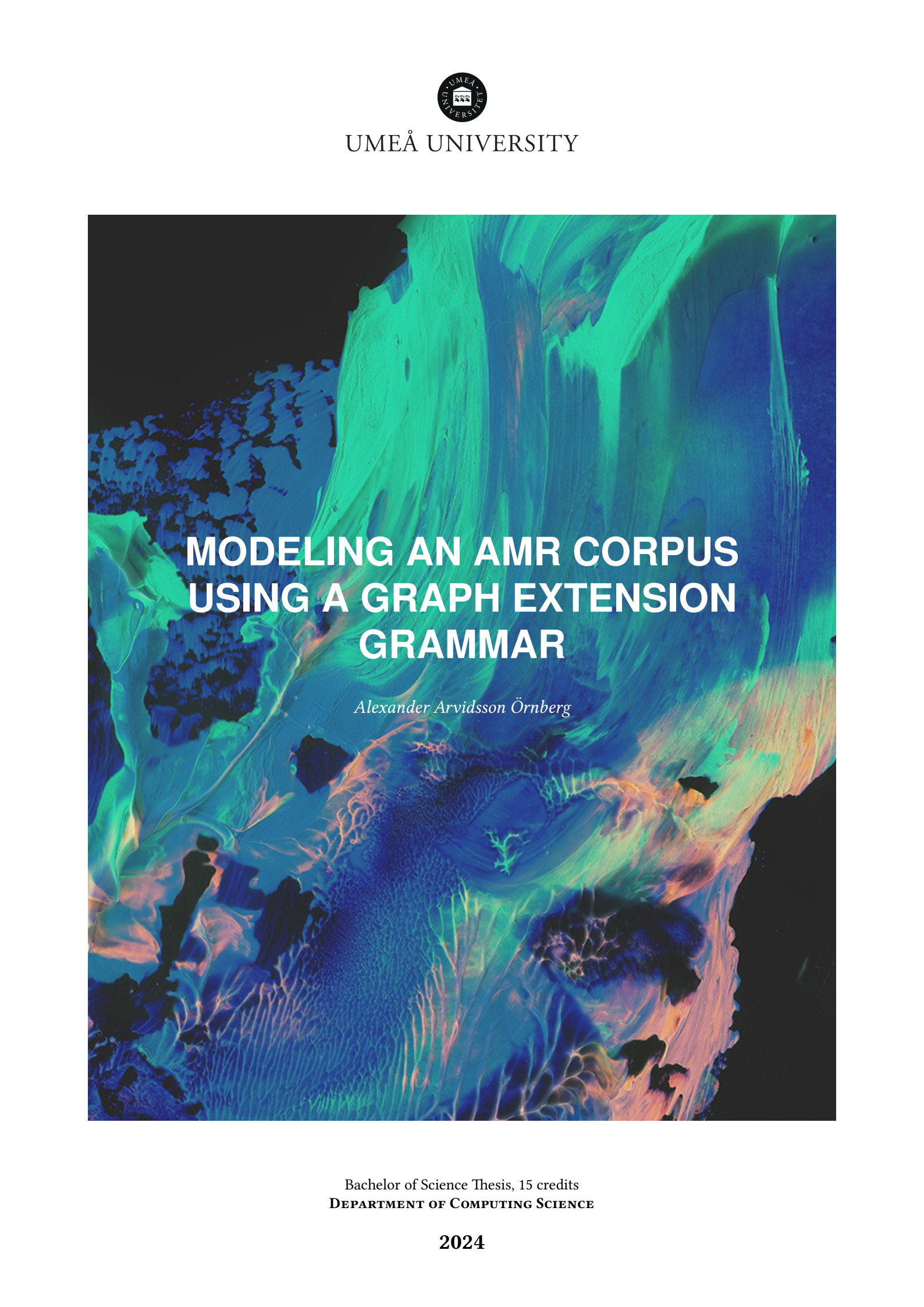

Bachelor of Science Thesis, 15 credits

Department of Computing Science

2024

Abstract

Abstract Meaning Representation (AMR) is a type of semantic graph, which is a convenient and popular way of representing natural language. Linguistic concepts are modeled as nodes with edges between them describing their relationships. A Graph Extension Grammar (GEG) is a type of graph grammar that can generate semantic graphs akin to AMR. The aim of this thesis is to explore the limitations and suitability of using graph extension grammars for semantic generation. This is done by modeling a single GEG after an AMR corpus. A large portion of this thesis is focused on generic structures in AMRs, and how to model them in a GEG. Further improvements to the formalism are also presented. The conclusion states that the GEG formalism is suitable for semantic graph generation and that it is possible to generate a corpus using a single GEG. However, large corpora may be difficult and time-consuming to model due to complex reentrancies.

Keywords: Graph Grammars, Graph Extension Grammar, AMR, Corpus Modeling

Acknowledgements

I would like to thank Anna Jonsson for taking the time to supervise this thesis, and for introducing me to the concept of graph grammars. She is one of the best teachers I have had during my three years at Umeå University, and it would not have been possible to do this without her. I would also like to thank her for believing in me and encouraging me throughout the time of writing this thesis.

Contents

List of Definitions and Theorems

1 Introduction

Semantic graphs are a convenient and popular way of representing natural language, where linguistic concepts are modeled as nodes with edges between them describing their relationships. They are particularly flexible and versatile and are relatively easy for both humans and computers to interpret and manipulate. Additionally, they have a strong mathematical foundation. An example of such a formalism is the Abstract Meaning Representation (AMR) [1]. Graph grammars provide a way to model semantic graphs using generative rules, and a recent graph grammar formalism is the Graph Extension Grammar (GEG) [2]. There have been various studies on the suitability of graph grammar formalisms for semantic graph generation, and the goal of this thesis is to investigate the suitability of graph extension grammars. The goal of this thesis is to answer the following questions:

(i) Is it possible to manually create a graph extension grammar that generates all AMRs in a given AMR corpus?

(ii) What AMR corpus is small enough to make the above feasible but still contains a variety of semantic structures to challenge the grammar type?

Additionally, the purpose of (i) is to identify semantic structures that are difficult to model, and to identify and suggest further improvements or modifications for the formalism.

The research questions are answered by selecting an appropriate AMR corpus, and manually designing a graph extension grammar. To aid with testing and verification, a tool introduced by Stade [3] is used. This tool can verify if a given GEG can generate all semantic graphs in an AMR corpus. Furthermore, a purpose-made domain-specific language (DSL), which acts as a frontend for the parsing tool, is used to create the GEG.

The structure of this thesis is as follows. Chapter 2 introduces related work, including other formalisms closely related to graph extension grammars. AMR is introduced in Chapter 3, including the chosen corpus for this thesis, along with a motivation of why the corpus was chosen. Chapter 4 introduces the GEG formalism and all necessary background. The methodology, including a membership algorithm, an AMR parsing tool, and a domain-specific language is introduced in Chapter 5. Additionally, various transformations that are needed for an AMR to be able to be parsed by the aforementioned parsing tool are introduced in this chapter. Finally, a discussion on the results obtained, and modeling AMRs using a GEG in general are put forth in Chapter 6. This chapter also highlights various improvements that can be made to the GEG formalism, and that the GEG formalism is suitable to model a corpus. Closing thoughts and conclusions are highlighted in Chapter 7.

2 Related Work

Abstract Meaning Representation (AMR) was introduced by Banarescu *et al.* [1] to aid sembanking (a collection of semantic structures). They found that existing methods of representing the semantics behind natural language were fragmented, with multiple different annotations and evaluations. The motivation behind AMR was to create a simple and readable sembank of English sentences for statistical natural language understanding (NLU). AMR is formally introduced in Chapter 3.

The Hyperedge Replacement Grammar (HRG) formalism was originally introduced in the seventies by Feder [4] and Pavlidis [5]. It has since been studied intensively by various authors, and Drewes and Kreowski [6] published a summary of its basic features. HRG is a context-free graph grammar that can generate graph-based languages. A hyperedge is an atomic item with a terminal or non-terminal label and a fixed number of tentacles. A graph containing hyperedges is called a hypergraph, and a non-terminal hyperedge can be replaced by a hypergraph containing at least the same number of nodes as tentacles in the hyperedge. Hyperedges are replaced until there are no more non-terminal ones, and the resulting graph is part of the language generated by the HRG.

The suitability of HRGs for AMR corpora generation was investigated by Jonsson [7], where they look at two variants in which the membership problem is solvable in polynomial time. The two variants are the predictive top-down (PTD) parsable grammar [8] and the restricted directed acyclic graph (rDAG) grammar [9]. They find that neither variant is suitable for AMR generation and that other formalisms should be considered. Drewes and Jonsson [10] conclude that the reason HRGs cannot be used for AMR generation is that they cannot refer back to previous nodes, also known as *reentrancies*. To get around this problem, they opted to use the Contextual Hyperedge Replacement Grammar (CHRG) formalism in [11]. They allow matching nodes based on labels and allow referencing previous nodes.

The Graph Extension Grammar (GEG) was introduced by Björklund, Drewes, and Jonsson [2] and is a special case of the CHRG. The design of the formalism allows it to generate graphs with *non-structural reentrancies*, which is a type of node sharing that is common AMR (see Chapter 3). They also provide a parsing algorithm for GEGs, which is proved to be correct and run in polynomial time. Furthermore, they later introduced an updated version of the formalism in [12], which allows using logical formulas in contextual node matching instead of using their label. GEG is formally introduced in Chapter 4.

The usage of GEGs for corpora generation of AMRs was examined by Andersson [13]. Their main focus was to develop a tool named Lovelace, which given a GEG can generate a corpora of semantic graphs. They also investigate which functionalities and design aspects are important in corpora generation. Stade [3] continues the research on GEGs by investigating their language theoretic properties. They further improve the parsing algorithm introduced in [2] and made a parsing tool that given a GEG and an AMR can verify if it is possible to derive the semantic graph from the grammar. Their parsing tool forms the basis for this thesis and is introduced in further detail in Chapter 5.

3 Abstract Meaning Representation

Abstract Meaning Representation (AMR) is a semantic representation language with a primary focus on the English language [1]. It is concerned with concepts and relationships between them, rather than the exact syntax of sentences. For example, words such as "as" or "it" are abstracted away as they are considered syntactic sugar. This allows one AMR to represent the semantic meaning behind an array of various sentences that convey the same message, abstractly and concisely. The full AMR specification is available on GitHub[1].

Before continuing, we must formally define the concept of a graph, a path in a graph, and a reentrant graph, as well as some notation inspired by [2]. Given a set S, the set of all finite sequences of elements from S is denoted by S^*. The subset of S^* where no element of S occurs more than once in each sequence is denoted by $S^{\circledast}$. The set of natural numbers, including the number zero, is denoted by $\mathbb{N}$. For some $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$. Furthermore, given a sequence w, we can express w as a set by $[w]$.

Definition 3.1 (Base Graph)**.** *A* labeling alphabet $\mathbb{L} = (\dot{\mathbb{L}}, \bar{\mathbb{L}})$ *is a pair of finite sets containing node and edge labels. A* (directed) graph over $\mathbb{L}$ *is a triple* $G = (V, E, lab)$ *where*

- V *is a finite set of nodes,*
- $E \subseteq V \times \bar{\mathbb{L}} \times V$ *is a finite set of edges, and*
- $lab : V \to \dot{\mathbb{L}}$ *assigns a label to each node.*

Definition 3.2 (Path)**.** *A* path *in a directed graph* G *is a sequence* $P = (v_1, \ell_1, \ldots, \ell_{n-1}, v_n)$ *such that* $(v_i, \ell_i, v_{i+1}) \in E$ *for all* $i \in [n-1]$ *where* $n \in \mathbb{N}$*. If there exists a path* $P = (v_1, \ell_1, \ldots, \ell_{n-1}, v_n, \ell_n, v_1)$ *in* G*, then* G *is said to be* cyclic *and* P *a* cycle*. The* length *of an acyclic path, denoted by* $|P|$*, is equal to the number of edges in the path. The longest path in a graph* G *is said to be the* depth *of* G*.*

The textual representation of an AMR may contain what appears to be cyclic relations. These are, however, inverse relations. For example, an edge from a node A to B with the label "arg1-of" means that the edge is directed from B to A, i.e., inversed. Inverting an edge in a graph can introduce a new root node, which is not allowed in AMR. This is not discussed in any official documentation and it is therefore not clear how this is handled.

Definition 3.3 (Reentrant Graph)**.** *A reentrant graph is a directed graph where there exists a node with two or more incoming edges. Consequently, a non-reentrant graph is a graph in which all nodes have at most one incoming edge.*

AMRs are rooted, directed, and labeled graphs with nodes and edges representing instances of concepts and relations. An example of a concept is "want", which has at least two arguments (or edges), the wanter and the thing wanted. The PropBank[2] is a database of concepts and their arguments used by AMR. There may be multiple versions of a concept and

[1] github.com/amrisi/amr-guidelines/blob/master/amr.md
[2] propbank.github.io/v3.4.0/frames/

in that case, they are differentiated by a number after the concept name. Furthermore, AMR abstracts away from co-references such as pronouns, reflexives, and control structures. Instead, AMR allows referencing the same instance of a concept multiple times. This is referred to as *reentrancies*, and is achieved by creating instances of concepts, which can be referred to more than once. For example, a node labeled `b / boy` would indicate an instance "b" of the concept "boy". If another node has the label "b" that indicates a reference to the previously created concept.

Figure 1 illustrates a textual AMR representation of the sentence "The boy wants the girl to believe the boy". The edge-label "ARG0" from the node "w / want-01" indicates who is wanting something, and in this case, the boy wants something. The other edge label, "ARG1", indicates what is wanted, which is the concept of believing. The exact meaning of each edge label is not important in this thesis, unless otherwise stated, but can be found by looking up a concept in the PropBank[1].

```
(w / want-01
    :ARG0 (b / boy)
    :ARG1 (b2 / believe-01
                :ARG0 (g / girl)
                :ARG1 b))
```

Figure 1: An example of a reentrant AMR (textual format).

Figure 2 illustrates the graphical representation of Figure 1. The textual representation is used when writing AMR, and the graph format is used computationally. An AMR corpus is a set of AMRs in their textual format.

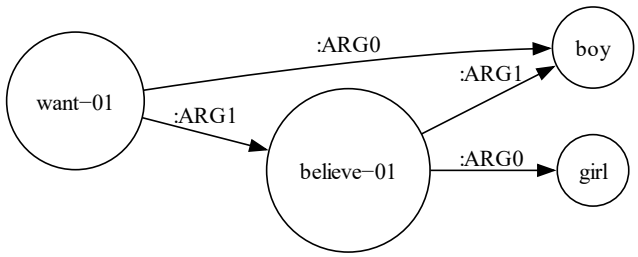

Figure 2: An example of a reentrant AMR (graph format).

3.1 The Little Prince Corpus

The second research question of this thesis is concerned with determining which corpus is small enough, yet still contains a variety of semantic structures to challenge the GEG formalism. Quantifying the suitability of a corpus is not a straightforward task. The authors behind AMR have released various corpora, which are meant to highlight the formalism and are a suitable starting point. Their only free corpus is based on the novel *The Little Prince*

[1] propbank.github.io/v3.4.0/frames/

by Antoine de Saint-Exupéry, published in 1943. This corpus has been revised and published numerous times. The version used in this thesis was published on March 14th, 2016.

The Little Prince corpus[1] is split into three different corpora, namely, a *development*, a *training*, and a *test* corpus. They contain 145, 1274, and 143 semantic graphs, respectively, with a total of 1,562 graphs representing the complete novel. The development and training corpora are meant for evaluating neural networks that are trained on the training data, and are therefore representative of the full corpus (i.e., the set of all graphs representing the novel). The development corpus was chosen for this thesis since it is small enough to be processed manually.

As previously stated, the corpus contains 145 AMRs and analyzing the graphs yields that it contains 433 unique concepts. There are 186 unique nodes with respect to their outgoing edge labels and 48 unique edge labels (i.e., *relations*). The deepest graph in the corpus has a depth of nine. The fact that it contains relatively deep graphs is preferable since that increases the size and complexity of a graph grammar needed to represent it. Below is a list of randomly selected sentences from the corpus, where each sentence has an AMR representation with an increasing depth, ranging from one to nine.

1. Anywhere.
2. Straight ahead of him.
3. Then the little prince said, earnestly: “That doesn’t matter.”
4. Where I live, everything is so small!
5. And, with perhaps a hint of sadness, he added: “Straight ahead of him, nobody can go very far …”
6. After a reflective silence he answered: “The thing that is so good about the box you have given me is that at night he can use it as his house.”
7. My friend broke into another peal of laughter: “But where do you think he would go?”
8. The first time he saw my airplane, for instance (I shall not draw my airplane; that would be much too complicated for me), he asked me: “What is that object?”
9. My Drawing Number Two looked like this: The grown-ups’ response, this time, was to advise me to lay aside my drawings of boa constrictors, whether from the inside or the outside, and devote myself instead to geography, history, arithmetic and grammar.

[1] amr.isi.edu/download/amr-bank-struct-v1.6.txt

4 Graph Extension Grammar

This chapter introduces the Graph Extension Grammar (GEG) formalism, and a domain-specific language used to construct them. GEG was introduced by Björklund, Drewes, and Jonsson [2] and is a variant of the CHRG [11] graph grammar. It consists of a regular tree grammar (RTG) and an algebra. The RTG is used to generate trees which are then evaluated over the algebra to create semantic graphs. Section 4.1 covers the mathematical notation needed before introducing the formalism in Section 4.2. All definitions in this chapter, with the exception of Definition 4.1, are based on definitions from [2, 3, 12, 13].

4.1 Preliminaries

The graph extension grammar formalism requires a redefinition of the concept of a graph with an additional component $port \in V^{\circledast}$, which is a finite sequence of nodes. A graph is defined as a quadruple $(V, E, lab, port)$ where V, E, lab are the same as defined in Definition 3.1. This definition of a graph is used going forward unless explicitly stated otherwise. Transforming a portless graph into a ported graph may be done using a graph port transformation.

Definition 4.1 (Graph Port Transformation)**.** *Let $A = (V, E, lab)$ be a base graph as defined in Definition 3.1, and $R \subseteq V$ the set of all root nodes in A (i.e., all nodes with no incoming edges). A* graph port transformation *function is defined as $\phi\colon A \to B$ where $B = (V, E, lab, R_\delta)$ and R_δ is a non-repeating sequence of all nodes in R, i.e., $[R_\delta] = R$. The order of nodes in R_δ is chosen in a non-deterministic manner.*

The graph port transformation function can be applied to an AMR graph to get an equivalent graph with ports that can be used in the context of graph extension grammars. Similarly, the port component of a graph can be removed to get a base graph.

Given a graph G, the *type* of G is defined as $type(G) = G_\tau = |port|$. The set of all graphs of type τ_1 may be written as $\mathbb{G}_{\tau_1}$. The empty graph $(\varnothing, \varnothing, lab, port)$ is denoted by $\varnothing$. Furthermore, all graphs must have at least one port, and each root node must be a port. Additionally, given two functions $f_1\colon S_1 \to T_1$ and $f_2\colon S_2 \to T_2$, we define the function $f_1 \sqcup f_2\colon S_1 \cup S_2 \to T_1 \cup T_2$ as

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

Definition 4.2 (Ranked Alphabet)**.** *A* ranked alphabet *is a pair (Σ, σ) where Σ is a finite set of symbols and σ a function that assigns each $f \in \Sigma$ a rank. The set consisting of all symbols of rank $k \in \mathbb{N}$ in Σ is denoted by Σ_k. If the rank is understood from context or not relevant, then σ may be omitted and the ranked alphabet denoted by Σ.*

Example 4.3 (Ranked Alphabet)**.** Let Σ be a ranked alphabet that represents arithmetic expressions, where $\Sigma_0 = \{1, 2, 3\}$, $\Sigma_1 = \{-\}$, and $\Sigma_2 = \{\times, +\}$. The ranked alphabet thus contains symbols of ranks zero, one, and two.

Definition 4.4 (Tree). *The set of all well-formed trees T_Σ over a ranked alphabet Σ is defined inductively. For all $f \in \Sigma_k$ where $k \in \mathbb{N}$ and every tree $t_1, \ldots, t_k \in T_\Sigma$, if $k = 0$ then $f[]$ (or simply f) $\in T_\Sigma$, otherwise, $f[t_1, \ldots, t_k] \in T_\Sigma$.*

Example 4.5 (Tree). Let Σ be the ranked alphabet from Example 4.3. Figure 3 illustrates an example of a tree that can be constructed using the symbols in Σ. An equivalent way of representing the same tree is $-(\times(+(1,2),3))$.

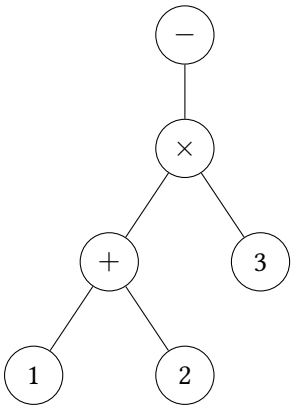

Figure 3: A graphical visualization of the tree $-(\times(+(1,2),3))$.

Definition 4.6 (Algebra). *Let Σ be a ranked alphabet. A Σ-algebra over a domain $\mathbb{A}$ is a pair $\mathcal{A} = (\mathbb{A}, \{f_\mathcal{A} : \mathbb{A}^k \to \mathbb{A} \mid f \in \Sigma_k, k \in \mathbb{N}\})$. The result of evaluating a tree $t \in T_\Sigma$ with respect to $\mathcal{A}$ is given by $val_\mathcal{A}(t)$, and is defined recursively. If $t = f[t_1, \ldots, t_k]$ then $val_\mathcal{A}(t) = f_\mathcal{A}(val_\mathcal{A}(t_1), \ldots, val_\mathcal{A}(t_k))$.*

In the context of graph extension grammars, the domain $\mathbb{A}$ is the set of all graphs over an alphabet $\mathbb{L}$ and the empty graph. The function $f_\mathcal{A} : \mathbb{A}^k \to \mathbb{A}$ accepts k graphs, and returns one graph, from $\mathbb{A}$. This essentially means that the rank of a symbol in Σ is equal to the arity of the function it represents in the algebra.

Example 4.7 (Algebra). Let Σ be the ranked alphabet from Example 4.3, and $\mathcal{A}$ an algebra in which the domain is the set of all integers $\mathbb{Z}$. All symbols $f \in \Sigma$ with rank ≥ 1 must be present in $\mathcal{A}$, and those with a rank of zero can be seen as constants. In this case, we let $- \in \Sigma_1$ represent negation in the algebra, i.e., the function $-(x) \to -x$ where the initial $-$ is the function name. Similarly, we let $+$ and $\times$ represent addition and multiplication, respectively, in the algebra. Evaluating the tree t in Figure 3 from Example 4.5 bottom-up, that is $val_\mathcal{A}(t)$, yields the result -9. The algebra may be written as $\mathcal{A} = (\mathbb{Z}, \{+, -, \times\})$.

Definition 4.8 (Regular Tree Grammar). *A* Regular Tree Grammar *(RTG) is a quadruple $\gamma = (N, \Sigma, P, S)$ where*

- *N is a ranked alphabet with rank 0, called nonterminals,*
- *Σ is a ranked alphabet, called terminals, and is disjoint from N,*
- *P is a finite set of productions on the form $A \to f[B_1, \ldots, B_k]$, where $f \in \Sigma_k$ and $A, B_1, \ldots, B_k \in N$, and*
- *$S \in N$ is the initial nonterminal.*

Example 4.9 (RTG). Let Σ be the ranked alphabet from Example 4.3. Consider the RTG $R = (\{A, B, C, D\}, \Sigma, P, A)$ where P is equal to:

1. $A \to -(B)$
2. $B \to \times(C, D)$
3. $C \to +(D, D)$
4. $D \to 1 \mid 2 \mid 3$

The tree in Figure 3 is an example of a tree that can be generated by R. The symbol $\Rightarrow_x$ denotes an application of rule x in R. Figure 3 is derived by $A \Rightarrow_1 -(B) \Rightarrow_2 -(\times(C, D)) \Rightarrow_4 -(\times(C, 3)) \Rightarrow_3 -(\times(+(D, D), 3)) \Rightarrow_4 -(\times(+(1, D), 3)) \Rightarrow_4 -(\times(+(1, 2), 3))$.

4.2 Formal Definition

A graph extension grammar contains two types of operations, an extension operation and a disjoint union operation, with arity one and two, respectively. These operations are functions in the algebra and the operations themselves are used as symbols in the ranked alphabet. The ranked alphabet also contains the empty graph of arity zero, and has no associated function in the algebra. Trees generated by the regular tree grammar are evaluated bottom-up, with respect to the algebra, to build a graph.

Definition 4.10 (Extension Operation)**.** *An* extension operation *is defined as a graph with the additional component $dock \in V^{\circledast}$, formally defined as $\Phi = (V_\Phi, E_\Phi, lab_\Phi, port_\Phi, dock_\Phi)$. The* underlying graph *of Φ is denoted by $\underline{\Phi}$ and defined as $(V_\Phi, E_\Phi, lab_\Phi, port_\Phi)$. Nodes in the set $CONT_\Phi = V_\Phi \setminus ([port_\Phi] \cup [dock_\Phi])$ are referred to as* contextual nodes*, and nodes in the set $NEW_\Phi = [port_\Phi] \setminus [dock_\Phi]$* new nodes*. An extension operation must satisfy the following conditions:*

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

(ii) $[dock_\Phi] \setminus [port_\Phi] \subseteq \{t \in V_\Phi \mid (s, \ell, t) \in E_\Phi\}$.

The first condition ensures that edges can only start from new nodes, and end in all nodes except new nodes. This ensures that all graphs created by a GEG are directed acyclic graphs. The second condition states that a target node of an edge must be a dock but not a port or a contextual node, and ensures that all nodes are reachable in the final graph. An extension operation has a rank of one in Σ, and arity one since it accepts one input graph.

Given an input graph $g = (V, E, lab, port)$ and an extension operation Φ, applying the extension operation on the graph yields a new graph:

$$g' = (V_\Phi \cup V, E_\Phi \cup E, lab_\Phi \sqcup lab, port_\Phi).$$

The type of g must equal $|port_\Phi|$ for Φ to be applicable to g, since all ports in g are fused with the docks in Φ. The ports in the resulting graph g' are those that originated from Φ, as all ports in g were fused with the docks. For each contextual node $C \in CONT_\Phi$, it is fused to a node in g with the same label as C. If there are multiple nodes in g with the same label as C, then the node to fuse with is chosen in a non-deterministic manner. It is also important to note that nodes that are marked as docks do not have a label, as they inherit their label from the port node they are fused with.

It is also possible for the underlying graph in an extension operation to contain isolated nodes. They are nodes with no incoming or outgoing edges and are used to connect a node

in an input graph with an underlying graph in an extension graph not further up in the tree. Furthermore, isolated nodes must be a port and a dock, and cannot be a contextual node.

Example 4.11 (Extension operation)**.** Figure 4 illustrates an example of an extension operation of the concept of "wanting". The root node is marked as a port with a 1 above the node, and both children are marked as docks with a dock number beneath each node.

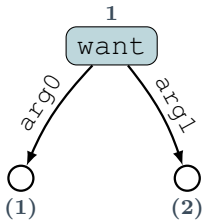

Figure 4: An example of an extension operation Φ.

Example 4.12 (Applying an extension operation)**.** Consider the graph g in Figure 5 that contains two nodes with no edges. Since there are two nodes and both nodes have a port number, the type of the graph is two. The extension operation Φ in Figure 4 contains two docks and can therefore be applied to g. Applying Φ to g yields the semantic graph g' in Figure 6. The graph g' is a representation of the sentence "The boy wants the girl" (and other similar sentences).

Figure 5: A graph g with two nodes and no edges.

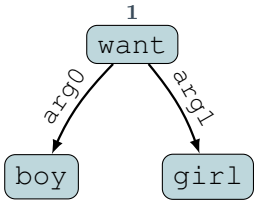

Figure 6: The result of applying the extension operation Φ to g.

Definition 4.13 (Sequence Concatenation)**.** *Given two finite sequences, $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$, concatenating the two sequences, represented by the symbol $\cdot$, yields a new finite sequence $A \cdot B = (a_1, \ldots, a_n, b_1, \ldots, b_m)$.*

The second type of operation, a disjoint union, consumes two input graphs and returns a new graph. Essentially, given two input graphs A_{τ_1} and B_{τ_1}, applying the disjoint union operation returns a new graph of type $\tau_1 + \tau_2$. The reason for the type is that the ports of A and B are concatenated to form the port sequence of the output graph. If any two nodes in A and B have the same identity, then an isomorphism is applied such that A and B become disjoint.

Definition 4.14 (Disjoint Union Operation)**.** *A* disjoint union *is a tuple (τ_1, τ_2) written as $\uplus_{\tau_1,\tau_2}$ with the type $\tau_1 + \tau_2$. The* associated function with a disjoint union *is defined as $f_{\uplus_{\tau_1,\tau_2}} : \mathbb{G}_{\tau_1} \times \mathbb{G}_{\tau_2} \to \mathbb{G}_{\tau_1+\tau_2}$. Applying the disjoint union on the graphs $g_{\tau_1} = (V, E, lab, port)$ and $g'_{\tau_2} = (V', E', lab', port')$ yields the resulting graph $(V \cup V', E \cup E', lab \sqcup lab', port \cdot port')$ with the type $\tau_1 + \tau_2$. If V and V' are not disjoint, an isomorphism is applied to g', before applying the operation, such that all nodes in g and g' are distinct.*

Example 4.15 (Applying a disjoint union operation)**.** Let $g = (\{A\}, \varnothing, lab_g, (A))$ and $g' = (\{B\}, \varnothing, lab_{g'}, (B))$ be two distinct graphs, where A and B are both port nodes. Applying the disjoint union operation on g and g', denoted by $g \uplus_{11} g'$ results in a new graph $g'' = (\{A, B\}, \varnothing, lab_g \sqcup lab_{g'}, (A, B))$. The new graph contains all nodes from both input graphs and the port sequences of both graphs have been concatenated to form a new port sequence. Furthermore, Figure 5 is an example of two graphs that have been converted to one graph using a disjoint union.

Definition 4.16 (Graph Extension Grammar)**.** *Given a ranked alphabet* Σ*, a* Graph Extension Grammar *(GEG) is a pair* $\Gamma = (\gamma, \mathcal{A})$ *where* γ *is a regular tree grammar over* Σ*, and* $\mathcal{A}$ *is an algebra over* Σ*.*

Example 4.17 (An example of a tree and its evaluation in a GEG)**.** Let $\Gamma = (\gamma, \mathcal{A})$ be a GEG. Figure 7 illustrates a tree $t \in L(\gamma)$. The circles represent extension and union operations, respectively. The symbol next to each circle represents the non-terminal in γ for each rule. The dotted line in the figure illustrates how the contextual node "one" is fused with another node with the same label further down the tree. Evaluating the tree, or $val_{\mathcal{A}}(t)$, yields the semantic graph in Figure 8. The semantic graph represents the sentence "When a mystery is too overpowering, one dare not disobey" (and other similar sentences).

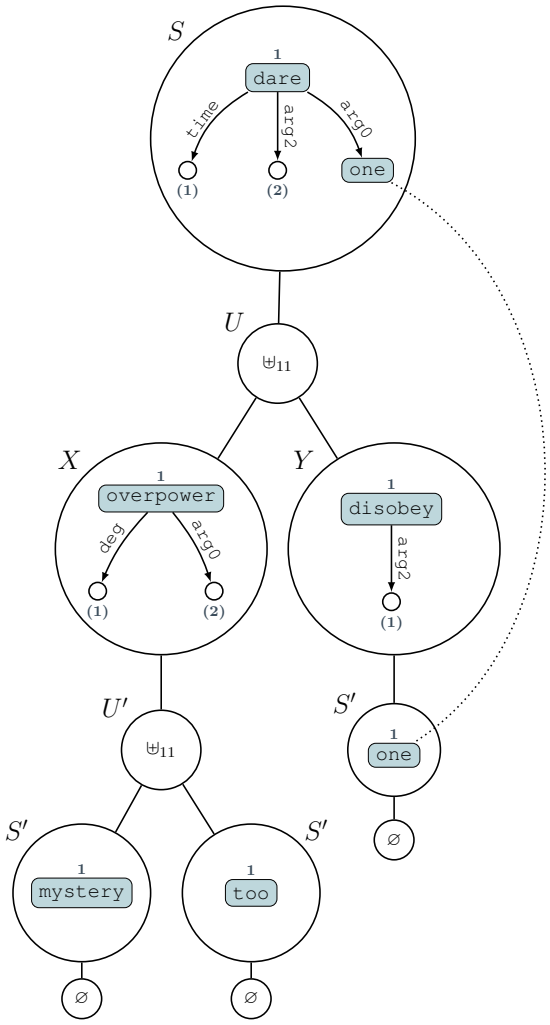

Figure 7: A visualization of the tree $S(U(X(U'(S', S')), Y(S')))$ in $L(\gamma)$.

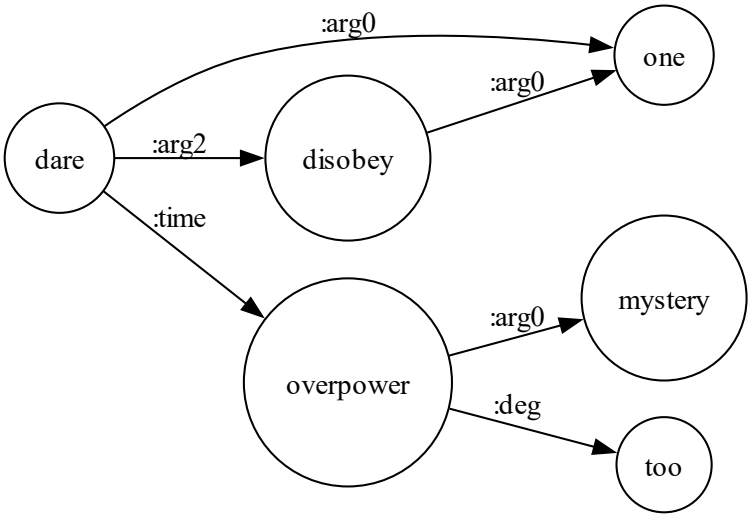

Figure 8: Result of evaluating the tree in Figure 7 with respect to $\mathcal{A}$.

5 Methodology

This chapter describes the methodology of the empirical study in this thesis. Given a set $C = \{\phi(A) \mid A \in X\}$, where X is the set of all AMR graphs in The Little Prince corpus and ϕ is the graph transformation function defined in Definition 4.1, this thesis aims to determine if it is possible to construct a single GEG Γ such that $C \cap L(\Gamma) = C$. This is achieved by using a Python implementation of the membership algorithm introduced in Section 5.1, and a purpose-made domain-specific language introduced in Section 5.3.

5.1 A Membership Algorithm

The graph language generated by a graph extension grammar Γ is the set union of all graphs that can be generated by Γ, and is formally defined as

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

where $L(\gamma)$ is the set of all trees generated by the RTG in Γ. Given an AMR graph A, the membership problem is concerned with determining if $\phi(A) \in L(\Gamma)$.

In addition to introducing the graph extension grammar formalism, Björklund, Drewes, and Jonsson [2] introduced a polynomial time membership algorithm with a theoretical upper bound of $\mathcal{O}(n^{2c+1})$. This algorithm was improved by Stade [3], where the degree of the polynomial bounding its runtime was decreased by a factor of two, resulting in an upper bound of $\mathcal{O}(n^{c+1})$. Furthermore, Stade provided a concrete Python implementation of their algorithm, and forms the basis of the empirical study in this thesis. The Python implementation provides a parsing algorithm that can parse a textual representation of an AMR into a graph format. Although, the AMR parsing in the original implementation is not fully compatible with the AMR specification. As such, AMRs need to be transformed before being parsed, and the transformation is described in Section 5.2.

5.2 AMR Transformations

The AMR specification allows edges to literal values, as opposed to concepts or references. An edge to a literal means that there is an edge to something that is not a node, such as a string or a number. For example, the edge ":quant 5" specifies a quantity of something, but the number of things is not a concept. The AMR parsing algorithm described in Section 5.1 only supports concepts and references to previously declared concepts. To work around this, AMRs must be transformed in a way that preserves the semantics and only changes the textual representation of it. This is done by converting literal values into concepts. Furthermore, AMR does not specify the order in which concepts are defined. For example, when parsing an AMR from top to bottom, a reference to a concept may be used before the concept is defined. To work around the aforementioned problems, we define an AMR transformation function in

Definition 5.1.

Figure 9 illustrates the AMR representation of the sentence “Once when I was six years old I saw a magnificent picture in a book, called True Stories from Nature, about the primeval forest.”. It contains both references to concepts that are defined later and edges to literals. The exact meaning of the AMR in Figure 9 is not important, rather, it serves as an example of edges that must be transformed.

```
(s / see-01
    :ARG0 i
    :ARG1 (p / picture
        :mod (m / magnificent)
        :location (b2 / book
                :name (n / name
                    :op1 ”True”
                    :op2 ”Stories”
                    :op3 ”from”
                    :op4 ”Nature”)
                :topic (f / forest
                    :mod (p2 / primeval))))
    :mod (o / once)
    :time (a / age-01
        :ARG1 (i / i)
        :ARG2 (t / temporal-quantity :quant 6
                :unit (y / year))))
```

Figure 9: An AMR with edges to literal values and concepts not defined top-to-bottom.

Definition 5.1 (AMR λ-Transformation)**.** *Given an AMR A, the function $\lambda : A \to A_\lambda$ denotes the transformation of A to A_λ, where literals in A_λ have been converted to concepts and all concepts are defined from top to bottom. Instantiations of newly created concepts are prefixed with an underscore, followed by the name of the concept and a numerical suffix to allow multiple instances of the same concept. For numerical concepts, the instance name becomes “num” instead of the number, and they are also prefixed with an additional underscore. Furthermore, all upper-case letters are converted into lower-case, and string literals are stripped of their surrounding quotation marks.*

Applying the λ transformation to the AMR in Figure 9 yields the AMR shown in Figure 10. In this instance, edges such as `:op1 ”True”` were converted to `:op1 (_true0 / true)`.

5.3 A Domain-Specific Language

To aid in creating a GEG for corpus generation, we present a purpose-made domain-specific language (DSL) named *geglang*. The DSL supports union and extension operations, where the combination of an operation and an RTG is referred to as a rule. In the following sections, angle brackets and parenthesis act as placeholders to be replaced, and parenthesis may also be removed instead of replaced. For example, <literal> indicates that <literal> must be replaced with a string literal such as `"text"`. If a parenthesis is followed by an asterisk, then the contents within the parenthesis may be repeated zero or more times.

```
(s / see-01
    :arg0 (i / i)
    :arg1 (p / picture
        :mod (m / magnificent)
        :location (b2 / book
            :name (n / name
                    :op1 (_true0 / true)
                    :op2 (_stories0 / stories)
                    :op3 (_from0 / from)
                    :op4 (_nature0 / nature))
            :topic (f / forest
                    :mod (p2 / primeval))))
    :mod (o / once)
    :time (a / age-01
        :arg1 i
        :arg2 (t / temporal-quantity
            :quant (__num0 / 6)
            :unit (y / year))))
```

Figure 10: The AMR in Figure 9 transformed such that it can be parsed.

5.3.1 Variables

The syntax of a string-array variable, which is the only supported variable type, is `let <label> = [<literal>, ..., <literal>];`. The `<label>` must be replaced with a name for the variable, and all `<literal>` placeholders must be replaced with string literals. An example of a variable is `let x = ["boy", "girl"];`.

5.3.2 Union Rules

The syntax of a union rule is
`union <name> → <target> and <target> (and <target>)*;`,
where `<name>` is the name of the union rule and `<target>` represents another union or extension rule. Allowing more than two targets in a union is not permitted in a graph extension grammar union operation, however, the DSL contains syntactic sugar that allows multiple targets in one union rule. For example, the union `union U → A and B and C;` would evaluate to two unions: `union U → U' and C;` and `union U' → A and B;`. The type (or number of ports) of a union is automatically inferred and does not need to be explicitly stated. A union may also be used inline in an extension rule by omitting the name and arrow. Union rules with more than two targets are grouped left-to-right.

5.3.3 Extension Rules

The syntax of an extension rule is shown in Listing 5.1, where `<name>` represents the name of the rule and `<target>` represents an inline union or a union or extension rule name. An extension rule may go to multiple different targets by using the `or` syntax. If an extension rule does not have a target rule, then the rule goes to the empty graph.

The `<root>` and `<label>` placeholders represent a string literal, a string array, or a reference to a variable. In the case of a reference to a variable, the name of the variable must be prefixed with a dollar sign. The `<root>` is the label for the root node in the underlying graph of the extension rule. An `<edge>` represents the name of an edge from the root node

```
extension <name> (to <target> (or <target>)*) → <root> <[port]> {
    :<edge>  <(dock)> ([port]),  # Dock node
    <(dock)> <[port]>,           # Isolated node
    ...,
    :<edge>  <label>             # Context node
};
```

Listing 5.1: Extension rule syntax.

to a child node and may be omitted to indicate an isolated node (in which case the node is no longer a child node). Ports and docks are represented by <[port]> and <(dock)>, respectively, and must be written as [N] and (N) where N is a natural number. An isolated node must be a dock and have a port. A child node may be a dock or context node, and only a dock node can have a port. The port of a root node may be omitted and automatically inferred if no children have ports. An extension rule may be prefixed with `start` to indicate the initial nonterminal.

Definition 5.2 (Variable Expansion)**.** *Let E be an extension rule with a root node label represented by a variable V. If V contains $n \in \mathbb{N}, n \geq 1$ string literals, then n copies of E are created. For each E_i where $i \in [n]$, the root label of E_i is equal to the ith element of V. This process is also applicable to contextual nodes.*

A concrete example combining an extension rule, a union rule, and a variable, is shown in Listing 5.2. The GEG represents the sentences "The boy wants the girl" and "The girl wants the boy". The type of `S` is one, and the type of the union is two. Only one tree can be derived by the GEG, namely, $S(U(S, S))$. A visualization of the RTG and Algebra in the GEG is shown in Figure 11.

```
let x = ["boy", "girl"];

union U → S and S;

extension S → $x [1];

start extension S to U → "want" [1] {
    :arg0 (1),
    :arg1 (2)
};
```

Listing 5.2: Example GEG.

The GEG in Listing 5.2 can be written in a more compact manner by using inline unions and arrays, as shown in Listing 5.3. Since ports can be inferred, they are omitted. Going forward, all examples make use of inline unions and variables.

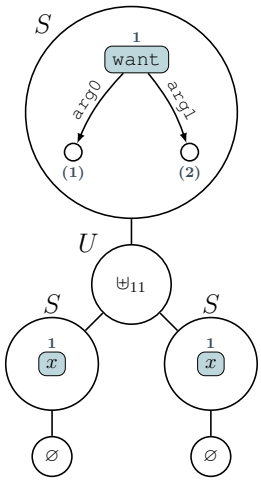

Figure 11: Visualization of the only tree generated by the GEG in Listing 5.2, where $x = \{boy, girl\}$.

```
extension S → ["boy", "girl"];

start extension S' to union S and S → "want" {
    :arg0 (1),
    :arg1 (2)
};
```

Listing 5.3: Example GEG (using inline notation).

6 Modeling AMRs

This chapter covers the results obtained from the empirical study, with the majority of the chapter focusing on modeling generic structures that exist in *The Little Prince* corups. The result is presented in Section 6.6, and the sections leading up to it covers generic structures. Furthermore, we present and prove two theorems regarding the set of minimum extension rules to model a semantic graph.

6.1 Fundamental Principles

The depth of an extension operation is the depth of its underlying graph. It is theoretically possible for an extension operation to have a depth greater than one, however, greater depth reduces the reusability of extension operations. Therefore, only extension operations with a depth of one are considered in this thesis. Moreover, an extension operation may contain more than one root node where each root node introduces a new concept. However, this also reduces the reusability of extension rules and therefore they will not be considered. Thus, Theorems 6.1 and 6.2 are only applicable to extension operations with a single concept and a depth of one.

Theorem 6.1 (Minimum Set of Extension Rules)**.** *Given an AMR A and a GEG $\Gamma = (\gamma, \mathcal{A})$, let C denote the set of all unique concepts in A. There must exist at least $|C|$ distinct extension rules in $\mathcal{A}$ for $\phi(A) \in L(\Gamma)$.*

Proof. The only way to add a new node to a graph in a GEG is by an extension operation, where the root node is added as a new node to the input graph. If there exists $|C|$ unique concepts in a graph, then there must exist at least $|C|$ extension operations that each introduce one concept. □

Additionally, given an AMR graph $A = (V, E, lab)$ and two arbitrary nodes $B, C \in V$, let E_1 and E_2 be the sets of labels of all outgoing edges of B and C, respectively. If E_1 and E_2 are equivalent, we say that B and C are *edge-label equivalent*. The root labels of B and C are not considered when determining if they are edge-label equivalent. If two or more nodes in an AMR are edge-label equivalent, then a single extension rule may represent them if the extension rule is allowed to use variables, such as in geglang.

Theorem 6.2 (Minimum Set of Variable Extension Rules)**.** *Given an AMR A and a GEG $\Gamma = (\gamma, \mathcal{A})$ where extension rules in $\mathcal{A}$ are allowed to use variables for their labels, and a set C consisting of all unique concepts in A. Let R be the subset of all unique nodes in A with respect to their outgoing edge labels, such that no two nodes in R are edge-label equivalent. It then holds that $|R| \leq |C| + 1$ and that there must exist at least $|R|$ distinct extension rules in $\mathcal{A}$ for $\phi(A) \in L(\Gamma)$.*

Proof. Extension operations are the only way of introducing edges when constructing a graph. Since each node in R represents a unique node with respect to its outgoing edge labels, it represents all outgoing edges possible in a graph. If there were less than $|R|$ extension operations,

it would not be possible to generate all edges. Thus, there must exist at least $|R|$ extension rules. To see why $|R| \leq |C| + 1$ holds, let $A = (V, E, lab)$ be a base graph where each outgoing edge label is unique. Since a concept can only have a fixed set of outgoing edge labels, there must exist $C = V$ unique concepts. Grouping all nodes with respect to their outgoing edge labels also yields $R = V$ (including nodes with no outgoing edges). If there were two nodes with the same concept in A, then the edges would also be identical, and thus both R and C would have one element removed. At the other end of the spectrum, consider a graph in which all nodes represent the same concept and all edges are identical. In that case, C would only contain one node and R would contain two nodes. This is because R must contain an operation with no edges and an operation that generates all edges. Thus, the inequality always holds. □

Consequently, the more edge-label equivalent nodes exist in an AMR A, the less amount of extension rules are needed to represent it. During variable expansion, $|R|$ extension rules are expanded to at most $|C| + 1$ extension rules.

6.2 Non-Reentrant Graphs

Non-reentrant graphs, or those where all nodes have exactly one incoming edge, are the easiest graphs to model in a graph extension grammar. Given an AMR A, one rule can be created in a GEG for each concept in A. Figure 12 illustrates an example AMR of depth four without reentrancies. A GEG that generates the AMR is given in Listing 6.1.

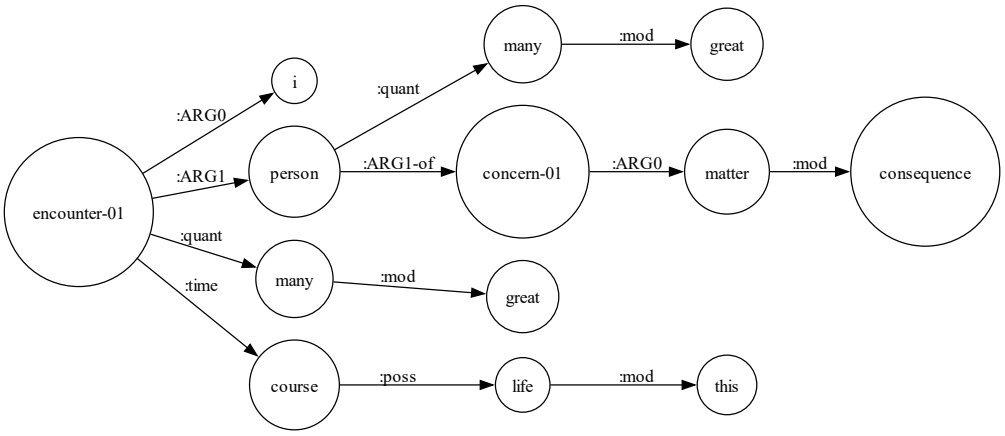

Figure 12: A non-reentrant AMR.

6.3 Reentrant Graphs

There are two ways to model reentrancies in a graph extension grammar, namely, using contextual nodes or docks and ports. Contextual nodes lead to more versatile rules and are preferred. The constraint in Definition 6.3 must be satisfied for an extension rule with a contextual node to be valid.

```
start extension S to
union Pronoun and QuantArg1Of and Mod and Poss
→ "encounter-01" {
    :arg0  (1),
    :arg1  (2),
    :quant (3),
    :time  (4)
};

extension Mod to Modifier → ["many", "life", "matter"] {
    :mod (1)
};

extension QuantArg1Of to union Mod and Arg0 → "person" {
    :quant   (1),
    :arg1-of (2)
};

extension Arg0 to Mod → "concern-01" {
    :arg0 (1)
};

extension Poss to Mod → "course" {
    :poss (1)
};

extension Modifier → ["great", "this", "consequence"];

extension Pronoun → "i";
```

Listing 6.1: GEG for Figure 12.

Definition 6.3 (Contextual Node Validity). *Let $\Gamma = (\gamma, \mathcal{A})$ be a GEG and $E_1 \in \mathcal{A}$ an extension operation where the root node has the label x. If there exists another extension operation $E_2 \in \mathcal{A}$ with a contextual node labeled x, then E_2 is valid if and only if there exists a tree $t \in L(\gamma)$ such that E_1 comes under E_2.*

As an example, consider the extension rule in Listing 6.2. For this rule to be valid, a node labeled "mystery" must be present in the evaluation of the partial derivation tree up to and including `B`.

```
extension A to B → "overpower-01" {
    :arg0   "mystery",
    :degree (1)
};
```

Listing 6.2: Extension rule with a contextual node.

As previously stated, instead of using a contextual node in Listing 6.2, it can instead be marked as a dock. However, it must be ensured that the input graph contains a port for that dock. For a reentrant graph, this can be achieved by marking the node defining a concept as a port as well so that two or more edges can reference it. If the node being referenced is not directly in the target rule of an extension rule, then using ports to forward references quickly becomes unfeasible. The only way to achieve this is using isolated nodes, as discussed in Section 6.3.1.

As an example, a simple reentrant graph is illustrated in Figure 13. Modeling a GEG with an extension rule for each concept yields the GEG in Listing 6.3. Since the node referenced by the contextual node is defined below in the tree it satisfies the constraint in Definition 6.3.

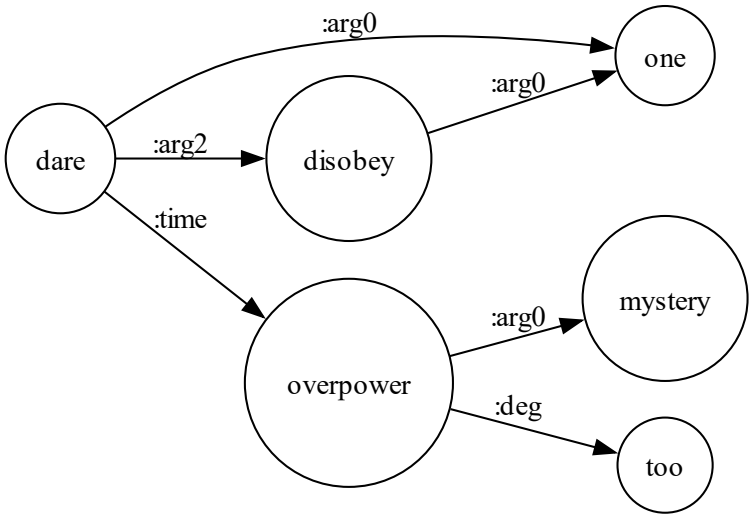

Figure 13: A simple reentrant AMR.

```
start extension S to union Arg0 and Arg0Degree → "dare-01" {
    :arg0 "one",
    :arg2 (1),
    :time (2)
};

extension Arg0 to Single → "disobey-01" {
    :arg0 (1)
};

extension Arg0Degree to union Single and Single → "overpower-01" {
    :arg0   (1),
    :degree (2)
};

extension Single → ["one", "mystery", "too"];
```

Listing 6.3: GEG for Figure 13.

An unavoidable side-effect of reentrancies is that they may lead to duplicated extension rules, with respect to the root node label and its outgoing edge labels. They are not true duplicates, but rather structurally similar. For example, consider the extension rule in Listing 6.4.

If there is a need for another extension rule that is identical, but with a contextual node instead of a dock, then a completely new extension rule must be defined. This quickly adds up when modeling complex graphs in a corpus. This may be avoided by extending the GEG formalism to allow child nodes to either be a dock or a contextual node.

```
extension A to B → "want-01" {
    :arg0 (1),
    :arg1 (2)
};
```

Listing 6.4: A simple extension rule.

6.3.1 Isolated Nodes

Figure 14 illustrates an AMR where isolated nodes are required to satisfy the constraint in Definition 6.3. The reason for this is because the referenced node "i" can either be defined under "fly-01" or "useful-05", but not both. Since both nodes need to reference it, it is not possible to model this graph conventionally.

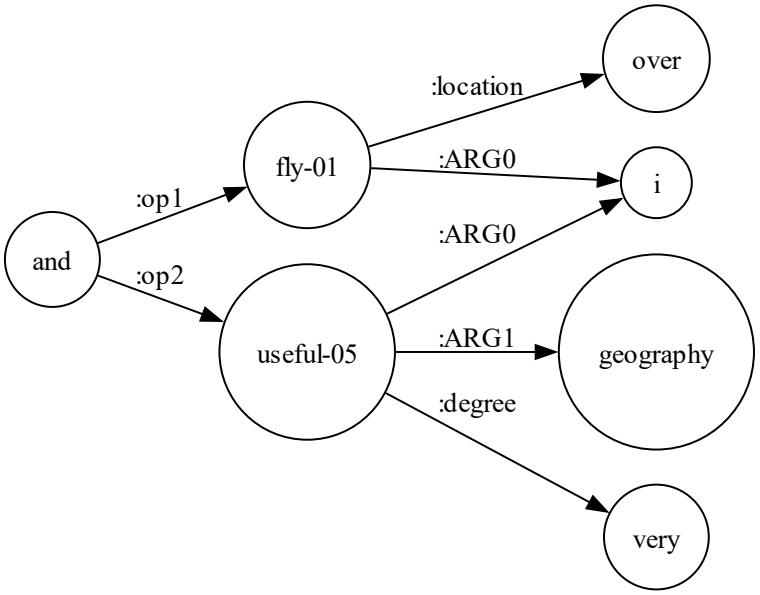

Figure 14: An AMR that necessitates the use of isolated nodes.

A solution to this problem is to let "i" be defined under "fly-01" and let "fly-01" be defined under "useful-05" in an isolated node. This allows one node to define "i" and the other to reference it with a contextual node. The node "useful-05" thus contains two ports and *propagates* "fly-01" to its correct placement in the graph. A GEG that can generate the graph in Figure 14 is shown in Listing 6.5.

It is important to note that this is a simplified case, and it quickly becomes complex to model this type of graph the larger it gets. The deeper the references are used in a graph, the more challenging it becomes to model them in a GEG. Similarly to only using contextual nodes, the use of isolated nodes leads to highly specialized rules that are unlikely to be reusable. It can be thought of as rules being duplicated, in the sense that outgoing edge labels from the root node are the same between two rules, but with additional isolated nodes that do not affect the semantic meaning behind the rule.

```
start extension S to Arg01Degree → "and" {
    :op1 (1),
    :op2 (2)
};

extension Arg01Degree to
union Arg0Location and Single and Single → "useful-05" [2] {
    :arg0   "i",
    (1) [1],     # Propagate Arg0Location to "op1"
    :arg1   (2),
    :degree (3)
};

extension Arg0Location to union Single and Single → "fly-01" {
    :arg0     (1),
    :location (2)
};

extension Single → ["i", "over", "geography", "very"];
```

Listing 6.5: A GEG for Figure 14.

6.4 Inverse Relations

The textual representation of an AMR may contain what appears to be cyclic relations. Consider the AMR in Figure 15 where there is an edge ":ARG1-of" that appears to create a cyclic graph. However, this is an inverse relation. What this means is that the edge from "i" to "endanger-01" is reversed, and instead the edge goes in the opposite direction. This appears to violate the specification of what an AMR is since they must contain exactly one root node, and inverting the edge in Figure 15 creates an additional root node. It is not clear how this is handled and is not mentioned in the specification. Furthermore, it is not possible to represent backward-facing edges in a GEG, those edges must be converted so that they are forward-facing. This is done by marking the target node as an additional root node and removing the "-of" suffix from the edge label.

```
(t / take-01
    :ARG0 (i / i
        :location (h / home)
        :ARG1-of (e / endanger-01
            :ARG0 (d2 / die-01
                :ARG1 i))))
```

Figure 15: What appears to be a cyclic AMR.

By inverting the relation of an *arg-of* edge, the original target node in Figure 15 becomes an additional root node. Figure 16 illustrates Figure 15 in which the inverse relation has been removed and *endanger-01* converted to an additional root node. It is possible to model a graph with multiple root nodes in a graph extension grammar, in which the initial extension rule must have the same amount of ports as root nodes in the graph.

If the goal is to model an entire corpus with a single GEG, as it is in this thesis, then it is not possible to model a graph with more than one root node. This is because the start rule

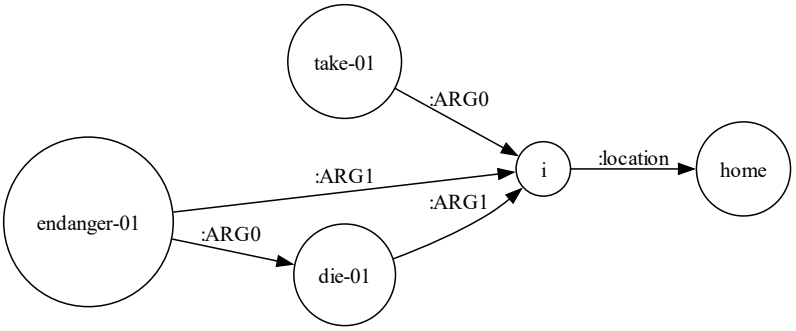

Figure 16: A multi-root acyclic AMR.

cannot have multiple types, thus rendering it impossible to mix graphs with one and two (or more) root nodes. Furthermore, when an additional root node is introduced via an inverse relation, the order of ports is non-deterministic per Definition 4.1. If there exists a graph with more than one inverse relation, the number of additional root nodes and combinations of port sequences would necessitate a large number of rules that are essentially duplicates. This is because there would need to exist one rule for each port sequence combination. Given $n \in \mathbb{N}$ root nodes, this would result in $n!$ combinations of port sequences and thus at least $n!$ rules to model the initial rule for *one* graph.

6.5 Literal Attributes

The AMR specification allows edges to literals that act as attributes, as discussed in Chapter 5. For example, there may exist an edge with the label "polarity" with an associated polarity literal. These attributes can be attached to a node to affect the semantic meaning behind the graph, without introducing additional concepts. Consider the extension rule in Listing 6.6, which represents the concept of wanting something.

```
extension A to B → "want-01" {
    :arg0 (1),
    :arg1 (2)
};
```

Listing 6.6: An extension rule with two outgoing edges.

The meaning behind the concept can be inverted by adding a polarity attribute. For example, if the edge ":polarity -" is added then it would mean to not want something. The only way to represent this in a graph extension grammar is to introduce it as a new concept as shown in Listing 6.7. This reduces the reusability of an extension rule since there needs to be a new extension rule for each combination of attributes, even if the concept is the same. Ideally, there should exist some mechanism in the formalism to mark such edges as optional.

```
extension A to B → "want-01" {
    :arg0     (1),
    :arg1     (2),
    :polarity (3)
};
```

Listing 6.7: An extension rule with polarity.

6.6 A Corpus-Generating GEG

A single GEG was made by creating an initial set of rules for the first AMR in the corpus. It was then extended for each successive AMR until all AMRs that could be generated were included. Two of the AMRs in the corpus contain inverse relations and as such could not be modeled by the GEG. As mentioned in Chapter 3, the entire corpus contains 433 unique concepts and 186 unique nodes with respect to outgoing edge labels. This sets the lower bounds of the number of extension rules that must be included in the GEG, as per Theorems 6.1 and 6.2, respectively.

The final GEG contains 290 variable extension rules and 2100 extension rules in total after variable expansion. In total, there are 2484 rules including union rules. The reason for additional rules besides the minimum required amount is due to "duplicated" rules, caused by contextual nodes and isolated nodes.

7 Conclusion

The goal of this thesis is to evaluate and explore the suitability of the graph extension grammar formalism for semantic graph generation. This is done by taking an existing corpus of AMR graphs, and hand-crafting a single GEG to generate all graphs in the corpus. The first research question, introduced in Chapter 1, is if it is possible to construct such a GEG. As discussed in Chapter 6, all graphs without inverse relations in the corpus were able to be modeled by a single GEG. Since the official specification states that all AMRs must be acyclic, one can conclude that it is indeed possible to construct a GEG that can generate all semantic graphs in a corpus. The discrepancy regarding cyclic graphs is discussed in Chapter 6.

The second research question is what corpus is suitable to use when evaluating the first research question. The chosen corpus, "The Little Prince", was chosen due to it being published by the authors of AMR. It contains all the features of the AMR specification, and has a wide variety of graphs that challenge the GEG formalism. The full details of the corpus are discussed in Section 3.1.

In Chapter 6 we lay out general structures present in the corpus and how to model them. We present further improvements to the GEG formalism to reduce the amount of excess work. The first improvement suggested is allowing child nodes in an extension operation to be either a dock or a contextual node. This would eliminate the need for "duplicated" rules when there is a need for two identical extension rules except that one contains a context node and the other a dock. The second improvement we suggest is that there should exist a mechanism to mark edges as optional to allow one rule to contain optional attributes such as polarity.

Furthermore, we discuss how the use of isolated nodes is required to satisfy the constraint in Definition 6.3 when it is not possible to model a graph conventionally. However, this leads to specialized rules that are unlikely to be reusable. The necessity of using isolated nodes makes it challenging and time-consuming to model a large AMR corpus. To conclude this thesis, it is possible to model a corpus by hand, and the GEG formalism can represent all semantic structures that can be represented by AMR. It is a suitable tool for semantic graph generation, but to model larger sets of graphs the formalism should be further improved.

References

[1] L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider, "Abstract meaning representation for sembanking," in *Proceedings of the 7th linguistic annotation workshop and interoperability with discourse*, 2013, pp. 178–186.

[2] J. Björklund, F. Drewes, and A. Jonsson, "Polynomial graph parsing with non-structural reentrancies," *arXiv preprint arXiv:2105.02033*, 2021.

[3] Y. Stade, "Language theoretic properties of graph extension languages: An investigation of graph extension grammars with context matching and logic," M.S. thesis, Umeå University, 2022.

[4] J. Feder, "Plex languages," *Information Sciences*, vol. 3, no. 3, pp. 225–241, 1971.

[5] T. Pavlidis, "Linear and context-free graph grammars," *Journal of the ACM (JACM)*, vol. 19, no. 1, pp. 11–22, 1972.

[6] F. Drewes and H.-J. Kreowski, "A note on hyperedge replacement," in *Graph Grammars and Their Application to Computer Science: 4th International Workshop Bremen, Germany, March 5–9, 1990 Proceedings 4*, Springer, 1991, pp. 1–11.

[7] A. Jonsson, "Generation of abstract meaning representations by hyperedge replacement grammars–a case study," M.S. thesis, Umeå University, 2016.

[8] F. Drewes, B. Hoffmann, and M. Minas, "Predictive top-down parsing for hyperedge replacement grammars," in *Graph Transformation: 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings 8*, Springer, 2015, pp. 19–34.

[9] H. Björklund, F. Drewes, and P. Ericson, "Between a rock and a hard place : Parsing for hyperege replacement dag grammars," in *Proc. 10th International Conference on Language and Automata Theory and Applications (LATA 2016) :*, ser. Lecture Notes in Computer Science, vol. 9618, 2016, pp. 521–532, ISBN: 978-3-319-29999-0. DOI: 10.1007/978-3-319-30000-9_40.

[10] F. Drewes and A. Jonsson, "Contextual hyperedge replacement grammars for abstract meaning representations," in *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms*, 2017, pp. 102–111.

[11] F. Drewes and B. Hoffmann, "Contextual hyperedge replacement," *Acta Informatica*, vol. 52, pp. 497–524, 2015.

[12] J. Björklund, F. Drewes, and A. Jonsson, "Generation and polynomial parsing of graph languages with non-structural reentrancies," *Computational Linguistics*, vol. 49, no. 4, pp. 841–882, 2023.

[13] E. Andersson, "Generating corpora of semantic graphs based on graph extension grammar," Bachelor's Thesis, Umeå University, 2023.

UMEÅ UNIVERSITY