thus reads as follows:

**Definition 4** (Relational Structure of Graphs)
Let $G$ be a graph. Then we identify $G$ with the finite relational structure $(V_G, (\mathbf{lab}_a)_{a\in\dot{\mathbb{L}}}, (\mathbf{edg}_z)_{z\in\bar{\mathbb{L}}}, (\mathbf{port}_i)_{i\in[type(G)]})$ such that[5]

- $V_G$ is the domain (or universe) of the structure,
- for every $a \in \dot{\mathbb{L}}$, $\mathbf{lab}_a$ is the unary predicate such that $\mathbf{lab}_a(u)$ holds if and only if $lab_G(u) = a$,
- for every $z \in \bar{\mathbb{L}}$, $\mathbf{edg}_z$ is the binary predicate such that $\mathbf{edg}_z(u, v)$ holds if and only if $(u, z, v) \in E_G$, and
- for every $i \in [type(G)]$, $\mathbf{port}_i$ is the unary predicate such that $\mathbf{port}_i(u)$ holds if and only if $u = port_G(i)$.

We use predicate logic to express properties of (nodes in) graphs. While, in principle, any logic may be used, we focus on counting monadic second-order (cmso) logic. Thus, formulas can make use of individual and set variables, denoted by (possibly indexed) lowercase letters $x, y, \ldots$ and uppercase letters $X, Y, \ldots$. In the following, we let $\mathcal{V}_0$ and $\mathcal{V}_1$ be disjoint countably infinite sets of individual and set variables, respectively. The set *CMSO* of all cmso formulas expressing graph properties is inductively defined to be the smallest set of formal expressions satisfying the following conditions:

- The formulas *true* and *false* are in *CMSO*.
- For all $x, y \in \mathcal{V}_0$, $a \in \dot{\mathbb{L}}$, $z \in \bar{\mathbb{L}}$, and $i \in \mathbb{N}$, the formulas $x = y$, $\mathbf{lab}_a(x)$, $\mathbf{edg}_z(x, y)$, and $\mathbf{port}_i(x)$ belong to *CMSO*.
- For all $x \in \mathcal{V}_0$ and $X \in \mathcal{V}_1$, the formula $x \in X$ is in *CMSO*.
- For all $X \in \mathcal{V}_1$ and all $r, s \in \mathbb{N}$ with $r < s$, the formula $\mathbf{card}_{r,s}(X)$ is in *CMSO*.
- For all $\xi \in \mathcal{V}_0 \cup \mathcal{V}_1$, $Q \in \{\forall, \exists\}$, and formulas $\varphi, \varphi' \in$ *CMSO*, the formulas $(Q\,\xi.\varphi)$, $(\varphi \wedge \varphi')$, and $(\neg\varphi)$ belong to *CMSO*.

As usual, we can omit parentheses when writing down formulas if there is no danger of confusion.

A cmso formula $\varphi$ may be denoted by $\varphi(\boldsymbol{X}, \boldsymbol{x})$, where $\boldsymbol{X} \in \mathcal{V}_1^{\circledast}$ and $\boldsymbol{x} \in \mathcal{V}_0^{\circledast}$, to express the fact that the free variables occurring in $\varphi$ are in $[\boldsymbol{X}] \cup [x]$.[6] Given a graph $G$, an **assignment** appropriate for a formula $\varphi(\boldsymbol{X}, \boldsymbol{x})$ is a mapping *asg* that assigns a subset of $V_G$ to every $X \in [X]$ and an element of $V_G$ to every $x \in [\boldsymbol{x}]$. Given such an assignment, $\varphi$ can be evaluated in $G$ in the usual way, where $\mathbf{card}_{r,s}(X)$ (with $X \in \boldsymbol{X}$) is satisfied if and only if $|asg(X)| = r \pmod{s}$. If $asg(\boldsymbol{X}) = \boldsymbol{V}$ and $asg(\boldsymbol{x}) = \boldsymbol{v}$, we let $G \vDash \varphi(\boldsymbol{V}, \boldsymbol{v})$ denote the statement that $\varphi$ is satisfied (i.e., evaluates to true) in $G$ under *asg*. As usual, we can make use of other Boolean connectives such as $\vee$ and $\rightarrow$ in formulas since they can be expressed in terms of $\wedge$ and $\neg$. For example, for cmso formulas $\varphi$ and $\varphi'$, the formula

5 Courcelle and Engelfriet (2012) denote this structure by $\lfloor G \rfloor$.
6 Note that this notation does *not* imply that each of the variables actually occurs in $\varphi$.

$\varphi \to \varphi'$ is equivalent to $\neg(\varphi \wedge \neg\varphi')$, in the sense that the two formulas are satisfied by the same set of assignments, and $\varphi \vee \varphi'$ is equivalent to $\neg(\neg\varphi \wedge \neg\varphi')$.

In Section 5, we will make use of the so-called Backwards Translation theorem for quantifier-free operations. These quantifier-free operations map relational structures to other relational structures, in our case graphs to graphs. The formal definition of these operations takes some getting used to (as so much in the area of formal logic). In the next paragraphs, we thus try to convey the intuition first. However, we would also like to point out that quantifier-free operations and the Backwards Translation theorem are "merely" technical tools we use to formulate the proof of our main result. They are not used in the graph extension grammars themselves. Hence, readers who are just interested in the formalism and the overall parsing strategy may safely skip the remainder of this section.

The idea behind quantifier-free operations, which is a special case of the more general cmso operations, is to use a collection of logical formulas to describe a mapping from one type of relational structure to another. Hence, the input is a relational structure consisting of a set of objects (like the nodes of a graph) and a number of relations on these objects. The output is supposed to be a similar relational structure, though in general it can be a structure involving other relations. Now, we can use logical formulas to define both the domain of the resulting structure and each of its relations in terms of the input structure. For this, we need two things:

1. a **domain formula** $\delta$ with one free individual variable (the "argument" of the formula) which, when applied to an object in the input structure, determines whether this object is to be an object in the output structure (value *true*) or not (value *false*);[7]
2. for every relation $R_i$ of the output structure, a formula $\theta_{R_i}$ with $\beta_i$ free variables, where $\beta_i$ is the arity of $R_i$. If this formula, applied to objects $v_1, \ldots, v_{\beta_i}$ of the input structure, yields *true*, then $(v_1, \ldots, v_{\beta_i})$ is a tuple in the relation $R_i$ of the output structure (provided that all of $v_1, \ldots, v_{\beta_i}$ are in its domain, as dictated by the domain formula).

Naturally, both the domain formula and the formulas determining the output relations can make use of the relations of the input structure. The mappings definable in this way are called quantifier-free operations because we shall forbid the formulas to make use of quantifiers.

We are actually only interested in the case where both input and output structures are graphs. Hence, the purpose of $\delta$ is to pick the nodes to be included in the output graph, while the formulas $\theta_{R_i}$ define its node labels, edges, and ports.

The following formal definition provides one more element, which for simplicity was left out in the explanation above: A quantifier-free operation can have additional parameters. These parameters are represented by free set variables $X_1, \ldots, X_\ell$ in the formulas and are thus to be instantiated by sets of nodes of the input graph when the operation is "called". Hence, the operation can yield different output graphs for one

7 Thus the output structure cannot be bigger than the input structure in terms of the number of objects in it. One can remedy this by considering domain formulas with several free variables, which then determine whether a tuple of objects of the original formula is an object of the output formula, but we will not need this in the current article.

and the same input graph depending on those parameters. In later parts of the article, these parameters will be used to be able to select certain nodes that are meant to play a distinguished role in the construction formalized by the operation.

**Definition 5** (Quantifier-Free Operation on Graphs)
Let $R_1, \dots, R_k$ be a list of the relations in Definition 4, where $R_i$ has the arity $\beta_i$ for all $i \in [k]$. Let $X_1, \dots, X_\ell \in \mathcal{V}_1$ and $x, x_1, x_2, \dots \in \mathcal{V}_0$ be (pairwise distinct) variables. A **quantifier-free operation** $\xi$ **in** $X_1, \dots, X_\ell$ is specified by quantifier-free formulas $\delta, \theta_{R_1}, \dots, \theta_{R_k}$ such that

- $\delta$ is a formula with the free variables $X_1, \dots, X_\ell, x$, and
- every $\theta_{R_i}$, for $i \in [k]$, is a formula in $X_1, \dots, X_\ell, x_1, \dots, x_{\beta_i}$,

and we write $\xi = (\delta, \theta_{R_1}, \dots, \theta_{R_k})$. Given a graph $G$ and node sets $U_1, \dots, U_\ell \subseteq V_G$, its image $H = \xi(G, U_1, \dots, U_\ell)$ under $\xi$ is defined as follows:

- The nodes of $H$ are all $v \in V_G$ such that $G \models \delta(U_1, \dots, U_\ell, v)$.
- Given nodes $v_1, \dots, v_{\beta_i} \in V_H$, $R_i(v_1, \dots, v_{\beta_i})$ holds in the image $H$ if and only if $G \models \theta_{R_i}(U_1, \dots, U_\ell, v_1, \dots, v_{\beta_i})$.

*Remark 1*
A quantifier-free operation as defined above does not necessarily map graphs to graphs, but to slightly more general relational structures. More precisely, the resulting structures do not necessarily have exactly one port of each kind, as we may have $G \models \theta_{\mathbf{port}_i}(U_1, \dots, U_\ell, v_1)$ for any number of nodes $v_1 \in V_H$ (including zero). Hence, the output of a quantifier-free operation may strictly speaking not be a graph. We may disregard this formal inconsistency for two reasons. Firstly, the quantifier-free operations constructed later in this article yield true graphs by construction, meaning that the problem does not occur. Secondly, the results we are going to use (the Backwards Translation theorem and Courcelle's theorem) both apply not only to graphs but to general relational structures.

**Example 1** (Quantifier-Free Operation)
Let $k \in \mathbb{N}$, $\mathbb{L} = \{\cdot\}$, $\mathbb{L} = \{\text{—}, \text{---}\}$, and $G \in \mathbb{G}_k$ a graph, viewed as a relational structure $(V_G, (\mathbf{lab}_a)_{a \in \mathbb{L}}, (\mathbf{edg}_z)_{z \in \mathbb{L}}, (\mathbf{port}_i)_{i \in [type(G)]})$ in the way described in Definition 3. We discuss a quantifier-free operation $\xi = (\delta, (\theta_R)_{R \in \mathcal{R}})$ in the set variables $X_1, X_2, X_3 \in \mathcal{V}_1$, where $\mathcal{R} = \{\mathbf{lab}_a \mid a \in \mathbb{L}\} \cup \{\mathbf{edg}_z \mid z \in \mathbb{L}\} \cup \{\mathbf{port}_i \mid i \in [k]\}$. We choose constitute formulas of $\xi$ so that the application of $\xi$ to $G$ with set arguments $U_1, U_2, U_3 \subseteq V_G$ has the following effect: The set of nodes is restricted to $(U_1 \cup U_2) \setminus U_3$. All edges that involve a node not in $U_2$ are kept, but switch their label (from "—" to "---" or vice versa), and the nodes in $U_2$ form a clique of edges labeled "—". The output graph has no ports. To achieve this, we define the formulas as follows:

- $\delta(X_1, X_2, X_3, x) = ((x \in X_1) \vee (x \in X_2)) \wedge \neg(x \in X_3)$,
  expressing that a node (represented by the variable $x$) belongs to the output graph if and only if it is in $(X_1 \cup X_2) \setminus X_3$,

- $\theta_{\mathbf{lab}_a}(X_1, X_2, X_3, x_1) = \mathbf{lab}_a(x_1)$ for every $a \in \dot{\mathbb{L}}$,
expressing that nodes in the output graph inherit their label from the input graph,
- $\theta_{\mathbf{edg\_}}(X_1, X_2, X_3, x_1, x_2) = \mathbf{edg}_{\text{---}}(x_1, x_2) \vee (x_1 \in X_2 \wedge x_2 \in X_2)$,
expressing that there is an edge labeled "—" between two nodes of the output graph if these two nodes are connected by a "---" edge in the input graph or both belong to $X_2$,
- $\theta_{\mathbf{edg}_{\text{---}}}(X_1, X_2, X_3, x_1, x_2) = \mathbf{edg\_}(x_1, x_2) \wedge \neg(x_1 \in X_2 \wedge x_2 \in X_2)$,
expressing that there is an edge labeled "---" between two nodes of the output graph if these two nodes are connected by a "—" edge in the input graph and at least one of them does not belong to $X_2$,
- $\theta_{\mathbf{port}_i}(X_1, X_2, X_3, x) = \mathit{false}$ for every $i \in [k]$, expressing that the output graph has no ports.

Figures 2 and 3 show an example pair of input and output graphs, where the parameters corresponding to $X_1, \ldots, X_3$ are $U_1, \ldots, U_3$, indicated by the green, blue, and orange areas of Figure 2. In these figures, edges labeled "—" and "---" are drawn as solid and broken lines, respectively, and we do not indicate edge directions, for simplicity.

We note here that the quantifier-free operations defined in Courcelle and Engelfriet (2012) satisfy $\ell = 0$, that is, they do not depend on arguments $U_1, \ldots, U_\ell$. However, the extension to $\ell \geq 0$ will turn out to be technically convenient and is mathematically insignificant because one can alternatively consider relational structures with additional unary predicates $U_1, \ldots, U_\ell$ to achieve the same effect in the setting of Courcelle and Engelfriet (2012). A similar remark applies to Theorem 1 below.

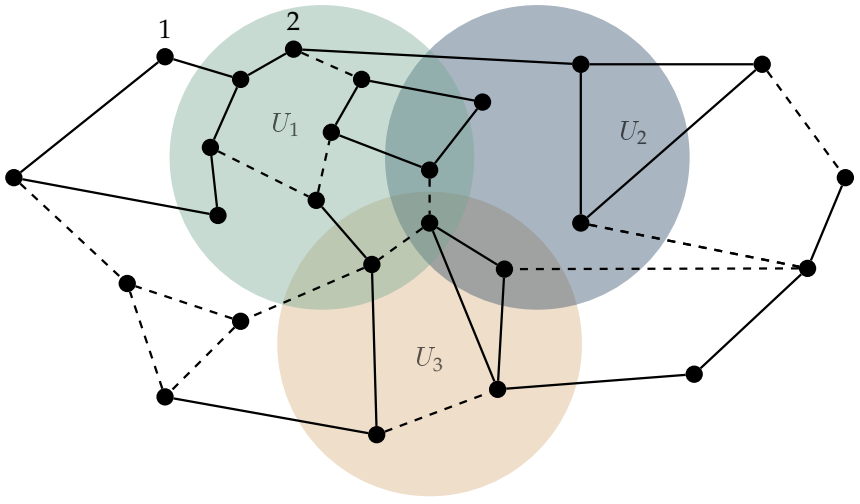



**Figure 2**
An input graph $G$ to the quantifier-free operation $\xi$ in Example 1. The sets of nodes $U_i$, $i \in [3]$, are parameters that (in this particular example) control which nodes are to be included in the output graph $H$ shown in Figure 3. The graph has two ports, indicated by the numbers 1 and 2. Solid and dashed lines represent edges with two different labels; see Example 1 for details.

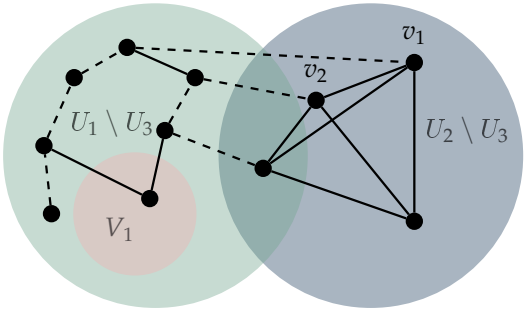



**Figure 3**
The result $H$ of applying the quantifier-free operation $\xi$ in Example 1 to the graph $G$ in Figure 2. Nodes outside $U_1 \cup U_2$ and those in $U_3$ have been discarded, edges involving some node outside $U_2$ have switched their labels, and the nodes belonging to $U_2 \setminus U_3$ now form a clique. The node that previously was the second port has become an ordinary node.

Now we can state the Backwards Translation theorem (which is a weak version of a similar theorem for the much more general *CMSO*-transductions [Courcelle and Engelfriet 2012, Theorem 7.10]). This theorem states that, if we have a quantifier-free operation $\xi$, any property of $\xi(G)$ that can be expressed by some cmso formula $\varphi$ can also be expressed as a property of the original graph $G$ by means of a cmso formula $\varphi'$ (obtained by, intuitively, "backwards translating" $\varphi$ along $\xi$).

**Theorem 1** (Backwards Translation Theorem [cf. Courcelle and Engelfriet 2012, Theorem 5.47])
For every quantifier-free operation $\xi$ in set variables $X_1, \ldots, X_\ell \in \mathcal{V}_1$ and every formula $\varphi(Y_1, \ldots, Y_k, y_1, \ldots, y_{k'}) \in \mathit{CMSO}$, there is a formula $\varphi'(X_1, \ldots, X_\ell, Y_1, \ldots, Y_k, y_1, \ldots, y_{k'}) \in \mathit{CMSO}$ such that the following holds:

> Let $G$ be a graph, $U_1, \ldots, U_\ell \subseteq U_G$, and $H = \xi(G, U_1, \ldots, U_\ell)$. Then we have $G \models \varphi'(U_1, \ldots, U_\ell, V_1, \ldots, V_k, v_1, \ldots, v_{k'})$ if and only if $H \models \varphi(V_1, \ldots, V_k, v_1, \ldots, v_{k'})$, for all $V_1, \ldots, V_k \subseteq V_H$ and $v_1, \ldots, v_{k'} \in V_H$.

**Example 2** (Backward Translation)
Let us return to the quantifier-free operation $\xi$ of Example 1, and the pair of input and output graphs in Figures 2 and 3. Consider now a formula $\varphi$ which, say, states that a pair of nodes $v_1$ and $v_2$ given as parameters to this formula (i.e., $v_1$ and $v_2$ correspond to free variables $y_1$ and $y_2$ in the formula) are such that every node that is not in a given set $V_1$ (corresponding to a free set variable $Y_1$ in the formula) is reachable from at least one of $v_1$ and $v_2$ via an undirected path. Reachability is known to be definable in mso logic (and hence in its extension cmso). The property can, for example, be expressed through the predicate REACHABLE$(x, y)$, which is true if and only if $x = y$ or there exists a set $X$ of nodes such that:

1. $x, y \in X$,
2. every $z \in \{x, y\}$ has an edge to exactly one element in $X \setminus \{z\}$,
3. every $z \in X \setminus \{x, y\}$ has edges to exactly two elements in $X \setminus \{z\}$.

It should be straightforward to verify that each of the properties 1–3 is cmso definable.

Using the predicate REACHABLE, we can define $\varphi(Y_1, y_1, y_2)$ as $\forall y.\psi$, where

$$\psi = \neg(y \in Y_1) \to (\textsc{Reachable}(y, y_1) \vee \textsc{Reachable}(y, y_2))$$

If we have a graph such as $H$ in Figure 3, a set $V_1$ of nodes in $H$, and two further nodes $v_1$ and $v_2$ in that graph, $\varphi(V_1, v_1, v_2)$ expresses the property mentioned above, that is, all nodes except possibly those in $V_1$ are reachable from at least one of $v_1$ and $v_2$. In Figure 3, this is obviously true.

What the backwards translation theorem tells us is that we can construct another formula $\varphi'$ which, if applied to the input graph $G$ to $\xi$, checks whether $\varphi$ would be satisfied in its output graph $H$. In other words, in order to find out whether $\xi$ applied to $G$ will yield a graph in which $\varphi$ is satisfied, we do not need $H$ but can instead evaluate the new formula $\varphi'$ in $G$.

Recall, however, that $H$ depends not only on $G$ but also on the additional set parameters; in this particular example, $H = \xi(G, U_1, U_2, U_3)$. Clearly, this means that the formula $\varphi'$ will have to take $U_1, U_2, U_3$ into account. Hence, $\varphi'$ must be provided with these as auxiliary parameters, in addition to the original parameters of $\varphi$. This is why the backwards translation turns $\varphi(Y_1, y_1, y_2)$ into $\varphi'(X_1, X_2, X_3, Y_1, y_1, y_2)$ instead of the simpler form $\varphi'(Y_1, y_1, y_2)$.

We shall not attempt to construct the concrete formula $\varphi'$ for this example, but it should not be too hard to imagine that a suitable definition of $\varphi'$ may take the form

$$\varphi'(X_1, X_2, X_3, Y_1, y_1, y_2) = \forall y.(((y \in X_1) \vee (y \in X_2)) \wedge \neg(y \in X_3)) \to \psi'$$

for a suitable formula $\psi'$ obtained from $\psi$. Intuitively, this formula $\psi$ needs to "anticipate" the effect the application of $\xi$ to $G$ with the given parameters represented by $X_1, X_2, X_3$ would have and thus, for example, consider edges labeled "---" instead of "—" where $\xi$ would turn the former into the latter, and it would have to consider all pairs of nodes in $X_2 \setminus X_3$ to be connected by "—" edges. This can be achieved by replacing REACHABLE by a suitable predicate REACHABLE′. The details are technically complicated, but the Backwards Translation theorem tells us that it can be done (and the proof by Courcelle and Engelfriet [2012] shows how).

## 3. Graph Extension Grammars

Graph extension grammars generate graphs by repeated application of two types of graph operations. One type of operation takes the disjoint union of a pair of smaller graphs, in doing so concatenating their port sequences. The other type extends an existing graph with additional structure placed "on top" of that graph. The extension operation uses a template graph with designated nodes, so-called docks. The docks are to be attached to the ports of the argument graph, and the ports of the template become the ports of the combined graph. The template also contains a number of *context nodes* that can be duplicated (or **cloned**) and identified with arbitrarily chosen nodes in the argument graph. This is provided that the choice satisfies a given formula whose free variables are the targets of the cloned context nodes.

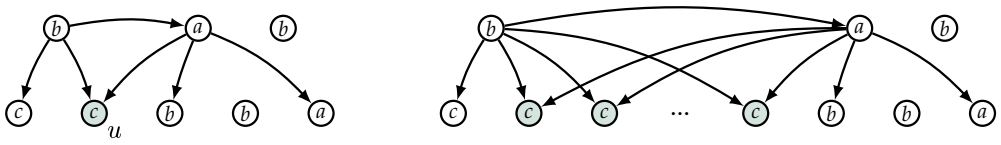



**Figure 4**
The cloning operation takes a graph $G$ and a subset $C$ of its nodes, and replaces each of the nodes in $C$ and their incidents edges by an arbitrary number of copies. The above figure shows $G$ on the left-hand side, with $C = \{u\}$. On the right-hand side, $u$ and attaching structure has been replaced by a number of clones.



To formally define the simpler one of these graph operations, disjoint union, let $k, k' \in \mathbb{N}$. Then $\uplus_{kk'} : \mathbb{G}_k \times \mathbb{G}_{k'} \to \mathbb{G}_{k+k'}$ is defined as follows: for $G \in \mathbb{G}_k$ and $G' \in \mathbb{G}_{k'}$, $\uplus_{kk'}(G, G')$ yields the graph in $\mathbb{G}_{k+k'}$ obtained by making the two graphs disjoint by a suitable renaming of nodes and taking their union. This is defined in the obvious way for the first three components of the argument graphs and concatenates the port sequences of both graphs. Thus, $\uplus_{kk'}$ is not commutative and only defined up to isomorphism. We usually write $G \uplus_{kk'} G'$ instead of $\uplus_{kk'}(G, G')$. To avoid unnecessary technicalities, we shall generally assume that $G$ and $G'$ are disjoint from the start, and that no renaming of nodes takes place. We extend $\uplus_{kk'}$ to an operation $\uplus_{kk'} : \wp(\mathbb{G}_k) \times \wp(\mathbb{G}_{k'}) \to \wp(\mathbb{G}_{k+k'})$ in the usual way: for $\mathcal{G} \subseteq \mathbb{G}_k$ and $\mathcal{G}' \in \mathbb{G}_{k'}$, $\mathcal{G} \uplus_{kk'} \mathcal{G}' = \{G \uplus_{kk'} G' \mid G \in \mathcal{G}, G' \in \mathcal{G}'\}$.

To introduce the second type of graph operation, we first define cloning. The purpose of cloning is to make the expansion operations (to be defined afterwards) more powerful by allowing them to attach edges to an arbitrary number of nodes in the argument graph. Cloning of nodes was originally introduced to formalize the structure of object-oriented programs (Drewes et al. 2010), and later adopted in computational linguistics (Björklund, Drewes, and Ericson 2019).

To define cloning, consider a graph $G = (V, E, lab, port)$ and let $C \subseteq V \setminus [port]$. Then $clone_C(G)$ is the set of all graphs obtained from $G$ by replacing each of the nodes in $C$ and their incident edges by an arbitrary number of copies. Figure 4 illustrates this construction. Formally, $G' = (V', E', lab', port') \in clone_C(G)$ if there is a family $(CL_v)_{v \in V}$ of pairwise disjoint sets $CL_v$ of nodes, such that the following hold:

- $CL_v = \{v\}$ for all $v \in V \setminus C$,
- $V' = \bigcup_{v \in V} CL_v$,
- $E' = \bigcup_{(u,z,v) \in E} CL_u \times \{z\} \times CL_v$, and
- for all $v \in V$ and $v' \in CL_v$, $lab'(v') = lab(v)$.

Note that cloning does *not* rename nodes in $V \setminus C$. In the following, we shall continue to denote by $CL_v$ ($v \in V$) the set of clones of $v$ in a graph belonging to $clone_C(G)$.

We can now define the second type of operation. It is a unary operation called **graph expansion operation** or simply **expansion operation**. We will actually use a restricted form, called *extension operation*, but define the more general expansion operation first. An expansion operation is described by a graph enriched by two additional components: a sequence of nodes called docks and a cmso formula that controls the matching

of context nodes to nodes of the argument graph. A graph expansion operation is given by a tuple

$$\Phi = (V_\Phi, E_\Phi, lab_\Phi, port_\Phi, dock_\Phi, \varphi_\Phi)$$

where

- $\underline{\Phi} = (V_\Phi, E_\Phi, lab_\Phi, port_\Phi)$ and $(V_\Phi, E_\Phi, lab_\Phi, dock_\Phi)$ are graphs, the former being referred to as the *underlying graph* of $\Phi$, and
- $\varphi_\Phi \in CMSO$ is a formula in which every free variable is a set variable $X_v$ for some $v \in V_\Phi \setminus ([port_\Phi] \cup [dock_\Phi])$.

For notational convenience, we define two abbreviations:

- $C_\Phi = V_\Phi \setminus ([port_\Phi] \cup [dock_\Phi])$ is the set of *context nodes* and
- $NEW_\Phi = [port_\Phi] \setminus [dock_\Phi]$ is the set of *new nodes*.

We assume in the following that the set $C_\Phi$ is implicitly ordered, so that we can view it as a sequence whenever convenient. Thus, if $C_\Phi = \{v_1, \ldots, v_k\}$, the last component of $\Phi$ has the form $\varphi_\Phi(X_{v_1} \cdots X_{v_k})$ with one set variable $X_{v_i}$ for every context node $v_i$.

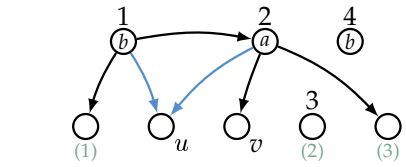
Throughout the rest of this article, we shall continue to denote the components of an expansion operation $\Phi$ by $V_\Phi$, $E_\Phi$, $lab_\Phi$, $port_\Phi$, $dock_\Phi$, $C_\Phi$, and $\varphi_\Phi$.

When applied to a graph, $\Phi$ adds (copies of the) nodes in $NEW_\Phi$ to the graph while those in $[dock_\Phi]$ and $C_\Phi$ are references to ports and nondeterministically chosen nodes in the input graph, respectively.

Formally, the application of $\Phi$ to an argument graph $G = (V, E, lab, port) \in \mathbb{G}_\ell$ is possible if $|dock_\Phi| = \ell$. It yields a graph of type $|port_\Phi|$ by fusing the nodes in $dock_\Phi$ with those in $port$. Moreover, clones of the context nodes are fused with arbitrary nodes in $V$, inheriting the labels from $G$. Thus, the application of the operation to $G$ clones the nodes in $C_\Phi$, fuses those in $dock_\Phi$ with those in $port$, and fuses all nodes in $CL_v$, for each $v \in C_\Phi$, injectively with nodes in $V$, provided that $\varphi_\Phi$ is satisfied under the assignment given by the mapping of cloned context nodes to nodes in $V$. The port sequence of the resulting graph is $port_\Phi$.

Formally, let $|port_\Phi| = k$ and $|dock_\Phi| = \ell$. Then $\Phi$ is interpreted as the nondeterministic operation $\Phi\colon \mathbb{G}_\ell \to \wp(\mathbb{G}_k)$ defined as follows. For a graph $G = (V, E, lab, port) \in \mathbb{G}_\ell$, a graph $H \in \mathbb{G}_k$ is in $\Phi(G)$ if it can be obtained by the following steps:

1. Choose a graph $G' \in clone_{C_\Phi}(\underline{\Phi})$ and a morphing $\mu$ of $G'$ to a graph $(V', E', lab', port')$ such that the following hold:
   (a) $\mu(NEW_\Phi) \cap V = \emptyset$,
   (b) $\mu(dock_\Phi) = port$,
   (c) for all nodes $v \in C_\Phi$, it holds that $\mu(CL_v) \subseteq V$, and
   (d) $G \vDash \varphi_\Phi(\mu(C_\Phi))$ (where we view $C_\Phi$ as a sequence).
2. Define $H = (V \cup V', E \cup E', lab \sqcup lab', port')$.




$$\varphi_\Phi(X_u, X_v) = \forall x.(x \in X_u \to \mathbf{lab}_c(x)) \wedge \exists x.(\mathbf{lab}_b(x) \wedge \forall y.y \in X_v \leftrightarrow x = y))$$

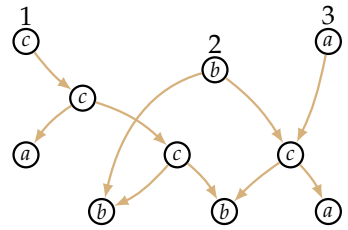



**Figure 5**
The figure on the left shows an extension operation $\Phi$ with four ports (indicated with numbers above the nodes), three docks (indicated with numbers in parentheses below the nodes), and contextual nodes $u$ and $v$. The formula $\varphi_\Phi$ requires that all clones of $u$ be mapped to nodes labeled $c$, and that $v$ is not to be cloned (i.e., cloned exactly once) and mapped to a node labeled $b$. The figure on the right shows a graph $G$ with three ports (again, indicated with numbers above the nodes). Note that $\Phi$ is applicable to $G$ because the number of ports of $G$ coincides with the number of docks of $\Phi$.

We observe that by the definition of the priority union $\sqcup$, the labels of nodes not belonging to $NEW_\Phi$ are disregarded. Hence, the labels of nodes that are fused with context nodes or are ports in $G$ are determined by $lab$.[8] This means that the labels of nodes not in $NEW_\Phi$ can be dropped when specifying $\Phi$, essentially regarding these nodes as unlabeled ones.

We now specialize expansion operations to extension operations by placing conditions on their structure. The intuition is to make sure that graphs are built bottom–up, that is, that $\Phi$ always extends the input graph by placing nodes and edges "on top", with edges being directed downwards, and in such a way that all nodes of the argument graph are reachable from the ports. For this purpose, edges must point from new to "old" nodes, and all nodes in $[dock_\Phi]$ which are not in $[port_\Phi]$ (i.e., intuitively, ports in the argument graph that are "forgotten") must have an incoming edge. Formally,

(R1) $E_\Phi \subseteq NEW_\Phi \times \bar{\mathbb{L}} \times (V_\Phi \setminus NEW_\Phi)$ and

(R2) $[dock_\Phi] \setminus [port_\Phi] \subseteq tar(E_\Phi)$.

Since edges introduced by an extension operation, owing to (R1), can only be directed from new nodes (which by definition of $NEW_\Phi$ must be ports) to nodes in the argument graph, it follows in particular that all graphs constructed from the empty graph with the help of union and extension operations are directed acyclic graphs (DAGs). By a straightforward induction, (R2) ensures that every node in a graph constructed in this way is reachable from a port.

8 This idea has its origins in personal discussions between Frank Drewes and Berthold Hoffmann in 2019.

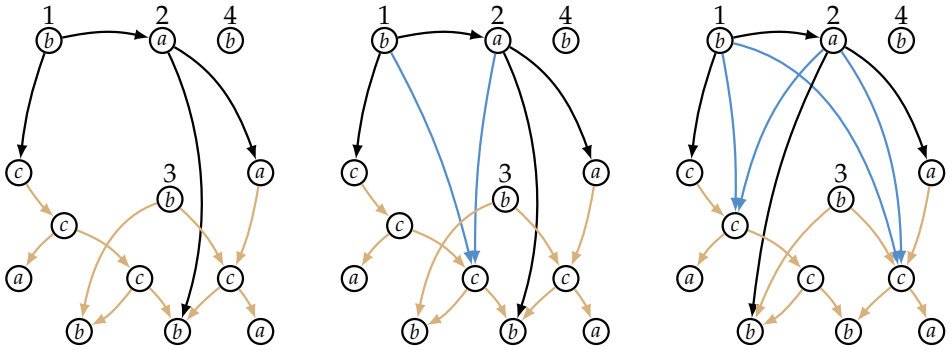


**Figure 6**
Three graphs in $\Phi(G)$ where $\Phi$ and $G$ are as in Figure 5. The differences between the graphs reflect the number of times the context node $u$ has been cloned, and the mapping of (cloned) context nodes to nodes in $G$. The node $u$ has been replicated 0 times to obtain the first graph, once to obtain the second, and twice to obtain the third. In all cases, the clones are mapped to nodes in $G$ which are labeled by $c$, as required by $\varphi_\Phi$. In the first two graphs, $v$ has been identified with one of the $b$-labeled nodes in $G$, and in the right graph with another.

**Example 3**
Figure 5 depicts an extension operation together with a graph to which it can be applied. In Figure 6, we see three different graphs, all resulting from the application of the extension operation in Figure 5 to the graph in Figure 6.

A *graph extension algebra* is a $\Sigma$-algebra $\mathcal{A} = (\wp(\mathbb{G}), (f_\mathcal{A})_{f\in\Sigma})$ where every symbol in $\Sigma$ is interpreted as an extension operation, a union operation, or the set $\{\phi\}$, where $\phi$ is the empty graph $(\emptyset, \emptyset, \emptyset, \varepsilon)$. Note that the operations of the algebra act on sets of graphs rather than on single graphs. This is necessary because of the nondeterministic nature of extension operations. It also takes care of the fact that operations are only defined on graphs of the right type: By convention, the application of an operation to a graph of an inappropriate type yields the empty set of results. This relieves us from having to deal with typed algebras.

**Definition 6** (Graph Extension Grammar)
A (tree-based) **graph extension grammar** is a pair $\Gamma = (g, \mathcal{A})$ where $\mathcal{A}$ is a graph extension $\Sigma$-algebra for some ranked alphabet $\Sigma$ and $g$ is a regular tree grammar over $\Sigma$. The graph language generated by $\Gamma$, denoted by $L(\Gamma)$, is defined as

$$L(\Gamma) = \bigcup_{t\in L(g)} val_\mathcal{A}(t)$$

The *width* $wd(\Gamma)$ of $\Gamma$ is the maximal result type of operations appearing in its operations, that is, $wd(\Gamma) = \max(\{type(f_\mathcal{A}) \mid f \in \Sigma\})$ where $type(\phi) = 0$, $type(\uplus_{k\ell}) = k + \ell$, and $type(\Phi) = |port_\Phi|$ for all extension operations $\Phi$.

For notational simplicity, we shall assume that, in a graph extension grammar as above, $f = f_\mathcal{A}$ for all $f \in \Sigma$, that is, we use the operations themselves as symbols in $\Sigma$.

Before entering on the topic of parsing, let us pause to consider a concrete example in the setting of natural language processing.

**Example 4**
The graph extension grammar in Figure 7 illustrates how the formalism can model semantic representations. For each extension operation $\Phi$, the formula $\varphi_\Phi$ checks that all context nodes of $\Phi$ are mapped to nodes in the argument graph with the same label. Furthermore, $\varphi_\Phi$ checks that no cloning is used or, formally, that every context node is cloned exactly once. Thus, similarly to the condition on $X_v$ in Figure 5, if $C_\Phi = \{v_1, \ldots, v_k\}$, where $lab_\Phi(v_i) = a_i$, then $\varphi_\Phi$ is the formula

$$\bigwedge_{i \in [k]} \exists x.\mathbf{lab}_{a_i}(x) \wedge \forall y.y \in X_{v_i} \leftrightarrow x = y$$

In Figure 7, these formulas are not explicitly shown.

In the context of this example, the concepts `girl` and `boy` represent entities that can act as agents (which verbs cannot), `try` and `persuade` represent verbs that require structural control, and `want` and `believe` represent verbs where this is not needed.

First, we take a top–down perspective to understand the tree generation. The initial nonterminal is $S$. The base case for $S$ is the generation of an extension operation that creates a single node representing a `girl` or a `boy` concept. In all other cases, $S$ generates an extension operation that adds a verb and its outgoing edges, and in which the verb is the single port. The nonterminal $C$ has the same function as $S$ with the following two differences: It has no corresponding base case, and the ports of the extension operations mark both the agent of the verb (designated by an `arg0` edge) and the verb itself. The nonterminal $S'$ can only generate extensions that create `girl` or `boy` nodes. As we shall see, this makes sure that `arg0` edges always point to persons. Finally, there is a pair of nonterminals $U$ and $U'$ that both create union operations, the former with two resulting ports, and the latter with three.

Now, we take a bottom–up perspective to see how the extension operations of a tree are evaluated. Unless the tree consists of a single node, we will have several extensions generated by $S$ and $S'$ that create `girl` and `boy` nodes as the leaves of the tree. In this case, we can apply one or more union operations to concatenate their port sequences and make them visible to further operations. The construction ensures that an `arg0` edge can only have a person, that is, a valid agent, as its target. After the application of a union operation, it is possible to apply any extension that is applicable to $U$ (or $U'$), meaning that none of the non-port nodes are contextual nodes. The resulting graph can have one or two ports—if the graph has one port, the applied extension operation was generated by $S$, and if the graph has two ports, the applied operation was generated by $C$. In other words: $C$ signals that the graph is ready for the addition of a control verb, and $S$ that the graph is a valid semantic graph with one port. When sufficiently many nodes have been generated, it is no longer necessary (but still possible) to use the union operations. Instead, we can add incoming edges to already generated nodes contextually (unless control is explicitly needed, as is the case for control verbs). The generated graphs can therefore contain both structural and non-structural dependencies. See Figure 8 for an example of a tree generated by the grammar in Figure 7, and its evaluation into a semantic graph.

While cloning is explicitly disabled in the extension operations above (by requiring that context nodes are cloned exactly once), in general cloning may play a central role

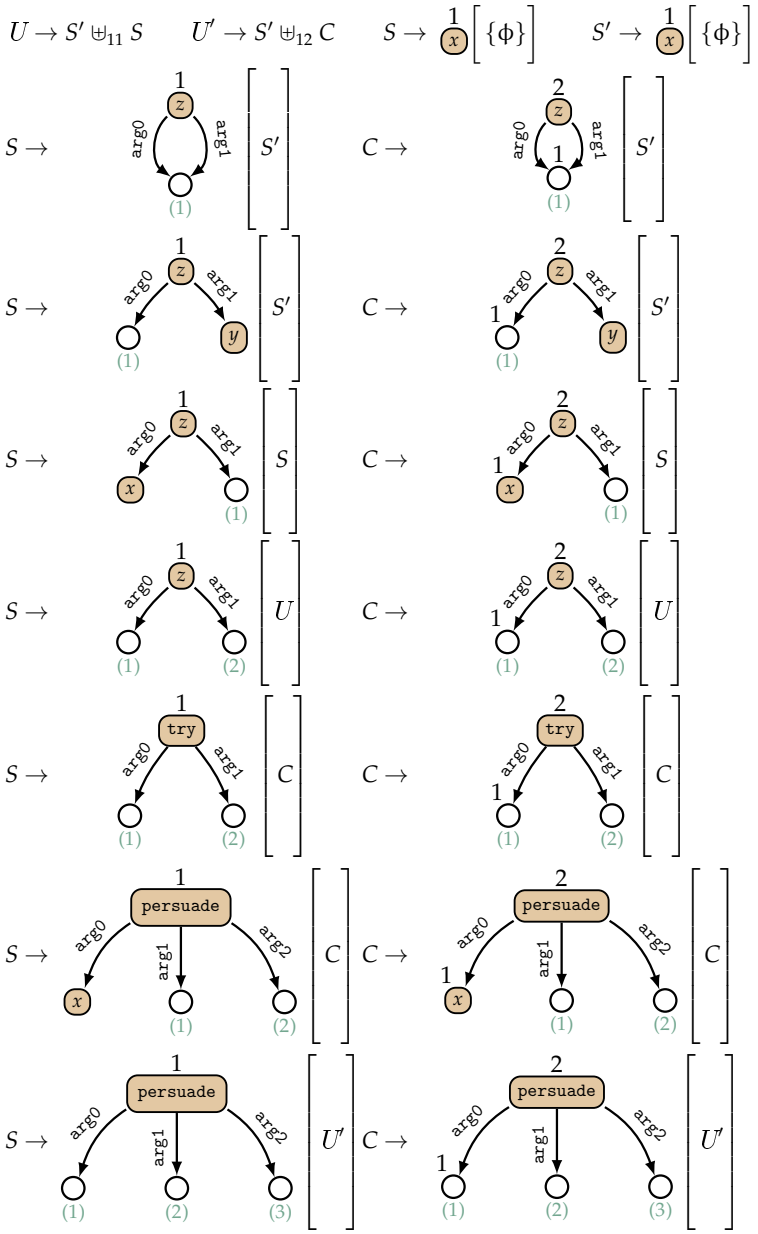



**Figure 7**
A (graphical representation of a) label-matching graph extension grammar demonstrating how to generate semantic graphs akin to AMR. Instead of repeating productions that only differ with respect to node labels, we use label variables, where $x \in \{\texttt{girl}, \texttt{boy}\}$, $y \in \{\texttt{girl}, \texttt{boy}, \texttt{want}, \texttt{believe}, \texttt{try}, \texttt{persuade}\}$, and $z \in \{\texttt{want}, \texttt{believe}\}$. The nonterminal $C$ generates graphs of type 2 whose ports are to be embedded in a control verb construction.

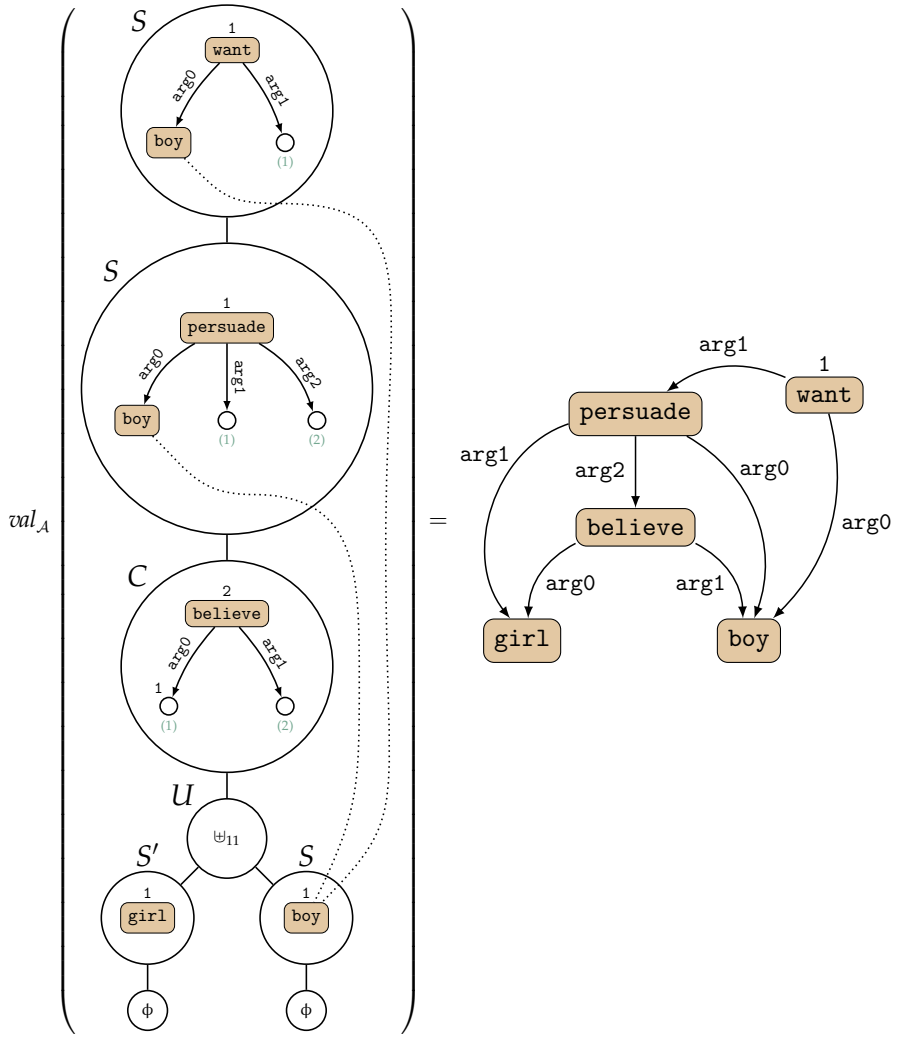
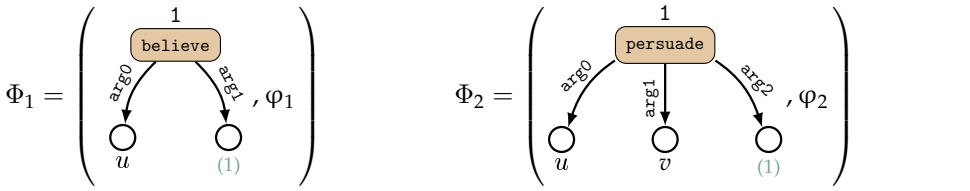



**Figure 8**
A graphical representation of a tree generated by the regular tree grammar in Figure 7 and the graph resulting from its evaluation. The nonterminal generating each subtree is shown next to the respective circle. Dotted lines indicate the identification of context nodes with nodes in argument graphs. In this example, there is only one possible candidate for each context node, hence the resulting graph is unique up to isomorphism.

for NLP applications, because it enables concepts to refer to an unbounded number of "arguments". (Note that in the edge-agnostic case, i.e., if the formulas $\varphi_\Phi$ of the extension operations are not allowed to make us of the predicates $\mathbf{edg}_a$, the mapping of cloned nodes to nodes in the actual graph is made solely based on labels and ports. Similarly, the decision about how *many* clones to create cannot depend on the presence or absence of edges. We may, for instance, say "create at least (or at most) $k$ clones if there is both an $a$- and a $b$-labeled node", but not "create a clone for every node which is the target of a $z$-labeled edge and map it to that node".)




$\varphi_1 = \exists x.(x \in X_u) \wedge \forall x.(x \in X_u \rightarrow \mathbf{lab}_{\texttt{girl}}(x) \vee \mathbf{lab}_{\texttt{boy}}(x))$

$\varphi_2 = \varphi_1(X_u) \wedge \varphi_1(X_v) \wedge \forall x, y.(\mathbf{port}_1(x) \rightarrow (\mathbf{edg}_{\texttt{arg0}}(x, y) \leftrightarrow y \in X_v))$

**Figure 9**
Extension operations such that $\Phi_2(\Phi_1(G))$ express situations in which a group of persons (boys and girls) persuades another group of persons to believe in something or someone.



Cloning can be used if we, for example, want to express the situation that an entire group of people is persuaded by another group to believe something or someone, thus using several agents as a target of any of the argument edges. For this, we can use operations such as those in Figure 9 (where an extension operation $\Phi$ is depicted as a pair consisting of the graph $\underline{\Phi}$ with docks indicated as before, and the formula $\varphi_\Phi$). Here, the formula $\varphi_2$ plays a vital role. The formula $\varphi_1$ only requires that the believers (targets of arg0-edges) are persons (and that there is at least one). While $\varphi_2$ incorporates the same requirement for both persuaders and persuaded persons, it additionally requires that the persuaded ones (represented by nodes in $X_v$) are exactly those who are also reachable via arg0-edges from the port of the argument graph, that is, the believers. It may be instructive to note that the corresponding structural requirement was expressed by "remembering" nodes as ports in the rules of Figure 7. This is no longer possible here (recall that $\mathbf{port}_1(x)$ picks the first port), because the set of nodes to be remembered is not a priori bounded in size. Hence, we have to express the desired coordination by means of the $\mathbf{edg}_{\texttt{arg0}}$-predicate. With this addition, the example is not any longer edge agnostic in the sense to be defined in Definition 7, and thus the result we are going to prove in Section 5 does not apply to it anymore. However, the property expressed by $\varphi_2$ is a local one in the sense to be defined in Section 6, and hence our main result (Theorem 5) does indeed cover it.

Readers familiar with HRGs or Courcelle's hyperedge replacement algebras (Courcelle 1991) may have noticed that productions using union and extension operations (disregarding context nodes and the effect of the component $\varphi_\Phi$) correspond to hyperedge replacement productions of two types:

(a) $A \rightarrow B \uplus_{k\ell} C$ corresponds to a production with two hyperedges in the right-hand side, labeled $B$ and $C$, where the first one is attached to $k$ of the nodes to which the nonterminal in the left-hand side (which is labeled $A$) is attached, and the second is attached to the remaining $\ell$ nodes of the left-hand side. In particular, there are no further nodes in the right-hand side.

(b) $A \rightarrow \Phi[B]$ corresponds to a hyperedge replacement production with a single nonterminal hyperedge labeled $B$ in its right-hand side. Such productions are the ones responsible for actually generating new terminal items (nodes and edges).

Generalized extension operations of arity greater than one can be constructed by defining so-called derived operations through composition of suitable extension and union operations. Given a set of such derived operations, the tree-to-tree mapping that replaces each occurrence of a derived operation by its definition is a tree homomorphism. As regular tree languages are closed under tree homomorphisms, this shows that the restriction to binary unions and unary extensions is no limitation (in contrast to requirements (R1) and (R2), which help reduce the complexity of parsing).

We also remark here that it is not a restriction that all new nodes introduced in an extension operation are ports. This is because we can add a non-port to a graph $G$ by evaluating $\Phi(\Phi'(G))$ where $\Phi'$ introduces the desired node as a port, say port $i$, and $\Phi$ "forgets" port $i$, that is, $dock_\Phi(i) \notin [port_\Phi]$. (Note, however, that (R2) then requires $dock_\Phi(i)$ to have an incoming edge from one of the ports of $\Phi$, making sure that the node introduced in this way is reachable from a port in $\Phi(\Phi'(G))$.)

## 4. NP-Completeness

Before turning to graph extension grammars for which parsing can be implemented to run in polynomial time, we confirm in this section that restrictions are required to accomplish this (unless P = NP), as the problem is NP-complete in general. Readers who are not interested in the proof may either skip this section or read it for the sake of seeing another example of a graph extension grammar.

**Theorem 2**
For all graph extension grammars $\Gamma$, it holds that $L(\Gamma)$ is in NP. Furthermore, there exist graph extension grammars $\Gamma$ such that $L(\Gamma)$ is NP-complete.

*Proof.* As an immediate consequence of the parsing algorithm to be presented in Section 5, it holds that $L(\Gamma) \in$ NP for all graph extension grammars $\Gamma$. This is because nondeterminism can be used to simply "guess" an appropriate matching in line 21 of Algorithm 1. The verification that such a guessed matching is indeed one can easily be implemented to run in polynomial time.

It remains to find a graph extension grammar that generates an NP-complete graph language. We do this by presenting a graph extension grammar $\Gamma$ whose generated graphs represent satisfiable formulas in propositional logic. The grammar consists of three parts. Taking a bottom–up view, these three parts serve the following purposes:

1. The first part, corresponding to the rules applied furthest down in the derivation tree, generates trees with node labels in $\{\neg, \vee, \wedge, var\}$, where every node labeled $\neg$ has one child, those labeled $\vee$ and $\wedge$ have two children, and nodes labeled *var* are leaves. These trees thus represent formulas in the usual way, where every node labeled *var* stands for an occurrence of an unnamed variable.
2. The second part introduces a chain of $=$- nodes on top of the root of the formula. From each of these nodes, any number of edges will point to some of the (nodes representing) variable occurrences. Two occurrences of variables will be considered to be occurrences of the same variable if and only if they are both targets of edges originating from the same $=$-labeled node.

**Algorithm 1** Parse a graph $G$ with respect to a graph extension grammar $\Gamma = (g, \mathcal{A})$

```
1: procedure PARSE(G = (V, E, lab, port))
2:     if G is acyclic and G = ▽G port then
3:         return PARSE_REC(S, port)          ▷ Invoke recursive part with initial nonterminal S
4:     else
5:         return false
6:     end if
7: end procedure
8: procedure PARSE_REC(A ∈ N, p ∈ V^⊛)
9:     if result[A, p] undefined then                          ▷ No memoised result available
10:        result[A, p] ← false                                ▷ For the moment, memoise failure
11:        if p = ε and (A → φ) ∈ P then
12:            return result[A, p] ← true                      ▷ Memoise success
13:        else
14:            for all (A → B_1 ⊎_{k_1 k_2} B_2) ∈ P do
15:                let p = p_1 p_2 where |p_i| = k_i for i = 1, 2
16:                if p_1^▽ ∩ p_2^▽ = ∅ and PARSE_REC(B_1, p_1) and PARSE_REC(B_2, p_2) then
17:                    return result[A, p] ← true              ▷ Memoise success
18:                end if
19:            end for
20:            for all (A → Φ[B]) ∈ P do
21:                for all matchings m of Φ to ▽G p do
22:                    if PARSE_REC(B, m(dock_Φ)) then
23:                        return result[A, p] ← true          ▷ Memoise success
24:                    end if
25:                end for
26:            end for
27:        end if
28:    end if
29:    return result[A, p]
30: end procedure
```

3. The third part of the grammar, corresponding to the topmost section of the derivation tree, consists of only one rule that "guesses" a satisfying truth assignment and uses its cmso formula to check that this truth assignment is indeed satisfying.

To simplify the grammar and make the rules more readable, we occasionally use rules whose right-hand sides are arbitrary trees over the given operations and nonterminals, rather than sticking to rules of the form $A \to f[A_1, \ldots, A_k]$ as in Definition 1. In a straightforward way, these rules can be decomposed into rules of the form $A \to f[A_1, \ldots, A_k]$ by introducing additional nonterminals.

The first part of the grammar is shown in Figure 10.

The rules of the second part of the grammar are shown in Figure 11. The cmso formula makes sure that edges from the generated nodes all point to variable occurrences.

Finally, the rule that ensures that the generated formula is satisfiable is shown in Figure 12. Its left-hand side $S$ is the initial nonterminal of the grammar. The conjuncts of the formula express that every node representing a subformula has precisely one eq-edge pointing to it (line 1, where we make use of the customary abbreviation $\exists^!$ for 'there exists exactly one'), every node representing a variable or logical operator is assigned either *true* or *false* (line 2), different occurrences of the same variable are

$$F \to \Phi_{\neg}[F], \quad F \to \Phi_{bin}[F \uplus_1 1F] \text{ for } bin \in \{\vee, \wedge\}, \quad F \to \Phi_{var}[\phi]$$

$$\Phi_{\neg} = \left( \begin{array}{c} 1 \\ \neg \\ \texttt{arg} \\ (1) \end{array} , \; true \right) \quad \Phi_{bin} = \left( \begin{array}{c} 1 \\ bin \\ \texttt{arg} \quad \texttt{arg} \\ (1) \quad (2) \end{array} , \; true \right) \quad \Phi_{var} = \left( \begin{array}{c} 1 \\ var \end{array} , \; true \right)$$

**Figure 10**
Rules that generate, from the nonterminal $F$, trees that represent propositional formulas over anonymous occurrences of variables.

$$E \to \Phi_{=}[E], \quad E \to \Phi_{=}[F]$$

$$\Phi_{=} = \left( \begin{array}{c} 1 \\ = \\ \texttt{eq} \quad \texttt{chain} \\ u \quad (1) \end{array} , \; \forall x.(x \in X_u \to \mathbf{lab}_{var}(x)) \right)$$

**Figure 11**
Rules that generate a chain or $=$-labeled nodes, each of which has edges to an arbitrary number of variable occurrences.

$$S \to \Phi_{\text{sat}}[E]$$

$$\Phi_{\text{sat}} = \left( \begin{array}{c} 1 \\ \texttt{root} \\ \texttt{chain} \\ T \quad (1) \quad F \end{array} , \; \begin{array}{l} \forall x.(\mathbf{lab}_{var}(x) \to \exists^{!} y.\mathbf{edg}_{\texttt{eq}}(x,y)) \wedge \\ \forall x.(\neg(\mathbf{lab}_{=}(x) \vee \mathbf{lab}_{\texttt{root}}(x)) \to (x \in X_F \leftrightarrow x \notin X_T)) \wedge \\ \forall x,y,z.(\mathbf{edg}_{\texttt{eq}}(x,y) \wedge \mathbf{edg}_{\texttt{eq}}(x,z) \to (y \in X_T \leftrightarrow z \in X_T)) \wedge \\ \forall x.(\mathbf{lab}_{\neg}(x) \to (x \in X_T \leftrightarrow \forall y.(\mathbf{edg}_{\texttt{arg}}(x,y) \to y \in X_F)) \wedge \\ \forall x.(\mathbf{lab}_{\wedge}(x) \to (x \in X_T \leftrightarrow \forall y.(\mathbf{edg}_{\texttt{arg}}(x,y) \to y \in X_T)) \wedge \\ \forall x.(\mathbf{lab}_{\vee}(x) \to (x \in X_T \leftrightarrow \exists y.(\mathbf{edg}_{\texttt{arg}}(x,y) \wedge y \in X_T)) \wedge \\ \exists x,y.(\bigvee_{\ell \in \{\wedge,\vee,\neg\}} \mathbf{lab}_{\ell}(y) \wedge \mathbf{edg}_{\texttt{chain}}(x,y) \wedge y \in X_T \end{array} \right)$$

**Figure 12**
The rule that checks the existence of a satisfying assignment of truth values (represented by the clones of the nodes $T$ and $F$).

assigned the same truth value (line 3), the truth assignment is compatible with the definition of the logical operators (lines 4–6), and the truth value assigned to the root node of the formula is *true* (line 7). It should be clear that the formula represented by a graph $G$ that can be generated from $E$ is satisfiable if and only if $\Phi_{\text{sat}}(G) \neq \emptyset$, which is the case if and only if $L(\Gamma)$ contains the graph $G'$ obtained from $G$ by adding a node labeled `root` with a `chain`-labeled edge pointing to the port of $G$ and making that new node the (unique) port of $G'$. (Note that this is the effect of applying $\Phi_{\text{sat}}$ to $G$

if the formula is satisfied for some choice of $X_T$ and $X_F$.) Moreover, given a propositional formula it is easy to construct the corresponding graph $G'$, that is, there is a logspace reduction of the NP-complete satisfiability problem for propositional formulas to $L(\Gamma)$. □

We note that the validity of the preceding proof does not depend on the use of isolated context nodes in $\Phi_{\text{sat}}$: The proof remains valid if we add (equally labeled) edges from the port to each of these context nodes. Then every node corresponding to a subformula would receive exactly one incoming edge from the port, regardless of the truth assignment hidden in the choice of $X_T$ and $X_F$, and adding those edges to the output graph $G'$ of the reduction would again result in a logspace reduction to $L(\Gamma)$.

## 5. Parsing for Edge-Agnostic Graph Extension Grammars

We now provide a blueprint of a parsing algorithm for graph extension grammars, which we afterwards instantiate to a polynomial time parsing algorithm for a special case of graph extension grammars, the so-called edge-agnostic ones. In the next section, we will discuss how this restriction can partially be lifted.

Throughout this section, let $\Gamma = (g, \mathcal{A})$ be a graph extension grammar, where $g = (N, \Sigma, P, S)$ and $\mathcal{A}$ is a graph extension $\Sigma$-algebra.

The goal is to decide the membership problem for $L(\Gamma)$ and, in the positive case, produce a corresponding derivation tree $t \in L(g)$ such that the given graph $G$ is in $val_{\mathcal{A}}(t)$. In fact, by the recursive structure of the proposed algorithm, it will be obvious how to obtain $t$. Hence, we focus on the membership problem, which is formally defined as follows:

Input: A graph $G$
Question: Does it hold that $G \in L(\Gamma)$?

By definition, extension operations keep the identities of nodes in the input graph unchanged, whereas new nodes added to the graph may be given arbitrary identities (as long as clashes with nodes in the input graph are avoided). For the union operation, it holds that renaming of nodes is necessary only if the node sets of the two argument graphs intersect. As a consequence, when evaluating a tree $t$, we can without loss of generality assume that operations never change node identities of argument graphs. In other words, if $G \in val_{\mathcal{A}}(t)$, then for every $\alpha \in addr(t)$, there is a (concrete) graph $G_\alpha \in val_{\mathcal{A}}(t/\alpha)$ such that

- if $t/\alpha = \Phi[t/\alpha 1]$, then $G_\alpha \in \Phi(G_{\alpha 1})$ and
- if $t/\alpha = t/\alpha 1 \uplus_{k\ell} t/\alpha 2$, we have $G_\alpha = (V_1 \cup V_2, E_1 \cup E_2, lab_1 \sqcup lab_2, port_1, port_2)$ where $G_{\alpha i} = (V_i, E_i, lab_i, port_i)$ for $i = 1, 2$,

and $G_\varepsilon = G$.

Thus, for every $\alpha \in addr(t)$, the graph $G_\alpha$ is a subgraph of $G$. More precisely, if $G = (V, E, lab, port)$ and $G_\alpha = (V', E', lab', port')$ then $V' \subseteq V$, $E' \subseteq E$, and $lab' = lab|_{V'}$. Note that, while the graphs $G_\alpha$ are usually not uniquely determined by $G$ and $t$, the important fact is that they do exist if and only if $G \in val_{\mathcal{A}}(t)$. We say in the following that the family $(G_\alpha)_{\alpha \in addr(t)}$ is a *concrete evaluation of t into G*. Hence, the membership problem amounts to deciding, for the input graph $G$, whether there is a tree $t \in L(g)$ that permits a concrete evaluation into $G$.

The following lemma, which forms the basis of our parsing algorithm, shows that $G_\alpha = (V', E', lab', port')$ is determined by $port'$ alone, as it is the subgraph of $G$ induced by the nodes that are reachable from $port'$. More precisely, consider a sequence $\boldsymbol{p} \in V^{\circledast}$ of nodes of $G$ and let $\boldsymbol{p}^{\triangledown}$ be the set of nodes reachable in $G$ by directed paths from (any of the nodes in) $\boldsymbol{p}$.[9] Now, we define $G \triangledown \boldsymbol{p} = (\boldsymbol{p}^{\triangledown}, E|_{\boldsymbol{p}^{\triangledown} \times \mathbb{L} \times \boldsymbol{p}^{\triangledown}}, lab|_{\boldsymbol{p}^{\triangledown}}, \boldsymbol{p})$.

**Lemma 1**
If $(G_\alpha)_{\alpha \in addr(t)}$ is a concrete evaluation of a tree $t \in T_\Sigma$ into a graph $G$, then $G_\alpha = G \triangledown port_{G_\alpha}$ for all $\alpha \in addr(t)$.

*Proof.* Let $G_\alpha = (V_\alpha, E_\alpha, lab_\alpha, port_\alpha)$ for every $\alpha \in addr(t)$. We first show the following claim:

*Claim 1.* We have $\{e \in E \mid src(e) \in V_\alpha\} \subseteq E_\alpha$ and $port_\alpha^{\triangledown} \subseteq V_\alpha$.

We prove Claim 1 by induction on the length of $\alpha$. (Thus, the induction proceeds top-down rather than bottom-up.) For $\alpha = \varepsilon$, the statement holds trivially. Now, assume that it holds for some $\alpha \in addr(t)$. Since the empty graph $\phi$ has neither ports nor edges, two relevant cases remain.

*Case 1.* $G_\alpha$ is of the form $G_{\alpha 1} \uplus_{k\ell} G_{\alpha 2}$.

Let $\{i, j\} = [2]$. We have to show that the statement holds for $G_{\alpha i}$. Thus, assume that $src(e) \in V_{\alpha i}$ and $v \in port_{\alpha i}^{\triangledown}$. Then $src(e) \in V_\alpha$ and thus, by the induction hypothesis, $e \in E_\alpha$. By the definition of $\uplus_{k\ell}$ this means that $e \in E_{\alpha i}$. Furthermore, $v \in port_{\alpha i}^{\triangledown}$ implies $v \in port_\alpha^{\triangledown}$ and thus, again by the induction hypothesis, $v \in V_\alpha$. Hence, it remains to argue that no node in $G_{\alpha j}$ can be reached from $port_{\alpha i}$. This follows readily from the just established fact that there is no edge $e' \in V \setminus V_{\alpha i}$ for which $src(e') \in E_{\alpha i}$ (together with the fact that $V_{\alpha i} \cap V_{\alpha j} = \emptyset$).

*Case 2.* $G_\alpha$ is of the form $\Phi(G_{\alpha 1})$ for an extension operation $\Phi$.

To show that the statement holds for $G_{\alpha 1}$, let $G_\alpha = (V_{\alpha 1} \cup V', E_{\alpha 1} \cup E', lab_{\alpha 1} \sqcup lab', port')$, where $(V', E', lab', port')$ is obtained from $\underline{\Phi}$ as in the definition of $\Phi(G_{\alpha 1})$. By the induction hypothesis, $src(e) \in V_\alpha$ implies $e \in E_\alpha$. Thus, $src(e) \in V_{\alpha 1}$ implies $e \in E_{\alpha 1}$ unless $e \in E'$. However, by requirement (R1), all $e \in E'$ satisfy $src(e) \in [port'] \setminus [port]$, which is equivalent to $src(e) \notin V_{\alpha 1}$ because $V_\alpha \setminus V_{\alpha 1} = [port'] \setminus [port]$. From this it follows that $e \in E_{\alpha 1}$.

9 We will only use the notation $\boldsymbol{p}^{\triangledown}$ when the graph $G$ being referred to is clear from the context.

Now, let $v \in port^{\triangledown}_{\alpha 1}$. If $v \notin V_{\alpha 1}$, consider the first edge $e$ on a path from a node in $[port_{\alpha 1}]$ to $v$ such that $tar(e) \notin V_{\alpha 1}$. Then $src(e) \in V_{\alpha 1}$ but $e \notin E_{\alpha 1}$, contradicting the previously established fact that $src(e) \in V_{\alpha 1}$ implies $e \notin E_{\alpha 1}$.

This finishes the proof of Claim 1. Next, we prove the converse inclusion of the second part of Claim 1:

*Claim 2.* $V_\alpha \subseteq port^{\triangledown}_{\alpha}$.

This time, we proceed bottom–up, by induction on the size of $t/\alpha$. The statement is trivially true for $G_\alpha = \phi$. Thus, as before, there are two cases to distinguish.

*Case 1.* $G_\alpha$ is of the form $G_{\alpha 1} \uplus_{k\ell} G_{\alpha 2}$.

By the induction hypothesis, Claim 2 holds for $G_{\alpha 1}$ and $G_{\alpha 2}$. Consequently, $V_\alpha = V_{\alpha 1} \cup V_{\alpha 2} \subseteq port^{\triangledown}_{\alpha 1} \cup port^{\triangledown}_{\alpha 2} = port^{\triangledown}_{\alpha}$.

*Case 2.* $G_\alpha$ is of the form $\Phi(G_{\alpha 1})$ for an extension operation $\Phi$.

Again, let $G_\alpha = (V_{\alpha 1} \cup V', E_{\alpha 1} \cup E', lab_{\alpha 1} \sqcup lab', port')$, where $(V', E', lab', port')$ is obtained from $\underline{\Phi}$ as in the definition of $\Phi(G_{\alpha 1})$. (In particular, $port' = port_\alpha$.) By the induction hypothesis, $V_{\alpha 1} \subseteq port^{\triangledown}_{\alpha 1}$. Moreover, by requirement (R2), for every node $v \in [port_{\alpha 1}]$ it either holds that $v \in [port_\alpha]$, or there are $u \in [port_\alpha]$ and $e \in E'$ with $src(e) = u$ and $tar(e) = v$. Hence, $[port_{\alpha 1}] \subseteq port^{\triangledown}_{\alpha}$ and thus $V_{\alpha 1} \subseteq port^{\triangledown}_{\alpha 1} \subseteq port^{\triangledown}_{\alpha}$. Since, furthermore, $V' \setminus V_{\alpha 1} \subseteq [port_\alpha]$, this shows that $V_\alpha \subseteq [port_\alpha]$, and thus $V_\alpha \subseteq port^{\triangledown}_{\alpha}$.

Together, Claims 1 and 2 state that $V_\alpha$ contains exactly the nodes reachable from $port_\alpha$ (by Claim 2 and the second part of Claim 1), and also all edges originating at those nodes (by the first part of Claim 1). This finishes the proof of the lemma. □

We note the following immediate consequence of Lemma 1:

**Corollary 1**
Let $(G_\alpha)_{\alpha \in addr(t)}$ be a concrete evaluation of a tree $t \in T_\Sigma$ into a graph $G$. If $G_\alpha = \triangledown G\varepsilon$ for some $\alpha \in addr(t)$, then $G_\alpha = \phi$.

In Lemma 1, we find the beginnings of a recursive parsing algorithm: In the following, consider an input graph $G = (V, E, lab, port)$, a nonterminal $A \in N$, and a sequence $\boldsymbol{p} \in V^{\circledast}$. To decide whether $G \triangledown \boldsymbol{p} \in val_{\mathcal{A}}(L_A(g))$ (and thus whether $G \in L(\Gamma)$ if $A = S$ and $\boldsymbol{p} = port_G$), we need to consider three cases, the first two of which are straightforward.

*Case 1.* If $\boldsymbol{p} = \varepsilon$, then we have $G \triangledown \boldsymbol{p} = \phi \in val_{\mathcal{A}}(L_A(g))$ if and only if the production $A \to \phi$ is in $P$.

Otherwise, we need to check each production with the left-hand side $A$ according to Cases 2 and 3, as follows:

*Case 2.* If the production is of the form $A \to B_1 \uplus_{k_1 k_2} B_2$, let $\boldsymbol{p} = \boldsymbol{p}_1\boldsymbol{p}_2$ where $|\boldsymbol{p}_i| = k_i$ for $i = 1, 2$. If $\boldsymbol{p}_1^{\triangledown} \cap \boldsymbol{p}_2^{\triangledown} \neq \emptyset$, then $G \triangledown \boldsymbol{p} \neq G \triangledown \boldsymbol{p}_1 \uplus_{k_1 k_2} G \triangledown \boldsymbol{p}_2$. If $\boldsymbol{p}_1^{\triangledown} \cap \boldsymbol{p}_2^{\triangledown} = \emptyset$, we recursively

need to determine whether $G \triangledown \boldsymbol{p}_i \in \mathit{val}_{\mathcal{A}}(L_{B_i}(g))$ for $i = 1, 2$ because in that case $G \triangledown \boldsymbol{p} = G \triangledown \boldsymbol{p}_1 \uplus_{k_1 k_2} G \triangledown \boldsymbol{p}_2$.

*Case 3.* If the production is of the form $A \to \Phi(B)$, assume first for simplicity that no cloning takes place (i.e., every node in $C_\Phi$ is cloned precisely once). The basic intuition is that we need to check all structure-preserving mappings of $\underline{\Phi}$ to $G \triangledown \boldsymbol{p}$ that map $\mathit{port}_\Phi$ to $p$. Such a mapping determines in particular the image of $[\mathit{dock}_\Phi]$ in $G \triangledown \boldsymbol{p}$, say $\boldsymbol{p}_1$, and similarly it defines the images $v_1, \ldots, v_m \in V$ of the context nodes $u_1, \ldots, u_m$ of $\Phi$. Hence, we now have to check whether $G \triangledown \boldsymbol{p}_1 \vDash \varphi_\Phi(\{v_1\}, \ldots, \{v_m\})$ and, recursively, whether $G \triangledown \boldsymbol{p}_1 \in \mathit{val}_{\mathcal{A}}(L_B(g))$.

However, this disregards the fact that nodes in $C_\Phi$ can be cloned. To deal with this additional difficulty, we define the notion of a matching of $\Phi$ to $G \triangledown \boldsymbol{p}$. Then, based on Lemma 1, the parsing strategy in this third case is to check whether there exists a matching $m$ of $\Phi$ to $G \triangledown \boldsymbol{p}$ such that, recursively, $G \triangledown m(\mathit{dock}_\Phi) \in \mathit{val}_{\mathcal{A}}(L_B(g))$. Thus, for every such sequence $m(\mathit{dock}_\Phi)$, the algorithm recursively invokes a procedure PARSE_REC$(B, m(\mathit{dock}_\Phi))$ (which is outlined in the upcoming Algorithm 1) and returns *yes* if one of the recursive calls does, and *no* otherwise.

To define the notion of matchings, consider a graph extension operation $\Phi$ with $|\mathit{port}_\Phi| = k$, as well as a directed acyclic graph $H$ with $\mathit{type}(H) = k$ (i.e., here $H$ takes the rôle of $G \triangledown \boldsymbol{p}$ above). We need to determine all possible $\boldsymbol{p}_1 \in V_H^{\circledast}$ such that $H \in \Phi(H \triangledown \boldsymbol{p}_1)$. To see how this can be done, let $V_{NEW} = \{\mathit{port}_H(i) \mid i \in [k] \text{ and } \mathit{port}_\Phi(i) \notin [\mathit{dock}_\Phi]\}$. In other words, $V_{NEW}$ is the set of nodes of $H$ which are images of nodes in $NEW_\Phi$. Recall that, by (R1), the nodes in $NEW_\Phi$ are the only nodes in $\Phi$ that may have edges to other nodes in $\Phi$. Thus, the edge set $E_{NEW} = \{e \in E_H \mid \mathit{src}(e) \in V_{NEW}\}$ contains precisely the edges of $H$ which are images of edges of $\Phi$. Hence, we have to find a suitable mapping of the nodes in $V_\Phi$ to subsets of $V_1 = \boldsymbol{p}_1^{\triangledown}$. We say that a function $m \colon V_\Phi \to \wp(V_1)$ is a *matching of* $\Phi$ *to* $H$ if the following conditions are satisfied:

(M1) The restriction of $H$ to $V_1$ and $E_{NEW}$ is a morph of a graph in $\mathit{clone}_{C_\Phi}(\underline{\Phi})$ by some morphing $\mu$ which satisfies the conditions $|\mu(CL_u)| = 1$ for all $u \in \mathit{dock}_\Phi$ and $\mu(CL_v) = m(v)$ for all $v \in V_\Phi$.

(In particular, $|m(v)| = 1$ for all $v \in [\mathit{port}_\Phi] \cup [\mathit{dock}_\Phi]$. Therefore, we shall in the following view the restriction of $m$ to $[\mathit{port}_\Phi] \cup [\mathit{dock}_\Phi]$ as a function from nodes to nodes rather than to sets of nodes, see, for example, (M2) and the left-hand side of (M3) below.)

(M2) In $H$, every node in $V_H \setminus [\mathit{port}_H]$ is reachable from (a node in) $m(\mathit{dock}_\Phi)$.

(M3) $H \triangledown m(\mathit{dock}_\Phi) \vDash \varphi_\Phi(m(u_1), \ldots, m(u_n))$, where $\{u_1, \ldots, u_n\} = C_\Phi$.

The pseudocode of the algorithm is shown in Algorithm 1. In the code, we use the statement "**return** $\mathit{result}[A, \boldsymbol{p}] \leftarrow \mathit{true}$" to denote memoisation: the variable $\mathit{result}[A, \boldsymbol{p}]$ gets the Boolean value *true* assigned to it and then that same value is returned by the procedure. Obviously, the condition in line 21 of the algorithm needs to be made more concrete, that is, we have to find (efficient) ways to check whether $m$ exists. However, let us first postpone this question and show that the algorithm is correct.

**Theorem 3**
Algorithm 1 decides whether the input graph $G$ is an element of $L(\Gamma)$.

*Proof.* By Lemma 1, it suffices to show that PARSE_REC$(A, \boldsymbol{p})$, for all $\boldsymbol{p} \in V^{\circledast}$, returns *true* if $A \to_g^* t$ for a tree $t$ such that $val_{\mathcal{A}}(t) = G \triangledown \boldsymbol{p}$, and *false* otherwise. By line 10, termination is guaranteed since there are only finitely many pairs $(A, \boldsymbol{p})$. Hence, it remains to be shown that PARSE_REC$(A, \boldsymbol{p})$ returns *true* if and only if $A \to_g^* t$ for a tree $t$ such that $val_{\mathcal{A}}(t) = G \triangledown \boldsymbol{p}$.

We first show that PARSE_REC$(A, \boldsymbol{p})$ returns *true* if there exists a tree $t \in L_A(g)$ such that $G \triangledown \boldsymbol{p} \in val_{\mathcal{A}}(t)$. We prove this by induction on the smallest tree $t \in L_A(g)$ such that $G \triangledown \boldsymbol{p} \in val_{\mathcal{A}}(t)$. The derivation of $t$ by $g$ can have one of three forms, depending on whether the root of $t$ is $\phi$, a union operation, or an extension operation.

If $t = \phi$, then $P$ contains the rule $A \to \phi$, so the algorithm returns *true* in line 12.

If $t = t_1 \uplus_{k_1 k_2} t_2$ with $t_i \in L_{B_i}(g)$ for $i = 1, 2$, then $G \triangledown \boldsymbol{p} = G_1 \uplus_{k_1 k_2} G_2$ where $G_i \in val_{\mathcal{A}}(t_i)$ for $i = 1, 2$. By the definition of $\uplus_{k_1 k_2}$, if we set $\boldsymbol{p} = \boldsymbol{p}_1 \boldsymbol{p}_2$ with $|\boldsymbol{p}_i| = k_i$ for $i = 1, 2$, then $\boldsymbol{p}_1^{\triangledown} \cap \boldsymbol{p}_2^{\triangledown} = \emptyset$ and $G_i = G \triangledown \boldsymbol{p}_i$. Since both $t_1$ and $t_2$ are smaller than $t$, the induction hypothesis yields that PARSE_REC$(B_i, \boldsymbol{p}_i)$ returns *true* for $i = 1, 2$, which means that PARSE_REC$(A, \boldsymbol{p})$ returns *true* in line 17.

If $t = \Phi[t_1]$ and $G \triangledown \boldsymbol{p} \in val_{\mathcal{A}}(t)$, then there is a graph $G_1 \in val_{\mathcal{A}}(t_1)$, say $G_1 = (V_1, E_1, lab_1, port_1)$, such that $G \triangledown \boldsymbol{p} \in \Phi(G_1)$. By the definition of extension operations, this means that there is a graph $G' \in clone_{C_\Phi}(\underline{\Phi})$ and a morphing $\mu$ of $G'$ to a graph $(V', E', lab', port')$, such that $G_1 \vDash \varphi_\Phi(m(u_1), \ldots, m(u_n))$ (where $\{u_1, \ldots, u_n\} = C_\Phi$) and $G \triangledown \boldsymbol{p} = (V_1 \cup V', E_1 \cup E', lab_1 \sqcup lab', port')$. From this we obtain a matching $m$ as follows: For every node $v \in V_\Phi$,

$$m(v) = \begin{cases} \mu(CL_v) & \text{if } v \in C_\Phi \\ \{\mu(v)\} & \text{otherwise} \end{cases}$$

By construction, $m$ satisfies (M1) (where $H = G \triangledown \boldsymbol{p}$). It also satisfies (M2), by Lemma 1 applied to $G_1$ (which applies because $G_1 \in val_{\mathcal{A}}(t_1)$). Finally, (M3) holds because $G_1 \vDash \varphi_\Phi(m(u_1), \ldots, m(u_n))$. Hence, if line 21 correctly loops over all matchings $m$, then PARSE_REC$(B, m_1(dock_\Phi))$ returns *true* by the induction hypothesis, and thus PARSE_REC$(A, \boldsymbol{p})$ does so in line 23.

We have thus finished the first direction of the proof. It remains to be shown that PARSE_REC$(A, \boldsymbol{p})$ returns *true* only if there exists a tree $t \in L_A(g)$ such that $G \triangledown \boldsymbol{p} \in val_{\mathcal{A}}(t)$. We proceed by induction on the number of recursive calls of PARSE_REC.

Since $G \triangledown \boldsymbol{p} = \phi$ if $\boldsymbol{p} = \varepsilon$, the assertion is true for the return statement in line 12.

If the return statement in line 17 is reached, then it follows from the condition $\boldsymbol{p}_1^{\triangledown} \cap \boldsymbol{p}_2^{\triangledown} = \emptyset$ that $G \triangledown \boldsymbol{p} = G \triangledown \boldsymbol{p}_1 \uplus_{k_1 k_2} G \triangledown \boldsymbol{p}_2$. We also know from the induction hypothesis that, for $i = 1, 2$, there are trees $t_i \in L_{B_i}(g)$ such that $G \triangledown \boldsymbol{p}_i \in val_{\mathcal{A}}(t_i)$. Hence, $t = t_1 \uplus_{k_1 k_2} t_2$ is a tree in $L_A(\mathcal{A})$ such that $G \triangledown \boldsymbol{p} \in val_{\mathcal{A}}(t)$.

Finally, assume that the return statement in line 23 is reached. Let $A \to \Phi[B]$ be the rule considered and $m$ the matching whose existence is guaranteed by the condition in line 23. Again, the induction hypothesis applies, this time stating that there is a tree $t_1 \in L_B(g)$ such that $G \triangledown \boldsymbol{p}_1 \in val_{\mathcal{A}}(t_1)$, where $\boldsymbol{p}_1 = m(dock_\Phi)$. Let $(V_1, E_1, lab_1, port_1) = G \triangledown \boldsymbol{p}_1$. By (M1), $m$ determines a clone $G' \in clone_{C_\Phi}(\underline{\Phi})$ of $\underline{\Phi}$ and a morphing $\mu$ of $G'$ to a graph $(V', E', lab', port')$ such that $G \triangledown \boldsymbol{p} = (V_1 \cup V', E_1 \cup E', lab_1 \sqcup lab', port')$. Using (M3), this implies that $G \triangledown \boldsymbol{p} \in \Phi(G \triangledown \boldsymbol{p}_1)$ and thus, by the definition of $val_{\mathcal{A}}$, that $\Phi(G \triangledown \boldsymbol{p}_1) \subseteq val_{\mathcal{A}}(t)$ because $G \triangledown \boldsymbol{p}_1 \in val_{\mathcal{A}}(t_1)$. This shows that $G \triangledown \boldsymbol{p} \in val_{\mathcal{A}}(t)$ and finishes the proof of the theorem. □

The remainder of the article will be devoted to the question of how line 21 can be concertized. In this section, this will first result in a polynomial-time algorithm for the special case of edge-agnostic graph extension grammars (to be defined formally in Definition 7). With this as a basis, the next section will generalize the result to local graph extension grammars.

For a fixed grammar, thanks to memoisation PARSE_REC will be called at most $O(n^{wd(\Gamma)})$ times, where $n$ is the number of nodes of the input graph. This is because there are only a constant number of possible choices for the first parameter and $n^{wd(\Gamma)}$ for the second. Hence, the total running time is $O(n^{wd(\Gamma)})$ times the maximal number of computation steps it takes to execute the body of the procedure (not counting recursive calls). This, in turn, is dominated by the time it takes to enumerate the matchings $m$ of $\Phi$ to $G \triangledown \boldsymbol{p}$ (line 21). For this, note that there is no need for line 21 to explicitly enumerate all matchings $m$ of $\Phi$ to $G \triangledown \boldsymbol{p}$ because the loop body only depends on (the nonterminal $B$ and) $m(dock_\Phi)$. In other words, the loop can be replaced with

```
⋮
21′: for all d ∈ V^⊛ do
22′:     if there is a matching m of Φ to G ▽ p with m(dock_Φ) = d then
23′:         if PARSE_REC(B, d) then
24′:             return result[A, p] ← true                    ▷ Memoise success
25′:         end if
26′:     end if
27′: end for
⋮
```

Hence, the question that remains is how to implement line 22′, that is, to decide for given $\boldsymbol{p}, \boldsymbol{d} \in V^{\circledast}$ whether there exists a matching $m$ of $\Phi$ to $G \triangledown \boldsymbol{p}$ with $m(dock_\Phi) = \boldsymbol{d}$. For this, we will now show that one can construct a cmso formula that checks whether the mapping of $dock_\Phi$ to $\boldsymbol{d}$ can be extended to a matching of $\Phi$ to $G \triangledown \boldsymbol{p}$. (This makes it possible to use Courcelle's theorem to come up with an efficient algorithm for the edge-agnostic case of graph extension grammars later on.)

To simplify our reasoning, we make use of the Backwards Translation theorem for quantifier-free operations (Courcelle and Engelfriet 2012, Theorem 5.47), by means of the following lemma.

**Lemma 2**
For every extension operation $\Phi$, there is a quantifier-free operation $\xi$ in $\ell = |dock_\Phi|$ variables such that the following holds. Let $H$ be a graph, and let $m\colon V_\Phi \to \wp(V_1)$ be a mapping that satisfies (M1) and (M2) (where $V_1 = \boldsymbol{p}_1^{\triangledown}$, as in the definition of matchings). Then $\xi(H, m(dock_\Phi)) = H \triangledown m(dock_\Phi)$.

*Proof.* Let $m$ be as in the lemma and $\boldsymbol{d} = m(dock_\Phi)$. We first show the following claim:

*Claim 3.* $H \triangledown \boldsymbol{d}$ is the graph $H'$ obtained from $H$ by removing all nodes in $[port_H] \setminus [\boldsymbol{d}]$ from it (together with their incident edges) and defining $port_{H'} = \boldsymbol{d}$.

To prove Claim 3, note first that every node in $V_H \setminus [port_H]$ is reachable from $\boldsymbol{d}$ by (M2). Thus, it remains to be shown that no node $v \in [port_H] \setminus [\boldsymbol{d}]$ is reachable from $\boldsymbol{d}$. However, this is clear by (M1) in combination with (R1). Furthermore, by definition $port_{H \triangledown \boldsymbol{d}} = \boldsymbol{d}$.

Specifying the quantifier-free operation $\xi$ is now straightforward. By Claim 3, it can be defined as follows:

- the domain formula $\delta$ expresses that a node belongs to the resulting graph if it is in $[\boldsymbol{d}]$ or not a port:

$$\delta(x_1, \ldots, x_\ell, y) = \bigvee_{i \in [\ell]} y = x_i \vee \neg \bigvee_{j \in [|port_\Phi|]} \mathbf{port}_j(y),$$

- edges and node labels are copied: for all $z \in \bar{\mathbb{L}}$ and $a \in \dot{\mathbb{L}}$

$$\theta_{\mathbf{edg}_z}(x_1, \ldots, x_\ell, y_1, y_2) = \mathbf{edg}_z(y_1, y_2) \quad \text{and} \quad \theta_{\mathbf{lab}_a}(x_1, \ldots, x_\ell, y_1) = \mathbf{lab}_a(y_1), \text{ and}$$

- the nodes in $[\boldsymbol{d}]$ become the ports:

$$\theta_{\mathbf{port}_i}(x_1, \ldots, x_\ell, y_1) = (y_1 = x_i).$$

This finishes the proof of the lemma. □

As a consequence, we can now prove the main lemma that—with the additional assumption of edge agnosticism—will lead to an efficient implementation of Algorithm 1 using Courcelle's theorem.

**Lemma 3**
For every extension operation $\Phi$ with $\ell = |dock_\Phi|$, there is a cmso formula $\psi_\Phi$ with the free individual variables $x_1, \ldots, x_\ell$ such that, for every graph $H \in \mathbb{G}_{type(\Phi)}$ and every $\boldsymbol{d} \in V_H^{\circledast}$ with $|\boldsymbol{d}| = \ell$, we have $H \models \psi(\boldsymbol{d})$ if and only if there is a matching $m$ of $\Phi$ to $H$ with $m(dock_\Phi) = \boldsymbol{d}$.

*Proof.* Let $type(\Phi) = k$ and $V_\Phi = \{v_1, \ldots, v_n\}$ for some $n \in \mathbb{N}$. The formula $\psi$ to be constructed has to check whether there exist sets $V_1, \ldots, V_n \subseteq V_H$ such that the mapping $m$ given by $m(v_i) = V_i$ maps $dock_\Phi$ to $\boldsymbol{d}$ and is a matching of $\Phi$ to $H$. Thus, the formula $\psi$ is of the form

$$\psi = \exists X_1, \ldots, X_n.\Big( \bigwedge_{\substack{(i,j) \in [n] \times [\ell] \\ v_i = dock_\Phi(j)}} \forall x.x \in X_{v_i} \leftrightarrow x = x_j \Big) \wedge \psi_{(\mathrm{M1})} \wedge \psi_{(\mathrm{M2})} \wedge \psi_{(\mathrm{M3})}.$$

Defining the conjuncts $\psi_{(\mathrm{M1})}$ and $\psi_{(\mathrm{M2})}$ is rather straightforward:

- $\psi_{(\mathrm{M1})}$ expresses that ports are bijectively mapped to ports, and for all pairs of nodes in the image of $m$, if one of them is a new node (i.e., an image of a node in $[port_\Phi] \setminus [dock_\Phi]$), then the edges between those nodes

are exactly the images of edges between their pre-images. To be precise, let $i_j \in [n]$, for $j \in [k]$, be the index such that $v_{i_j} = port_\Phi(j)$. Then

$$\psi_{(\mathrm{M1})} = \bigwedge_{j\in[\ell]} \forall x.\mathbf{port}_j(x) \to \bigwedge_{i\in[n]} (x \in X_i \leftrightarrow v_i = port_\Phi(j))$$

$$\wedge \bigwedge_{\substack{i,j\in[n],z\in\mathbb{L} \\ (v_i,z,v_j)\in E_\Phi}} \forall x,y.(x \in X_i \wedge y \in X_j \to \mathbf{edg}_z(x,y))$$

$$\wedge \bigwedge_{\substack{j\in[k],z\in\mathbb{L} \\ v_{i_j}\notin[dock_\Phi]}} \forall x,y.x \in \mathbf{port}_j(x) \to (\neg\mathbf{edg}_z(y,x) \wedge (\mathbf{edg}_z(x,y) \leftrightarrow \bigvee_{\substack{i\in[n] \\ (port_\Phi(j),z,v_i)\in E_\Phi}} y \in X_i))$$

It should be clear that $H \models \psi_{(\mathrm{M1})}(V_1,\dots,V_n,\boldsymbol{d})$ if and only if (M1) holds for the mapping $m$ that maps $v_i$ to $V_i$ for every $i \in [n]$.

- $\psi_{(\mathrm{M2})}$ expresses that for every node $v$ that is not a port, there is a port from which there is a path to $v$:

$$\psi_{(\mathrm{M2})} = \forall x.(\bigwedge_{i\in[k]} \neg\mathbf{port}_i(x)) \to \exists y.(\bigvee_{i\in[k]} \mathbf{port}_i(y) \wedge path(y,x))$$

This makes use of the fact that the predicate *path* can be expressed in monadic second-order logic; see Courcelle and Engelfriet (2012, Proposition 5.11).

- Finally, assume without loss of generality that $C_\Phi = \{v_1,\dots,v_c\}$ for some $c \in [n]$. Then $\psi_{(\mathrm{M3})}$ must be constructed in such a way that $H \models \psi_{(\mathrm{M3})}(V_1,\dots,V_c)$ if and only if $H \triangledown \boldsymbol{d} \models \varphi_\Phi(V_1,\dots,V_c)$. In fact, since $\psi_{(\mathrm{M1})}$ and $\psi_{(\mathrm{M2})}$ already ensure that (M1) and (M2) hold, Lemma 2 applies, providing us with a quantifier-free operation $\xi$, such that $\xi(H, X_{j_1}\cdots X_{j_\ell}) = H \triangledown d$ where $j_1,\dots,j_\ell \in [n]$ are the indices such that $v_{j_p} = dock_\Phi(p)$ for all $p \in [\ell]$. Hence, Theorem 1 applied to $\xi$ and $\varphi_\Phi$ yields the formula $\psi_{(\mathrm{M3})}$ needed. (Thus, $X_{j_1}\cdots X_{j_\ell}$ are to be substituted for $X_1,\dots,X_\ell$ in Theorem 1, $X_1,\dots,X_k$ play the role of $Y_1,\dots,Y_k$, and $k' = 0$.) □

Thus, line 22′ can be implemented using any algorithm which, for a fixed cmso formula $\psi$, an input graph $G$, and node sequence $\boldsymbol{d} \in V_G^{\circledast}$, checks whether $G \models \psi(\boldsymbol{d})$. Unfortunately, this is an intractable problem in general. Therefore, for efficient parsing, we aim to restrict $G$ to graphs of bounded treewidth, as in this case we can apply Courcelle's theorem (Courcelle and Engelfriet 2012, Theorem 6.4) which states that the problem can be solved in linear time on graphs of bounded treewidth. Since it is an important feature of graph extension grammars that they can generate graph languages of unbounded treewidth, we now define a special case that retains that ability while allowing us to consider graphs of bounded treewidth in the application of Lemma 3.

**Definition 7** (Edge-Agnostic Graph Extension Grammar)
A graph extension operation $\Phi$ is **edge agnostic** if $\varphi_\Phi$ does not contain any predicate of the form $\mathbf{edg}_a$ ($a \in \mathbb{L}$). An edge-agnostic graph extension grammar is a graph extension grammar in which all graph extension operations are edge agnostic.

Note that the graph extension grammar discussed in Example 4 is edge agnostic (with the exception of the discussion in its last paragraph). Recall that the purpose of the development of the graph extension grammar formalism is to capture not only structural but also non-structural reentrancies. The example grammar handles both—which despite the limitations of this example may indicate that even edge agnosticism is not as severe a restriction with respect to the linguistic usefulness of the formalism as one might think. The reason, as discussed in the Introduction, is that these non-structural reentrancies are often caused by language elements such as pronouns which, intuitively, are edge agnostic in themselves (as opposed to structural reentrancies like those caused by control).

We can now show that Algorithm 1 can be implemented to run in polynomial time for edge-agnostic graph extension grammars $\Gamma$.

**Theorem 4**
Let $\Gamma$ be an edge-agnostic graph extension grammar, and let $r$ be the maximal type occurring in its operations. Then Algorithm 1 can be made to run in time $O(n^{2r+1})$.

*Proof.* Let $G$ be the input graph, where $n = |V_G|$. Since there are $O(n^r)$ recursive invocations of the algorithm, and the loop in line 21′ is also executed $O(n^r)$ times, it suffices to argue that line 22′ can be implemented to run in linear time. We do this by using Lemma 3, but replacing the graph $G \triangledown \boldsymbol{p}$ by a graph $H$ of bounded treewidth such that $H \models \psi(\boldsymbol{d})$ if and only if $G \triangledown \boldsymbol{p} \models \psi(\boldsymbol{d})$.

The basic idea is very simple: Because $\varphi_\Phi$ does not make use of edge predicates, $\psi_{(\mathrm{M3})}$ does not do so either. (More precisely, in the definition of the quantifier-free operation $\xi$ of Lemma 2, we can replace $\theta_{\mathbf{edg}_z}$ by *false*. It follows that neither $\xi$ nor $\varphi_\Phi$ make use of the edge predicates of $G$, and as a consequence the backwards translation of $\varphi_\Phi$ does not either. The latter follows directly from the construction of that formula; see Courcelle and Engelfriet (2012), because then that entire construction effectively works with relational structures that lack the relations $\mathbf{edg}_z$ in the first place, i.e., we work with discrete graphs throughout.) Moreover, $\psi_{(\mathrm{M1})}$ only depends on edges originating at nodes that are in $[\boldsymbol{p}] \setminus [\boldsymbol{d}]$ and are thus not in $G \triangledown \boldsymbol{d}$. Hence, discarding the edges in $G \triangledown \boldsymbol{d}$ affects the truth values of neither $\psi_{(\mathrm{M1})}$ nor $\psi_{(\mathrm{M3})}$. We only need to get rid of the subformula $\psi_{(\mathrm{M2})}$ of $\psi$ because it obviously makes use of edge predicates regardless of whether or not $\varphi_\Phi$ does. However, this is easy as $\psi_{(\mathrm{M2})}$ does not contain $X_1, \ldots, X_n$, that is, the truth value it attains depends only on the choice of $d$ in line 21′. Consequently, we can add the requirement that all nodes of $G \triangledown \boldsymbol{p}$ that are not in $[\boldsymbol{p}]$ shall be reachable from $[\boldsymbol{d}]$ (which can be checked in linear time) to line 21′, delete $\psi_{(\mathrm{M2})}$ from $\psi$ to obtain

$$\psi' = \exists X_1, \ldots, X_n.\Big( \bigwedge_{\substack{(i,j)\in[n]\times[\ell] \\ v_i = dock_\Phi(j)}} \forall x.x \in X_{v_i} \leftrightarrow x = x_j \Big) \wedge \psi_{(\mathrm{M1})} \wedge \psi_{(\mathrm{M3})}$$

and check in line 22′ whether $H \models \psi'(\boldsymbol{d})$, where $H$ is obtained from $G \triangledown \boldsymbol{p}$ by removing all edges $e$ for which $src(e) \notin [\boldsymbol{p}] \setminus [\boldsymbol{d}]$. Since this graph $H$ has at most $r$ nodes that are

sources of edges, its treewidth is at most $k$. By Courcelle's theorem this means that $H \models \psi'(\boldsymbol{d})$ can be checked in linear time, which completes the proof. □

## 6. Parsing for Local Graph Extension Grammars

The restriction to the edge-agnostic case used in the preceding section in order to ensure a polynomial running time of the parsing algorithm is rather severe. In this section, we show that this restriction can partially be lifted. The NP-completeness proof in Section 4 may provide a hint: The mso formula in the rule depicted in Figure 12 inspects the entire graph. We shall now see that a certain amount of *local* structural conditions can be allowed without sacrificing efficient parsing. The idea is that we weaken the restriction imposed by edge agnosticism by adding primitive predicates to our logic that make it possible to express that a node belongs (or does not belong) to a part of the graph having a certain form. This type of predicate is inspired by the notion of (nested) graph constraint in the theory of graph transformation (see, for example Habel, Heckel, and Taentzer 1996; Arendt et al. 2014).

**Definition 8** (Local Condition and Local Node Predicate)
A **local condition** is a sequence $\chi = G_0 Q_1 G_1 Q_2 \cdots G_k$ where $G_0 \subseteq G_1 \subseteq \cdots \subseteq G_k$ are graphs[10] for some $k \in \mathbb{N}$, and $Q_1, \ldots, Q_k \in \{\exists, \neg\exists, \forall, \neg\forall\}$.

Let $G$ be a graph. We inductively define what it means for an injective morphism $\mu_0 \colon G_0 \to G$ to *satisfy* $\chi$. Every such morphism satisfies $\chi$ if $k = 0$. Assume now that $k > 0$ and let $\chi' = G_1 Q_2 \cdots G_k$. Then $\mu_0$ satisfies $\chi$ if one of the following cases holds:

1. $Q_1 = \exists$ and there exists an extension of $\mu_0$ to a morphism $\mu_1 \colon G_1 \to G$ that satisfies $\chi'$.[11]
2. $Q_1 = \neg\exists$ and there does not exist any extension of $\mu_0$ to a morphism $\mu_1 \colon G_1 \to G$ that satisfies $\chi'$.
3. $Q_1 = \forall$ and all extensions of $\mu_0$ to a morphism $\mu_1 \colon G_1 \to G$ satisfy $\chi'$.
4. $Q_1 = \neg\forall$ and there exists an extension of $\mu_0$ to a morphism $\mu_1 \colon G_1 \to G$ that does not satisfy $\chi'$.

A **local node predicate** is a unary predicate specified by a local condition $\chi$ as above, such that $G_0$ consists of a single node $u$ which is not a port. A node $v$ in a graph $G$ with the same label as $u$ satisfies $\chi$, that is, $\chi(v)$ is *true* in $G$, if the morphism that maps $u$ to $v$ satisfies $\chi$.

An example of a local node predicate (with $k = 2$) is shown in Figure 13.

**Lemma 4**
Let $\chi$ be a (fixed) local node predicate, $G$ a graph, and $v \in V_G$. Then it can be checked in polynomial time in the number of nodes of the graph $G$ whether $v$ satisfies $\chi$.

10 The reader may wish to recall the formal definition of the inclusion relation "$\subseteq$" from Section 2.
11 Thus, the requirement on $\mu_1$ is that its restriction to the subgraph $G_0$ of $G_1$ is $\mu_0$.

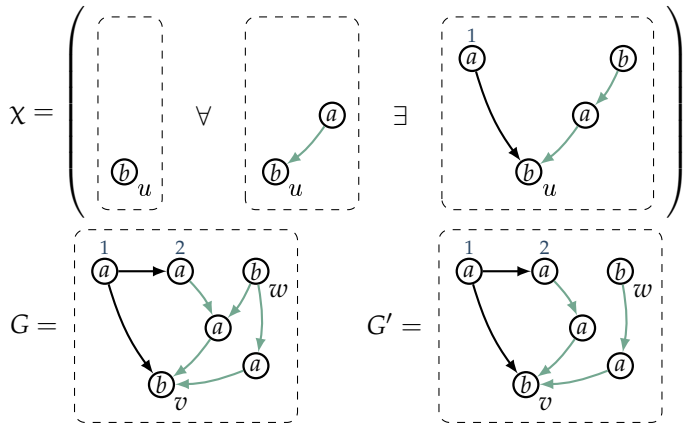



**Figure 13**
A local node predicate χ (top row) that is satisfied by a node $v$ if the following holds: The node $v$ itself has the label $b$, and for every $a$-labeled node $v'$ from which there is a green edge to $v$ (we use colors as edge labels), it holds that (1) $v'$ is the target of a green edge whose source is another $b$-labeled node and (2) there is a black edge from the first port (which carries the label $a$) to $v$. For example, in the graph $G$ (bottom row) we have that $\chi(v)$ is true because both of the $a$-labeled nodes from which there are green edges to $v$ have a green incoming edge from a $b$-labeled node (here: $w$); the first port is labeled $a$; and is the source of a black edge pointing to $v$. Note that, in fact, $\chi(w)$ is true in $G$ as well, despite the fact that no black edge connects it to the first port. This is due to the fact that $w$ does not have any incoming green edges from $a$-labeled nodes, which makes the universal condition vacuously true. In contrast, $\chi(v)$ is false in $G'$, because one of the $a$-labeled nodes from which there is a green edge to $v$ lacks the required incoming green edge from another $b$-labeled node. However, $\chi(w)$ continues to be true.

*Proof.* If $\chi = G_0Q_1G_1Q_2\cdots G_k$, it is straightforward to test whether $v$ satisfies $\chi$ by means of a recursive algorithm of fixed recursion depth $k$ mimicking the definition of satisfaction for local conditions. The body of the procedure consists of a loop which enumerates all possible extensions of the morphism that maps $G_{i-1}$ to $G$ (determined by the enclosing call) and test whether they, recursively, fulfill $G_iQ_{i+1}\cdots G_k$. Since the graphs $G_i$ are fixed, each loop runs in polynomial time, and hence the entire algorithm runs in polynomial time. □

We note here that, while in general the exponent of the polynomial bounding the running time of the test whether $v$ satisfies $\chi$ can be arbitrarily large, typical properties that need to be tested in practice are unlikely to require big and complex graphs $G_i$ or large $k$ and may therefore be expected to be of reasonable complexity.

**Definition 9** (Local Graph Extension Grammar)
Let $CMSO_{\mathrm{loc}}$ denote the logic obtained from *CMSO* by removing all predicates of the form $\mathbf{edg}_a$, where $a \in \mathbb{L}$, and adding all local node predicates. A graph extension grammar such that every extension operation $\Phi$ appearing in it satisfies $\varphi_\Phi \in CMSO_{\mathrm{loc}}$ is a **local graph extension grammar**.

**Theorem 5**
For every local graph extension grammar $\Gamma$, Algorithm 1 can be implemented to run in polynomial time.

*Proof.* Since the only difference between an edge-agnostic and a local graph extension grammar is that the extension operations of the latter can make use of the local node predicates, it suffices to show how to handle those. For this, we have to extend Lemma 3 accordingly, which we do by precomputing the required predicates for every relevant node and adding them to the relational structure. Note first that any given extension operation $\Phi$ makes use of a finite number of local node predicates. Let $\chi_1, \ldots, \chi_h$ be those predicates.

By Lemma 4 it takes polynomial time to compute, for every node $v$ of $H \triangledown \boldsymbol{d}$ (where $H$ and $\boldsymbol{d}$ are as in Lemma 3) and every $i \in [h]$, the truth value $\chi_i(v)$ with respect to $H \triangledown \boldsymbol{d}$.[12] Once this has been done, we can follow the construction in the proof of Lemma 3, but only after first having turned $H$ into the relational structure that is obtained by adding these pre-computed unary predicates $\chi_1, \ldots, \chi_h$ to it. Since the Backwards Translation theorem and Courcelle's theorem apply just as well to the resulting relational structures (which are essentially graphs with $h$ types of additional node labels), and these unary predicates do not increase the treewidth of the structures, the remainders of the proofs of Lemma 3 and Theorem 4 continue to be valid. □

We note there that the polynomial bounding the running time of the algorithm, in contrast to the edge-agnostic case, is an arbitrary one. This is unavoidable (as long as subgraph isomorphism is not in P, that is, P $\neq$ NP) because the exponent of the polynomial in Lemma 4 depends on the local node predicates occurring in the grammar. However, as mentioned above, practically relevant local node predicates are likely to be rather benign (at least for applications in NLP). For example, assuming that suitable data structures are used, the local node predicate required to implement the coordination discussed in connection with Figure 9 can be computed for all nodes $v$ of the host graph in accumulated linear time. In those cases, the running time will be essentially the same as in the edge-agnostic case.

## 7. Conclusion

We have introduced graph extension grammars, a simultaneous restriction and extension of hyperedge replacement graph grammars. Rules construct graphs using operations of two kinds: The first is disjoint union, and the second is a family of unary operations that add new nodes and edges to an existing graph $G$. The augmentation is done in such a way that all new edges lead from a new node to either

(1) a port in $G$, or

(2) any number of (arbitrarily situated) nodes in $G$, chosen by a cmso formula.

While graph extension grammars are inspired by the notion of contextual hyperedge replacement grammars (Drewes, Hoffmann, and Minas 2012), they differ from them in a rather fundamental way: Intuitively, the context nodes in a rule $r$ of a graph extension grammar refer to nodes in the subgraph generated by the very subderivation

12 Note that we need to evaluate $\chi_i(v)$ in the graph $H \triangledown \boldsymbol{d}$ rather than in $H$ because it is the former that is the potential argument of $\Phi$.

rooted in $r$. In contextual hyperedge replacement grammars, the situation is the opposite: context nodes refer to nodes generated by rule applications outside that subderivation. The latter creates a problem that does not occur in graph extension grammars, namely, that there may be cyclic dependencies which, intuitively, create deadlocks (see Drewes, Hoffmann, and Minas 2022).

By way of example, we have shown that graph extension grammars can model both the structural and non-structural reentrancies that are common in semantic graphs of formalisms such as AMR. We have presented a parsing algorithm for our formalism and proved it to be correct. If the formulas used to direct the placement of non-structural reentrancies make only "local" structural assertions, then the running time is polynomial. In general, the graph languages generated by graph extension grammars can be NP-complete (shown in Theorem 2), and hence there is little hope of being able to find an efficient membership algorithm for unrestricted graph extension grammars.

It is worthwhile repeating that the dynamic programming parsing algorithm presented here makes at most a polynomial number of recursive calls, and that the body of the algorithm runs in polynomial time provided that it can be decided in polynomial time whether a given mapping of the ports of an extension operation $\Phi$ to nodes in the input graph can be extended to a matching that satisfies $\varphi_\Phi$. To obtain the main result of this article, we exploited the fact that this is the case if $\varphi_\Phi$ is local. However, any other restriction enabling us to decide the existence of matchings in polynomial time would work just as well. In particular, there may be other ways to make sure that it suffices to test $\varphi_\Phi$ on some subgraph of bounded treewidth, such as a spanning forest. We currently do not know of a meaningful restriction that would allow us to do this, but there is another restriction that is even simpler: if no extension operation contains a context node, then graph extension grammars are special hyperedge replacement grammars,[13] and thus generate only graphs of bounded treewidth. This shows that Theorem 4 holds for such graph extension grammars, a result which is not entirely trivial because hyperedge replacement can, in general, generate NP-complete graph languages (see Section 1.1):

**Corollary 2**
Let $\Gamma$ be a graph extension grammar such that none of its extension operations contains a context node, and let $\tau$ be the maximal type occurring in its operations. Then Algorithm 1 can be implemented to run in time $O(n^\tau)$.

There are several promising directions for future work. On the theoretical side, it has to be said that the bounds on the running time proved in this article are rough upper bounds for worst-case scenarios. Since the exponents are rather high, it would be useful to have a closer look at the parameters that influence them, making a fixed-parameter analysis. A parameter that readily comes to mind is the number of reentrancies of a graph. Because the extreme case of a (directed acyclic) graph without reentrancies is a forest, we expect such an analysis to offer plenty of room for reducing the running time of our algorithms to much lower levels.

We would furthermore like to apply the new ideas presented here to the formalisms by Groschwitz et al. (2017) and Björklund, Drewes, and Ericson (2016), to see if also these can be made to accommodate contextual rules without sacrificing parsing efficiency. It is

13 Strictly speaking, this is not entirely true because, for an extension operation $\Phi$, the formula $\varphi_\Phi$ still restricts the applicability of extension operations to graphs having the property expressed by $\varphi_\Phi$, but this is well known not to increase the power of hyperedge replacement grammars, and even if it did, it would obviously not increase the treewidth of graphs.

also natural to generalize graph extension grammars to string-to-graph or tree-to-graph transducers to facilitate translation from natural-language sentences or dependency trees to AMR graphs.

On the empirical side, we are interested in a number of directions. A first step could be to check whether graph extension grammars can express the AMR languages of some existing AMR corpora, and see how the running times of parsing vary in practice depending on variables such as the number of reentrancies in the input graph or the grammar size. In addition, we would like to develop algorithms for inferring extension and union operations from AMR corpora, and in training neural networks to translate between sentences and AMR graphs using trees over graph extension algebras as an intermediate representation. Such efforts would make the new formalism available to current data-driven approaches in NLP, with the aim of adding structure and interpretability to machine-learning workflows.

**Acknowledgments**

We are immensely grateful to the reviewers for all the effort they put into their reports, which made a tremendous difference for the final manuscript. We also want to thank Yannick Stade for providing us with helpful comments on a draft of this article, and Valentin Gledel who prompted us to prove that the general membership problem for graph extension grammars is NP-complete. This work has been supported by the Swedish Research Council under grant no. 2020-03852, and by the Wallenberg AI, Autonomous Systems and Software Program through the NEST project *STING—Synthesis and analysis with Transducers and Invertible Neural Generators*.

**References**

Aalbersberg, IJsbrand Jan, Grzegorz Rozenberg, and Andrzej Ehrenfeucht. 1986. On the membership problem for regular DNLC grammars. *Discrete Applied Mathematics*, 13(1):79–85. https://doi.org/10.1016/0166-218X(86)90070-3

Allamanis, Miltiadis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *Conference Track Proceedings of the 6th International Conference on Learning Representations, ICLR 2018*, OpenReview.net, 17 pages.

Arendt, Thorsten, Annegret Habel, Hendrik Radke, and Gabriele Taentzer. 2014. From core OCL invariants to nested graph constraints. In *7th International Conference on Graph Transformation (ICGT 2014)*, pages 97–112. https://doi.org/10.1007/978-3-319-09108-2_7

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186.

Bauderon, Michel and Bruno Courcelle. 1987. Graph expressions and graph rewritings. *Mathematical Systems Theory*, 20(1):83–127. https://doi.org/10.1007/BF01692060

Berglund, Martin. 2019. Analyzing and pumping hyperedge replacement formalisms in a common framework. In *Proceedings of the Tenth International Workshop on Graph Computation Models (GCM@STAF 2019)*, pages 17–32.

Björklund, Henrik, Frank Drewes, and Petter Ericson. 2016. Between a rock and a hard place – uniform parsing for hyperedge replacement DAG grammars. In *the 10th International Conference on Language and Automata Theory and Applications, LATA 2016*, volume 9618 of *Lecture Notes in Computer Science*, pages 521–532, Springer. https://doi.org/10.1007/978-3-319-30000-9_40

Björklund, Henrik, Frank Drewes, and Petter Ericson. 2019. Parsing weighted order-preserving hyperedge replacement grammars. In *Proceedings of the 16th Meeting on the Mathematics of Language*, pages 1–11. https://doi.org/10.18653/v1/W19-5701

Björklund, Henrik, Frank Drewes, Petter Ericson, and Florian Starke. 2021. Uniform parsing for hyperedge replacement grammars. *Journal of Computer and System Sciences*, 118:1–27. https://doi.org/10.1016/j.jcss.2020.10.002

Björklund, Johanna, Frank Drewes, and Anna Jonsson. 2021. Polynomial graph

parsing with non-structural reentrancies. *CoRR*, abs/2105.02033.

Blum, Johannes and Frank Drewes. 2019. Language theoretic properties of regular DAG languages. *Information and Computation*, 265:57–76. https://doi.org/10.1016/j.ic.2017.07.011

Braune, Fabienne, Daniel Bauer, and Kevin Knight. 2014. Mapping between English strings and reentrant semantic graphs. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation, LREC 2014*, pages 4493–4498.

Carroll, John A., Nicolas Nicolov, Olga Shaumyan, Martine Smets, and David Weir. 1999. Parsing with an extended domain of locality. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pages 217–224. https://doi.org/10.3115/977035.977080

Chiang, David, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. 2013. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 924–932.

Chiang, David, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. 2018. Weighted DAG automata for semantic graphs. *Computational Linguistics*, 44(1):119–186. https://doi.org/10.1162/COLI_a_00309

Courcelle, Bruno. 1991. The monadic second-order logic of graphs V: On closing the gap between definability and recognizability. *Theoretical Computer Science*, 80:153–202. https://doi.org/10.1016/0304-3975(91)90387-H

Courcelle, Bruno and Joost Engelfriet. 2012. *Graph Structure and Monadic Second-Order Logic – A Language-Theoretic Approach*. Cambridge University Press. https://doi.org/10.1017/CBO9780511977619

Drewes, Frank. 2006. *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Drewes, Frank and Berthold Hoffmann. 2015. Contextual hyperedge replacement. *Acta Informatica*, 52(6):497–524. https://doi.org/10.1007/s00236-015-0223-4

Drewes, Frank, Berthold Hoffmann, Dirk Janssens, and Mark Minas. 2010. Adaptive star grammars and their languages. *Theoretical Computer Science*, 411:3090–3109. https://doi.org/10.1016/j.tcs.2010.04.038

Drewes, Frank, Berthold Hoffmann, and Mark Minas. 2012. Contextual hyperedge replacement. In *Applications of Graph Transformations with Industrial Relevance: 4th International Symposium, Revised Selected and Invited Papers*, number 7233 in *Lecture Notes in Computer Science*, pages 182–197, Springer. https://doi.org/10.1007/978-3-642-34176-2_16

Drewes, Frank, Berthold Hoffmann, and Mark Minas. 2017. Extending predictive shift-reduce parsing to contextual hyperedge replacement grammars. In *Proceedings of the 12th International Conference on Graph Transformation, ICGT 2017, Eindhoven, The Netherlands*, volume 11629 of *Lecture Notes in Computer Science*, pages 55–72. Springer International Publishing. https://doi.org/10.1007/978-3-030-23611-3_4

Drewes, Frank, Berthold Hoffmann, and Mark Minas. 2019. Formalization and correctness of predictive shift-reduce parsers for graph grammars based on hyperedge replacement. *Journal of Logical and Algebraic Methods in Programming*, 104:303–341. https://doi.org/10.1016/j.jlamp.2018.12.006

Drewes, Frank, Berthold Hoffmann, and Mark Minas. 2021. Rule-based top-down parsing for acyclic contextual hyperedge replacement grammars. In *Proceedings of the 14th International Conference on Graph Transformation, ICGT 2021*, Lecture Notes in Computer Science, pages 164–184. https://doi.org/10.1007/978-3-030-78946-6_9

Drewes, Frank, Berthold Hoffmann, and Mark Minas. 2022. Acyclic contextual hyperedge replacement: Decidability of acyclicity and generative power. In *Proceedings of the 15th International Conference on Graph Transformation (ICGT 2022)*, volume 13349 of *Lecture Notes in Computer Science*, pages 3–19. https://doi.org/10.1007/978-3-031-09843-7_1

Drewes, Frank and Anna Jonsson. 2017. Contextual hyperedge replacement grammars for abstract meaning representations. In *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms*, pages 102–111.

Drewes, Frank, Hans-Jörg Kreowski, and Annegret Habel. 1997. Hyperedge replacement graph grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific Publishing

Co., Inc., pages 95–162. https://doi.org/10.1142/9789812384720_0002

Groschwitz, Jonas, Meaghan Fowlie, Mark Johnson, and Alexander Koller. 2017. A constrained graph algebra for semantic parsing with AMRs. In *Proceedings of the 12th International Conference on Computational Semantics (Long papers), IWCS 2017*, 13 pages.

Groschwitz, Jonas, Alexander Koller, and Christoph Teichmann. 2015. Graph parsing with s-graph grammars. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1481–1490. https://doi.org/10.3115/v1/P15-1143

Groschwitz, Jonas, Matthias Lindemann, Meaghan Fowlie, Mark Johnson, and Alexander Koller. 2018. AMR dependency parsing with a typed semantic algebra. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Long Papers), ACL 2018*, pages 1831–1841. https://doi.org/10.18653/v1/P18-1170

Habel, Annegret, Reiko Heckel, and Gabriele Taentzer. 1996. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313. https://doi.org/10.3233/FI-1996-263404

Habel, Annegret and Hans-Jörg Kreowski. 1987. Some structural aspects of hypergraph languages generated by hyperedge replacement. In *STACS 1987*, pages 207–219, Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0039608

Joshi, Aravind and Leon S. Levy. 1982. Phrase structure trees bear more fruit than you would have thought. *American Journal of Computational Linguistics*, 8(1):1–11.

Kaplan, Ronald M. and Joan Bresnan. 1982. Lexical-functional grammar: A formal system for grammatical representation. In *The Mental Representation of Grammatical Relations*, pages 173–281, The MIT Press.

Kasper, Robert T. 1989. A flexible interface for linking applications to Penman's sentence generator. In *Proceedings of the workshop on Speech and Natural Language, HLT 1989*, pages 153–158. https://doi.org/10.3115/100964.100979

Lange, Klaus Jörn and Emo Welzl. 1987. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16(1):17–30. https://doi.org/10.1016/0166-218X(87)90051-5

Langkilde, Irene and Kevin Knight. 1998. Generation that exploits corpus-based statistical knowledge. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics (Volume 1)*, pages 704–710. https://doi.org/10.3115/980845.980963

Lautemann, Clemens. 1990. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27(5):399–421. https://doi.org/10.1007/BF00289017

Lindemann, Matthias, Jonas Groschwitz, and Alexander Koller. 2019. Compositional semantic parsing across graphbanks. In *Proceedings of the 57th Conference of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2019*, pages 4576–4585. https://doi.org/10.18653/v1/P19-1450

Lindemann, Matthias, Jonas Groschwitz, and Alexander Koller. 2020. Fast semantic parsing with well-typedness guarantees. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 3929–3951. https://doi.org/10.18653/v1/2020.emnlp-main.323

Mezei, Jorge and Jesse B. Wright. 1967. Algebraic automata and context-free sets. *Information and Control*, 11(1–2):3–29. https://doi.org/10.1016/S0019-9958(67)90353-1

Quernheim, Daniel and Kevin Knight. 2012. Towards probabilistic acceptors and transducers for feature structures. In *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 76–85.