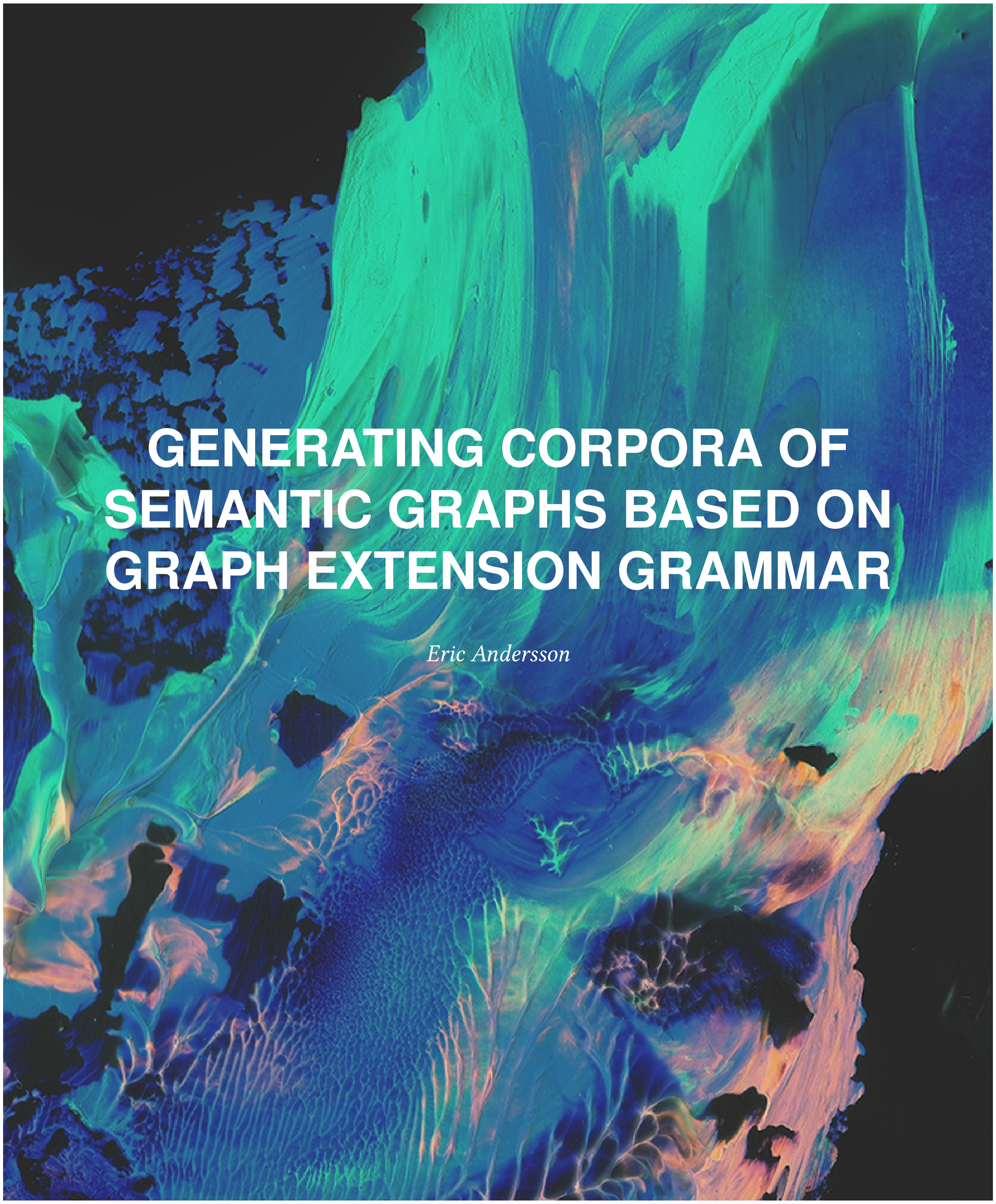
UMEÅ UNIVERSITY



GENERATING CORPORA OF SEMANTIC GRAPHS BASED ON GRAPH EXTENSION GRAMMAR

Eric Andersson

Bachelor 15 credits
Department of Computing Science

2023

Abstract

This thesis introduces the tool *Lovelace* which is used to generate corpora of semantic graphs to investigate which functionalities and design- as well as implementation aspects that are important in a corpus generator. *Lovelace* uses the graph grammar formalism graph extension grammar (*GEG*) to generate these corpora. A *GEG* consists of two parts, regular tree grammar (*RTG*) and graph operations. A tree generated by an *RTG* is used as an instruction of how the graph operations are applied to create a semantic graph. Since *Lovelace* can express variables as word classes the combination of semantic graphs and well-formed word classes means that the corpus generated by *Lovelace* is well-formed. In addition, *Lovelace* enables the user to configure parameters to specify the corpus generated. These corpora could be used as a tool to translate and process natural language. The thesis ends with a discussion about which parts are missing and what could be improved in the corpus generator, along with new insights into which functionalities are important for a user of a corpus generator.

Acknowledgements

This endeavour would not have been possible without the Foundation of Language Processing at Umeå University. I would like to express my deepest gratitude to Frank Drewes, Johanna Björklund and Anna Jonsson for believing in me and giving me the opportunity to create *Lovelace*. Throughout the entire process, I've been able to reach out to Anna Jonsson for feedback or knowledge, it was invaluable and I could not have asked for a better supervisor/colleague. Furthermore, I express my heartfelt gratitude to Mats Sjöberg for providing invaluable support on a personal level. Lastly, I extend my sincere appreciation to my fellow classmates in Computing Science, the class of 2020, for their friendship and support throughout these three years.

Contents

1 Introduction

One way of representing natural language is by using formal graph language [1] to model semantic graphs. This is because graph languages are both flexible and versatile when it comes to representing the human language. In a semantic graph, the nodes represent concepts and the edges represent relations between these concepts. An example of semantic graphs is *AMR*s [1].

There is a multitude of graph grammar formalisms that can represent natural language, each with its pros and cons. One of these graph grammars is graph extension grammar (*GEG*) [2] which is able to create a corpus of semantic graphs through the use of trees generated by a regular tree grammar in combination with graph operations. A corpus of semantic graphs is a structured resource of semantic graphs.

The goal of this thesis is to answer two questions:

1. What would be useful functionalities in a semantic graph corpus generator?
2. What are the requisite design considerations that must be taken into account for the successful implementation of a semantic graph corpus generator based on the graph extension grammar formalism?

To investigate these questions this thesis presents an implementation of a program that generates a corpus of semantic graphs by using *GEG*. The design decisions during the process of implementing the program are documented in the thesis. These decisions make it possible to analyze the pros and cons of the corpus-generating program. It also enables computer scientists to question which decisions were made and if there are any improvements to these decisions. The reason for creating the program is to create a tool that can automatically generate a corpus of semantic graphs.

The program is called *Lovelace* and can automatically generate a corpus of semantic graphs through the use of a *GEG*. *Lovelace* parses the trees into graphs by creating a graph for each tree based on the graph operations the tree represents. Each node in the tree must represent a graph operation in the given *GEG* grammar.

The idea is that *Lovelace* can enable the creation of semantic graph corpora researcher's field of interest which can assist them in solving natural language problems. It is also useful since there is no earlier implementation of a corpus generator based on *GEG* it leads to valuable information for future researchers that want to implement a corpus generator based on *GEG*.

Moving on to the structure of the thesis. Section 2 describes abstract meaning representation and explains the graph grammars *HRG* and *CHRG* which are closely related to *GEG*. Section 3 has two parts which include preliminaries which are required to understand *GEG* and a description of what *GEG* is. In Section 4 the implementation and design decisions of *Lovelace* are described in, the section ends with an example run of *Lovelace*. The aspects of implementing and designing *Lovelace* are discussed in Section 5 as well as a discussion of what could be viable implementations in the future. The thesis ends with a summary of the research question and how they were answered in Section 6.

2 Related work

This chapter introduces articles that are related to natural language generation. Langkilde and Knight [1] explain a specific type of semantic graph called abstract meaning representation (*AMR*) while Drewes and Kreowski [3] and Drewes et al. [4] describe the graph grammars *HRG* and *CHRG* which can be used to generate semantic graphs to represent natural language.

Langkilde and Knight [1] describe that there are two main methods for generating natural language, either using a natural language-generating template or a natural language generator (NLG). Templates are a static set of descriptions that are used to generate language objects over a specific domain while *NLG* can generate general-purpose and domain-independent natural language [1]. The advantage of templates is that they do not use any linguistic decision-making and do not consider complex knowledge resources and processing. There is no need for knowledge about gender, definiteness, tense, lexical items, etc [1]. The flaw with templates is that they are limited in what they can generate since they do not provide the flexibility, expressiveness, or scalability which is needed for a multitude of domains. Yet, before the introduction of *AMR* (which is a labelled directed graph that is used to represent semantics), templates were primarily used since the earlier *NLG*s were complex and required too many parameters which meant that it was easier to use templates. The earlier *NLG*s required sophisticated and large grammars, lexicons, ontologies, morphological tables, and collocation lists. When Langkilde and Knight [1] introduced *AMR* there was no longer a need for the excessive and extreme amount of knowledge that was required before the introduction of *AMR*. *AMR* made it possible to take in a string expressed as a meaning and use a lexicon combined with an *AMR* to generate a word lattice. Langkilde and Knight [1] also introduced their own *NLG* called Nitrogen which used *AMR*, but it was the introduction of *AMR* that had a larger impact on the field of natural language generation. The reason for this is that *AMR* made it possible to handle a large range of linguistic phenomena in a less complex way than previously.

Drewes and Kreowski [3] introduce the formalism *HRG* which is able to generate graph languages and thereby can handle *AMR*-based graph languages. As Drewes and Kreowski [3] explain in their paper, a hyperedge is an edge that has a fixed number of tentacles. Each tentacle connects to a node and therefore a hyperedge controls a sequence of nodes. These hyperedges can have both terminal and nonterminal labels. The hyperedge is replaced by a graph that matches the number of tentacles of the hyperedge. A graph that includes hyperedges is called a *hypergraph*. When no more nonterminals remain in the *hypergraph* the graph is in the language. This is very similar to context-free grammar, Drewes and Kreowski [3] provides a lemma to prove that *HRG* is context-free. This means that *HRG* is able to express context-free languages which are very powerful when it comes to natural language processing/generating.

Jonsson [5] proves that *HRG* is unable to generate the complete set of *AMR*s [1] over a given conceptual domain. Drewes and Jonsson [6] explains that the reason why *HRG* is unable to generate every possible *AMR* for some graph languages is that a hyperedge controls a fixed number of nodes (based on the number of tentacles) a derivation of a hyperedge leads to the

[1] The complete set of *AMR*s is every possible *AMR* that can be generated by the combinations of the concepts in a specific domain.

loss of control of the set of nodes any new hyperedge does not hang on to. When control of a set of nodes is lost there is no way to reach these nodes again since productions in *HRG* extend upon each other. Therefore Drewes et al. [4] presents a generalization of *HRG* called contextual hyperedge replacement grammar (*CHRG*). The generalization is done by extending context-free rules to contextual rules. The extension introduces the possibility of reaching back to nodes that are no longer in the set of controlled nodes, this is done by the introduction of contextual nodes. A contextual node is all nodes that are no longer in the set of controlled nodes. With the introduction of contextual nodes, the graph which replaced the hyperedge is able to nondeterministically choose any previously generated contextual node with a specific label that does not have to be in the set of controlled nodes of the hyperedge. These contextual nodes make it possible to represent all possible *AMR*s of a specific conceptual domain, it also reduces the size of the grammar since the grammar is able to use more than just the external nodes of the hyperedge when deriving the nonterminal of a hyperedge [4], thus increasing the expressiveness of a rule in *CHRG*.

3 Graph extension grammar

This section describes graph extension grammar which is a variant of the graph grammar *CHRG* [2]. An advantage of *GEG* compared to *CHRG* is that all edges of a node are introduced instantly when a node is created. Given the instant introduction of edges, it allows *GEG* to parse in non-uniform polynomial time. As explained in Section 1 *GEG* has two parts, graph operations and an *RTG* such as the one from Definition 3. An *RTG* is employed to construct derivation trees, with each tree node encompassing a single graph operation. A bottom-up traversal of the tree is executed to signify the sequence in which the graph operations are executed. The derivation tree can be perceived as an instructional guide on which order the graph operations should be applied. The result when the graph operations have been applied is a graph within the language of *GEG*. To enable a correct description of *GEG* it is required to explain some underlying concepts.

3.1 Preliminaries

Preliminaries consist of basic definitions to enable the definition of *GEG* formalism. These definitions are based on automata theory and discrete mathematics which are adapted from the ones in Björklund et al. [2]'s and Stade [7]'s papers. The set of natural numbers (including 0) is denoted by $\mathbb{N}$. The power set (a set of all subsets) of a set S is denoted by $\wp(S)$. All finite sequences of a set S are denoted by S^*, and all finite sequences with no element of S that occur more than once are written as $S^{\circledast}$. Both S^* and $S^{\circledast}$ contains the empty sequence ε.

The canonical extensions of a function $f\colon S \to T$ to $\wp(S)$ and S^* are denoted by function f. This means that $f(s_1 \cdots s_n) = f(s_1) \cdots f(s_n)$ for $s_1, \ldots, s_n \in S$, and $f(S') = \{f(s) \mid s \in S'\}$ for $S' \in \wp(S)$ [2].

To be able to understand the description of *GEG* there is a need to define the graph, tree, and the *RTG* used in this thesis.

Since graph grammars are based on the usage of graphs the first definition described is the definition of a graph. The graph grammar that is explored in this thesis is based on graphs that allow multiple edges (also known as multi-graphs), therefore the graphs defined in this section are multi-graphs.

Definition 1 (Graph). *A tuple $\mathbb{L} = (\dot{\mathbb{L}}, \bar{\mathbb{L}})$ of two finite sets $\dot{\mathbb{L}}$ and $\bar{\mathbb{L}}$ containing labels is a labelling alphabet. A graph over $\mathbb{L}$ is a quadruple $g = (V, E, lab, port)$, where*

- *V is a finite set of nodes,*
- *$E \subseteq V \times \bar{\mathbb{L}} \times V$ is a set of finite edges,*
- *$lab : V \longrightarrow \mathbb{L}$ is a labelling function for nodes, and*
- *$port \in V^{\circledast}$ is a finite amount of specially marked nodes.*

The definition requires a graph to have labelled nodes and labelled directed edges. In addition to the labels, some nodes are marked as *port* nodes. These marked nodes enable the graph operations, extension operation and disjoint union operation, which are explained in Section 3.2.

Given that the set of nodes V is finite, it implies that the alphabet $\bar{\mathbb{L}}$ is finite as well. If $\bar{\mathbb{L}}$ is finite then (by definition) the set of edges E is finite as well. A graph G has a type which is defined by $type(G) = |port|$ and the type τ of a graph is denoted by $\mathbb{G}_\tau$. An empty graph $(\emptyset, \emptyset, lab, \varepsilon)$ is denoted by ε.

Given a ranked alphabet $A = (\Sigma, rk)$ where Σ is a set of finite symbols, $rk = \Sigma \longrightarrow \mathbb{N}$. There is a function used to apply a rank to every $f \in \Sigma$, $f^{(k)}$ indicates $rk(f) = k$ and keep rk implicit we are able to define a tree.

Definition 2 (Tree)**.** *The set of all well-formed trees* T_Σ *over the ranked alphabet* Σ *is the minimum set of expressions such that for all* $f^{(k)} \in \Sigma$ *and every tree* $t_1, \ldots, t_k \in \mathrm{T}_\Sigma$ *we have:*

- $f[t_1, \ldots, t_k] \in \mathrm{T}_\Sigma$ *and*
- $f[]$ *for* $k = 0$, *which is abbreviated by* $f \in \mathrm{T}_\Sigma$.

Given the definition of trees, a set of trees can be defined with the help of a regular tree grammar (*RTG*). Therefore *RTG* is defined.

Definition 3 (Regular tree grammar)**.** *A regular tree grammar (*RTG*) is defined as a tuple* (N, Σ, P, S) *and includes the following components:*

- N *is a ranked alphabet of all symbols with rank 0, this alphabet is called* nonterminals.
- Σ *is a ranked alphabet which is disjoint with* N, *this alphabet is called* terminals.
- P *is a set of finite* productions *of the form* $R \longrightarrow f[R_1, \ldots, R_k]$ *where, for some values of* $k \in \mathbb{N}$, $f^{(k)} \in \Sigma$, *and* $R, R_1, \ldots, R_k \in N$.
- $S \in N$ *is the* initial nonterminal *of the* RTG.

To show an example of how a tree can be built through the use of an *RTG* we use the *RTG*:

1 $s \longrightarrow$ zs4(u)

2 $s \longrightarrow$ sx

3 $u \longrightarrow$ u(s' s)

4 $s' \longrightarrow$ s'x

The visualization of the generated tree is depicted in Figure 1

The symbol $\Rightarrow_x$ denotes the application of an *RTG* rule, where x indicates the specific rule that has been applied. The tree $zs4(u(s'xsx))$ is derived in the following way:

$$s \Rightarrow_1 zs4(u) \Rightarrow_3 zs4(u(s'\ s)) \Rightarrow_4 zs4(u(s'x\ s)) \Rightarrow_2 zs4(u(s'x\ sx))$$

Given these definitions, we are able to express graph extension grammar.

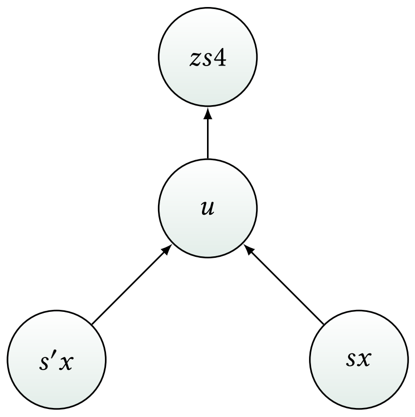


Figure 1: A visualization of a tree based on the tree string $zs4(u(s'x\ sx))$.

3.2 Description of graph extension grammar

To be able to extend the given graph $g = (V, E, lab, port)$ by applying graph operations one has to define an extension operation [2]. The extension operation is defined as a graph with the additional component $dock_\Phi$: Let

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

Before moving forward there is a need to introduce requirements for the extension operation. The extension operation is required to extend 'on top' of the underlying graph, similar to how the rules in *CHRG* are applied. This is required since the graph is built bottom-up. This also means that all nodes in the underlying graph are reachable from the ports. It also requires that only ports in the extension operation can have outgoing edges, ports in the underlying graph carry only receiving edges. This results in the following formal requirements [2]:

R1 When extending an underlying graph the source node of an edge has to be a new node (which also is a port) and the target node has to already exist in the underlying graph before the extension.

R2 The target node of an edge has to be a dock but can not be a port as well.

R1 ensures that all graphs generated by *GEG* are *DAG*s while *R2* requires all nodes within the graph to be reachable by the ports.

Given the graph $\underline{\Phi} = (V_\Phi, E_\Phi, lab_\Phi, port_\Phi)$ representing the *underlying graph* of the extension operation Φ [2]. The addition of the component $dock_\Phi$ transform Φ from a graph into an extension operation since $dock_\Phi$ makes it possible to connect the extension operation to a graph $G = (V, E, lab, port)$. To be able to apply an extension operation to an existing graph G it requires $G_\tau = |dock_\Phi|$ [2]. This is because $dock_\Phi$ are fused with $port_G$. Fusing means that the port nodes in $port_G$ take the position of the docks in $dock_\Phi$ in the resulting graph. If the dock node was both a dock- and port node the node in the resulting graph is a port node else it is a node without a port or dock. The application of the extension operation Φ on the graph G leads to a new graph:

$$\mathcal{G}' = (V_\Phi \cup V, E_\Phi \cup E, lab_\Phi \cup lab, port_\Phi)$$

There are also extension operations that fuse with contextual nodes, contextual nodes are nodes in the underlying graph that does not have a port. Contextual nodes can be expressed as $C = V/port$. The way the fusing is done between contextual nodes in Φ and contextual nodes in G is by matching the node name. If a contextual node with the same node name as the contextual node in Φ exists the nodes are fused. If multiple contextual nodes with that specific node name exist the contextual node is chosen in a non-deterministic way. The fusing between the contextual node in Φ and a contextual node is only possible if the contextual node has a node name (which is not the case when docks fuse with ports) [2].

The second operation is the disjoint union operation. This operation concatenates the port sequences of the argument graphs G and G' [2]. Let the graphs G and G' have the types τ and τ' where $\tau, \tau' \in \mathbb{N}$. This gives $G \in \mathbb{G}_\tau$ and $G' \in \mathbb{G}_{\tau'}$. A disjoint union operation over these argument graphs $\uplus_{\tau\tau'}(G, G')$ results in the graph $G_{\tau+\tau'}$ which is the result of making the disjoint union of the graphs G and G'. Although this sound simple there might occur problems if the argument graphs have nodes with the same underlying name (unique node number). How to handle this problem is discussed in Section 4.3.

The graphs $G \in \mathbb{G}_\tau$ and $G' \in \mathbb{G}_{\tau'}$, $\uplus_{\tau\tau'}(G, G')$ yields the graph in $\mathbb{G}_{\tau+\tau'}$ obtained by making the two graphs disjoint by a suitable renaming of nodes and taking their union [2].

An effect of the two types of graph operations is that the generated semantic graph does not have any cycles. The reason for this is that the definition of the graph used in *GEG* has directed edges combined with the fact that both the extension operation and disjoint union operation are unable to create cycles. The extension operation is always applying outgoing edges from the nodes in the operation to nodes in the underlying graph and the disjoint union never introduces any new edges. Since there are no cycles and the edges are directed in the generated graph the graph can be seen as a directed acyclic graph (DAG). Thus, all graphs generated by *GEG* are *DAG*s.

By considering the graph operations and Definition 2, a formal definition of *GEG* is introduced in Definition 4.

Definition 4 (Graph extension grammar)**.** *Given a tree grammar g over an alphabet $\mathcal{A}$ where every symbol corresponds to a graph operation we define* GEG *as a pair* $G = (g, \mathcal{A})$*:*

- *$\mathcal{A}$ is an algebra of graph operations and*
- *g generates well-formed trees over $\mathcal{A}$.*
- *Every tree generated from g is transformed into a graph by evaluation with respect to $\mathcal{A}$.*
- *Every tree generated by g is evaluated bottom-up.*

In Figure 5 an example of a *GEG* is shown. The *GEG* is an example of an $\mathcal{A}$ combined with the *RTG* depicted in Listing 4.1. Since the *RTG* include every possible graph operation in $\mathcal{A}$, $\mathcal{A}$ and g are seen as a correct pairing.

To show a use case of *GEG* we combine the *RTG* used as an example in Definition 3 with an $\mathcal{A}$ where the symbols represent the graph operations shown in Figure 2. Recall the tree $zs4(u(s'x\ sx))$ visualised in Figure 1. Since the tree was generated by the *RTG* it is in the language of the *RTG* which enables the *GEG* to derive the tree into a graph. Note that every operation used while deriving is in Figure 2 along with the name of the operation.

Since the evaluation of *GEG* is done bottom-up, the first operation applied is the extension operation *s'x*. *s'x* creates a port node with the node name x and port number one. The port is depicted by a number above the node (shown in Figure 2. When an operation is applied the tree node representing the operation is removed. The remaining tree and graph from applying *s'x* is shown in Figure 3(a). The second operation applied is another extension operation called *sx*. This operation creates a new graph that is identical to the graph created by *s'x*. The

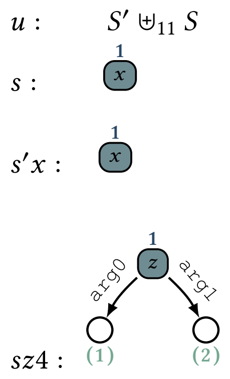
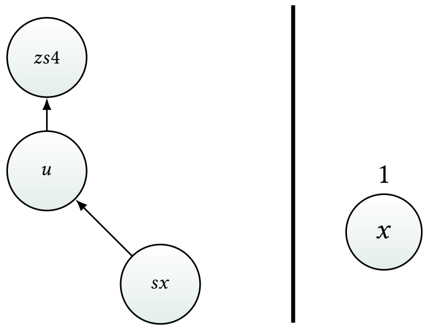
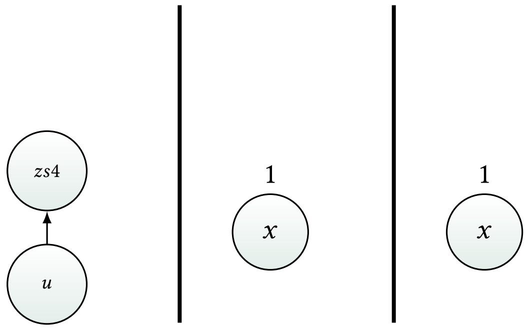


Figure 2: An illustration of the graph operations s′x, sx, sz4 and u where the variables are defined in the following way: $x \in \{\texttt{girl}, \texttt{boy}\}$ and $z \in \{\texttt{want}, \texttt{believe}\}$. Making the *GEG* able to generate semantic graphs that could for example express the sentence ´boy believes girl´. In the illustration one can see the ports represented as numbers above the nodes while the docks are represented under a port node on the format ($<$ *dock number* $>$).

two graphs and the remaining tree is shown in Figure 3(b). The third operation applied is a disjoint union operation *u*, which has the requirement of taking two underlying graphs in the state of the nonterminals *s'* and *s* depicted in Figure 2. Each argument graph is required to have strictly one port each indicated by the numbers next to the symbol $\uplus_{11}$ in Figure 2. The *u* operation combines the two underlying graphs into a single graph which represents the disjoint union of the graphs. In addition, all disjoint union operations concatenate the ports and make them into a sequence of ports where each port number is unique. The result of applying *u* to the underlying graph is depicted in Figure 4(a). The last operation is an extension operation *zs4* which is illustrated in Figure 2. The operation includes the port node *z* with port number one which has two outgoing edges to two dock nodes. The outgoing edges have the arguments *arg0* and *arg1*. The dock nodes are illustrated as nodes without labels and a number surrounded by parentheses that represent the dock number. The operation is applied to the underlying graph and the port numbers of the underlying graph are fused with the matching dock numbers in the extension operation. The result is shown in Figure 4(b). Given the definitions of the variables in Figure 2 the semantic graph in Figure 4(b) could express the sentence 'boy believes girl´. In addition, since the operations were able to connect their ports with docks in a correct fashion this means that the semantic graph is in the language of the *GEG*.

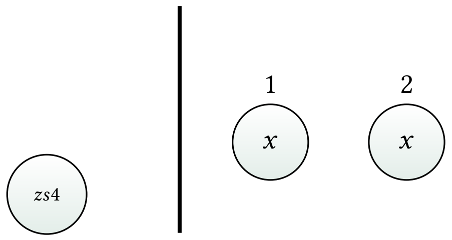


(a) First derivation step which applies the graph operation *s'x*. Showing the remaining tree with three nodes on the left side and the resulting graph of the applied operation on the right side.

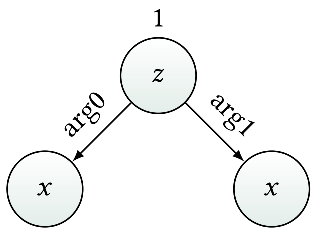


(b) Second derivation step applies the graph operation *sx*. Showing the remaining tree with two nodes on the left side, the graph from Figure 3(a) in the middle and a new graph created by the operation *sx* on the right side.

Figure 3: Visualization of the first and second derivation steps of the tree $zs4(u(s'x\ sx))$.



(a) Third derivation step applies the graph operation *u*. Showing the remaining tree with one node on the left side and a graph which is the result of the disjoint union of the two graphs depicted in Figure 3(b).



(b) Fourth derivation step applies the final graph operation *sz4*. This removes the last node from the tree and creates a new node in the graph called *z*. The operation also creates two outgoing edges from *z* to the *x* nodes with the labels *arg0* and *arg1*. The node *z* is also a port node with port number one

Figure 4: A depiction of the third and final derivation step of the tree based on the tree $zs4(u(s'x\ sx))$.

4 Generate corpus of semantic graphs with graph extension grammar

This chapter includes the methods and aspects used when designing and implementing *Lovelace*.

The process of creating the program has six parts: Section 4 describes the creation of the grammar files, Section 4.2 explains the parsing of input files, thereafter Section 4.3 exemplifies and details how a graph is built. Section 4.4 applies definitions to the generated corpus, Section 4.5 introduces the tool *Graphviz Online* which is used to visualize the semantic graphs and finally, Section 4.6 explain parameters to generate specific semantic graphs in the corpus. After the process of creating *Lovelace* has been described the chapter ends with an example run of *Lovelace* in Section 4.7.

4.1 Grammar creation

The first step of implementing *Lovelace* includes the creation of an *RTG* which was used to generate trees through the use of *BETTY*[1]. [8]. To be able to apply graph operation the decision was made to let each label of each tree node represent a production rule in the specific input *GEG*. The *GEG* that the label of the tree nodes is based on is Björklund et al. [2]'s grammar which is depicted in Figure 5. This *GEG* enable the creation of semantic graphs that can express sentences such as ´boy persuaded the girl to believe the boy´. Note that the graph operations in Figure 2 is a subset of the graph operations in Figure 5.

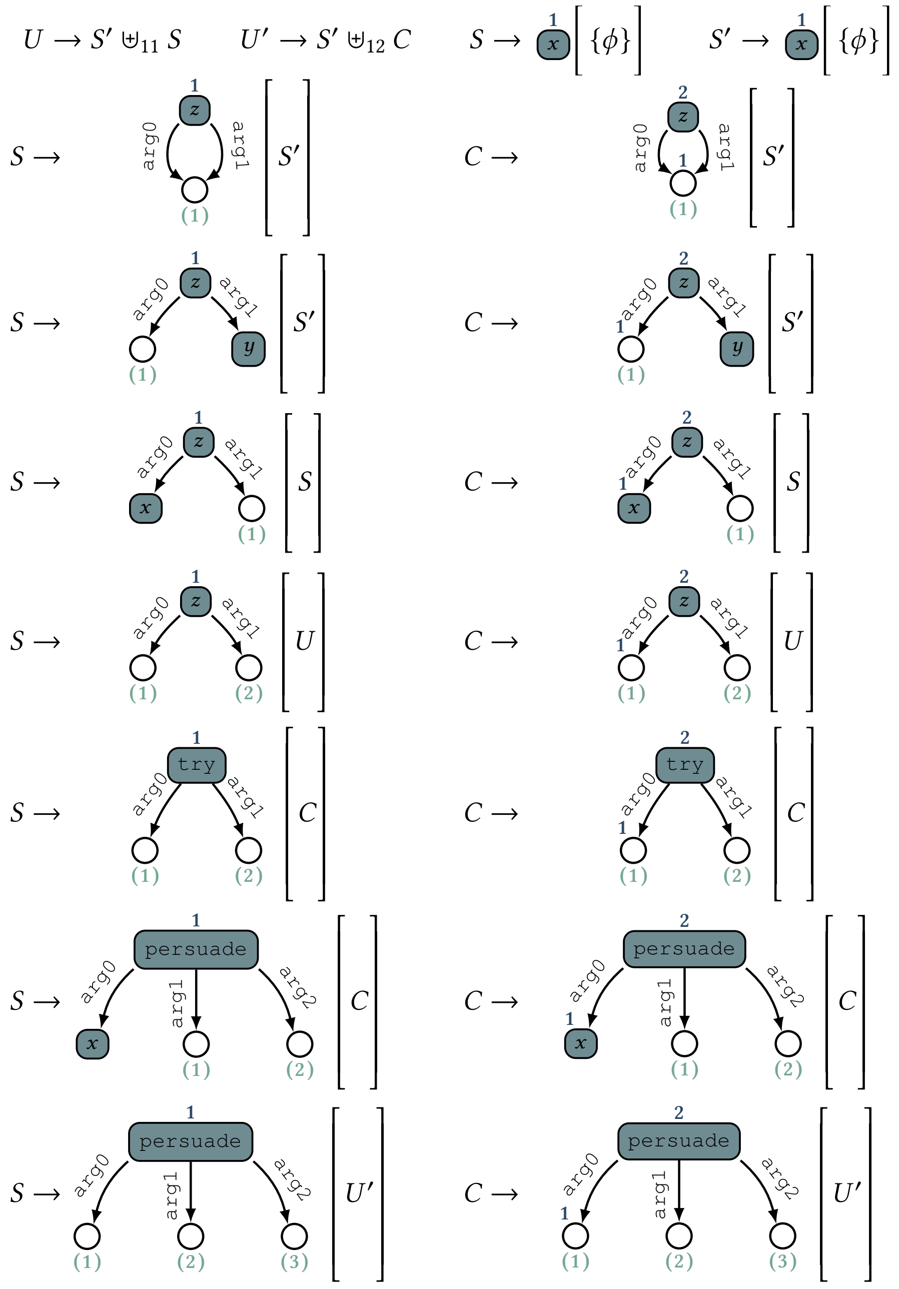
Each operation in the *GEG* shown in Figure 5 is translated into an operation name based on the nonterminal on the left-hand side combined with the name of the root node. For example the operation in Figure 6 would be given the operational name *zs1'*. This is because the root node of the extension operation is z and the left-hand nonterminal. The number 1 represents that it is the first occurrence of the combination of left-hand nonterminal and that specific root node in the *GEG*.

The *RTG* is written in a text file and structured after the requirement of *BETTY* [8]. The text file is shown in Listing 4.1.

```
s
s -> sx
s -> zs1(s')
s -> zs2(s')
s -> zs3(s)
s -> zs4(u)
s -> trys(c)
s -> persuades(c)
s -> persuadess(u')

c -> zc1(s')
```

[1]BETTY is a tool that generates a tree corpus based on an *RTG*.



$U \longrightarrow S' \uplus_{11} S$
$U' \longrightarrow S' \uplus_{12} C$
$S \longrightarrow$ x $\{\phi\}$
$S' \longrightarrow$ x $\{\phi\}$
$S \longrightarrow$ z arg0 arg1 (1) S'
$C \longrightarrow$ z arg0 arg1 (1) S'
$S \longrightarrow$ z arg0 arg1 y (1) S'
$C \longrightarrow$ z arg0 arg1 y (1) S'
$S \longrightarrow$ z arg0 arg1 x (1) S
$C \longrightarrow$ z arg0 arg1 x (1) S
$S \longrightarrow$ z arg0 arg1 (1) (2) U
$C \longrightarrow$ z arg0 arg1 (1) (2) U
$S \longrightarrow$ try arg0 arg1 (1) (2) C
$C \longrightarrow$ try arg0 arg1 (1) (2) C
$S \longrightarrow$ persuade arg0 arg1 arg2 x (1) (2) C
$C \longrightarrow$ persuade arg0 arg1 arg2 x (1) (2) C
$S \longrightarrow$ persuade arg0 arg1 arg2 (1) (2) (3) U'
$C \longrightarrow$ persuade arg0 arg1 arg2 (1) (2) (3) U'

Figure 5: A (graphical representation of a) label-matching graph extension grammar demonstrating how to generate semantic graphs. Instead of having multiple productions that are the same except for the name of a node the decision was made to use variables [2]. For this specific *GEG* the variables are defined in the following way $x \in \{$`girl`, `boy`$\}$, $y \in \{$`girl`, `boy`, `want`, `believe`, `try`, `persuade`$\}$, and $z \in \{$`want`, `believe`$\}$. Note that the graphs produced by the nonterminal S always generate one port while the nonterminal C always produce two ports.

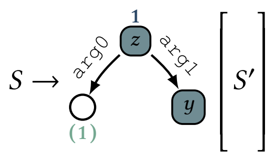


Figure 6: A graphical representation of a production rule in a graph extension grammar demonstrating how to generate semantic graphs

```
12 c -> zc2(s')
13 c -> zc3(s)
14 c -> zc4(u)
15 c -> tryc(c)
16 c -> persuadec(c)
17 c -> persuadecc(u')
18
19 u' -> u'(s' c)
20 u -> u(s' s)
21
22 s' -> s'x
```

Listing 4.1: An *RTG* based on the *GEG* in Figure 5 where each right hand side is represented by an extension- or disjoint union operation. The first nonterminal in the file represents the starting nonterminal. The union (u and u') nonterminals are the only nonterminals which include nonterminals that appear on the left and side on the right-hand side.

The graph operation file is created by translating the graph operations in Figure 5 into a text file which is made parsable by the use of the keywords that are defined in the following way:

OPERATION is used to indicate the name of the operation. The name is translated in the same way as the *RTG* file in Listing 4.1.

node is represented by an internal node number, followed by a label and a port number (if the node has a port). If the node does not have a port it only has a label and if the node has no name it is indicated by setting the name of the node to *undef*. An example of a node with a port can be seen in Listing 4.1 on row number 2, while a node without a port can be seen on row number 19 in the same Figure.

dock is represented by an internal node number followed by the specific dock number for that dock. This can be seen on row 11 in Listing 4.1.

edge is depicted as an internal node number followed by an arrow pointing towards another internal node number, thereafter the label of the edge is described. An example of an edge is seen on row 13 in Listing 4.1.

Note that a node can have strictly one or zero ports or docks. A sample of the *GEG* file is given in Listing 4.1.

```
1 OPERATION: sx {
2       0[label="x", port=1
3 }
4
```

```
OPERATION: s'x {
      0[label="x", port=1]
}

OPERATION: zs1 {
      0[label="z", port=1]
      1[dock=1]

      0 -> 1 [label="arg0"]
      0 -> 1 [label="arg1"]
}

OPERATION: zs2 {
      0[label="z", port=1]
      1[label="y"]
      2[dock=1]

      0 -> 1 [label="arg1"]
      0 -> 2 [label="arg0"]
}


OPERATION: zs3 {
      0[label="z", port=1]
      1[label="x"]
      2[dock=1]

      0 -> 1 [label="arg0"]
      0 -> 2 [label="arg1"]
}
```

Listing 4.2. A sample of the textual representation of graph operations. The complete textual representation can be found in the Listing 6 in the appendix.

4.2 Parsing of input

In *Lovelace* there are two instances of parsing, the first one is when the graph operations (depicted in Listing 4.1) are parsed into usable data. The second instance is when one or more trees in the tree file that can be derived from the *RTG* in Listing 4.1 are parsed into tree objects.

The parsing of graph operations is done by traversing the text file in Listing 4.1. For every occurrence of the keyword OPERATION, an object called Operation is created which includes the nodes and edges specified in the text file. An example of the parsing of operation *zs3* (shown in Figure 7) is given in Figure 8.

As shown in Figure 8 for every occurrence of the word OPERATION a new operation is created which includes an operation name, a list of nodes, and a list of edges. The nodes and edges include their own attributes. Nodes include a node name, a node number (which is unique for every node), a port number, and a list of docks where the node is going to create an edge. Edges include the label of the edge, start- and endpoint of the edge.

A problem that has to be taken into consideration is the distinction between extension

```
OPERATION: zs3 {
    0[label="z", port=1]
    1[label="x"]
    2[dock=1]

    0 -> 1 [label="arg0"]
    0 -> 2 [label="arg1"]
}
```

Figure 7: A textual representation of the graph operation zs3

```
Node{
    nodeName = z
    nodeNumber = 0
    portNumber = 1
}
Node{
    nodeName = x
    nodeNumber = 1
}
Node{
    nodeNumber = 2
    dockNumber = 1
}

Operation{
    operationName = zs3
    nodeList = [0, 1, 2]
    edgeList = [{0 -> 1 arg0}, {0 -> 2 arg1}]
}
```

Figure 8: A translation of the textual representation of the operation sz3 into usable data. The translation is given as a pseudo-code.

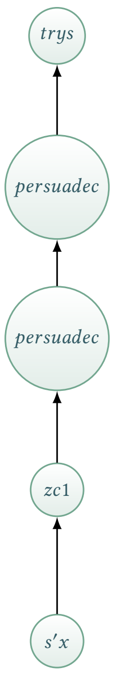


Figure 9: A visualization of the resulting tree based on the tree string $trys(persuadec(persuadec(zc1(s'x))))$

operation and disjoint union operation. Section 3.2 describes that extension operations extend the underlying graph and are able to connect to contextual nodes while disjoint union operations combine two argument graphs into one graph which is the disjoint union of two argument graphs. If the program is unable to decide whether an operation is an extension operation or a disjoint union operation it is not able to generate correct semantic graphs. This problem is addressed during the parsing of the graph operation file by identifying that a row in an operation that exclusively consists of two integers separated by a space (representing the number of port nodes in each underlying graph) has to be a disjoint union operation since that type of row is not allowed in the definition of an extension operation. If it is not a disjoint union operation it is by default an extension operation.

The second parsing in *Lovelace* occurs when a tree based on the *RTG* in Listing 4.1 is parsed. The parsing method is inspired by how *BETTY* [8] parses an *RTG* into multiple trees. The parsing is done recursively starting from the bottom of a tree and traversing the tree until the entire tree has been parsed into a tree object in *Lovelace*. The tree is completely parsed when the root of the tree has been found. This means that the algorithm returns the root of the tree object. The root has references to its children and the children have a reference to their parent node. The references from parent to child are not important when the tree object has been built therefore these references are not illustrated in any figure in this thesis. The tree object is used to decide in which order each of the graph operations is applied. The semantic graphs in *Lovelace* are built bottom-up by starting from a leaf node and applying graph operations (based on the name of the current node) until the root operation is executed. The entire algorithm of the tree building is depicted in Algorithm 1.

The depth of the tree is indicated by left- and right-handed parentheses. If a left-handed parenthesis is encountered while traversing the tree the depth of the tree is incremented by one and the other way around with a right-handed parenthesis. The example tree $trys(persuadec(persuadec(zc1(s'x))))$ is illustrated in Figure 9.

When the string representation of a tree has been parsed into a tree object the tree is used

Algorithm 1 BuildTree

```
if !$ContainsParsingSymbol(rhs)$ then
    $tree$ ← new TreeNode(rhs)
    if !$TreeNode.contains(tree)$ then
        $treeNodes.add(tree)$
    end if
    return $tree$
else
    $length$ ← length of $rhs$
    $parStack, currentString, treeString, child, tree$ ← empty
    $children$ ← empty LinkedList of TreeNodes
    for $i$ ← 0 to $length$ do
        $c$ ← $i$th character of $rhs$
        if $c$ = '(' then
            $treeString$ ← substring of $rhs$ from 0 to $i$
            $done, parStack.push(c)$ ← false
            while not $done$ do
                $i, c$ ← $i + 1$, $i$th character of $rhs$
                if $c$ = '(' then
                    $parStack.push(c)$
                    if size of $parStack$ > maxDepth then
                        setMaxDepth(size of $parStack$)
                    end if
                else if $c$ = ')' then
                    $parStack.pop()$
                end if
                if parStack.empty() or ($c$ = ' ' and $currentString \neq$ empty and size of $parStack$ = 1) then
                    $tempTree$ ← buildTree($currentString$, 0)
                    $children.addLast(tempTree), currentString$ ← empty
                else
                    $currentString$ ← $currentString + c$
                end if
            end while
        end if
    end for
    $size$ ← size of $children$, $tree$ ← buildTree($treeString$, $size$)
    while not $children$ is empty do
        $tree.hasChild(true)$
        $child$ ← first element of $children$, $children.removeFirst()$
        $child.setParent(tree), tree.addChild(child)$
    end while
    return $tree$
end if
```

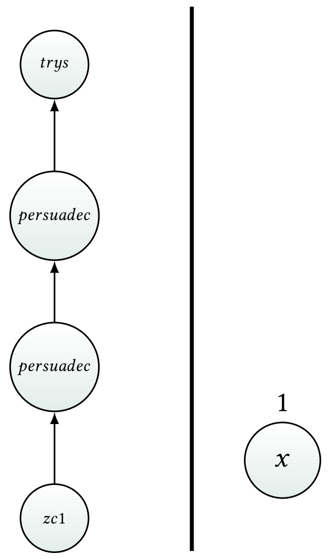
to build the semantic graph.

4.3 Graph building

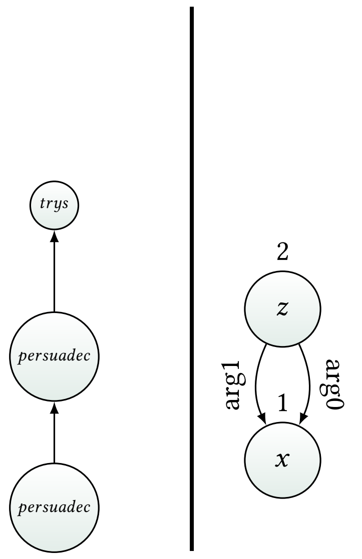
Graph building explains the process of applying graph operations in an order given by trees such as the tree shown in Figure 9. To show the derivation from a tree to a semantic graph this section includes a step-by-step derivation based on a tree with purely extension operations. This Section also discusses which problems one can face when deriving a tree that includes disjoint union operations. To make it more clear for the reader the decision was made to remove a tree node from the tree in every derivation step. In the implementation the tree nodes are never removed, they are merely traversed.

4.3.1 Derivation of extension tree

In Figure 10 the first two steps of the transition from tree to graph are depicted. In Figure 10(a) The graph operation *s'x* is removed from the tree and used to create a single port node *x* with port number one. Figure 10(b) removes the node *zc1* and applies its operation onto the node *x*. When applying the operation *zc1* the docks of the node *z* are connected to the port of the node *x* which creates two edges with the labels *arg0* and *arg1*. When the operation *zc1* is applied two new ports are created, port one on node *x* and port two on node *z*. This means that the next operation that is applied to the graph must have at least two docks with dock numbers that are the same as the port numbers (given how *GEG* is defined).

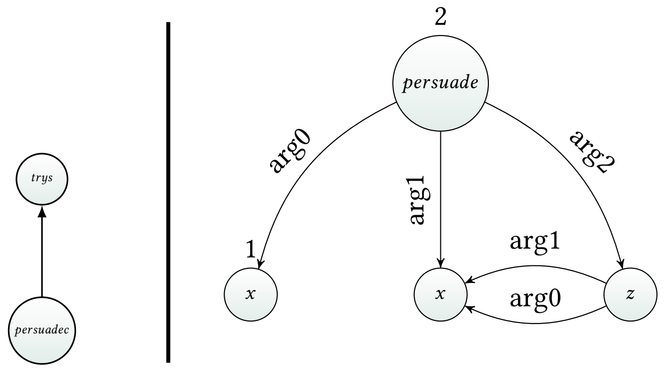


(a) First derivation step which applies the graph operation *s'x*. Showing the remaining tree on the left side and the resulting graph of the applied operation on the right side.

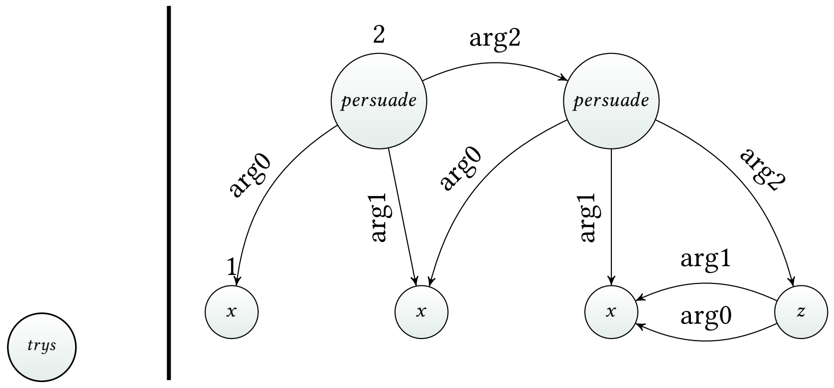


(b) Second derivation step, applying the graph operation *zs1* on the graph shown on the right side in Figure 10(a). This results in the remaining tree with three nodes on the left-hand side and the ensuing graph on the right-hand side.

Figure 10: An illustration of the two first derivations steps of the tree $trys(persuadec(persuadec(zc1(s'x))))$. The tree is derived into a *DAG* by applying graph operations. Each step removes a node from the tree and applies the graph operation which is denoted in the removed tree node



(a) Third derivation step, applying the graph operation *persuadec* on the graph shown on the right side in Figure 10(b). This results in the remaining tree with two nodes on the left-hand side and the ensuing graph on the right-hand side.



(b) Fourth derivation step, applying the graph operation *persuadec* on the graph shown on the right side in Figure 11(a). This results in the remaining tree with one node on the left-hand side and the ensuing graph on the right-hand side.

Figure 11: Depicting the third and fourth steps of the derivation of the tree $trys(persuadec(persuadec(zc1(s'x))))$.

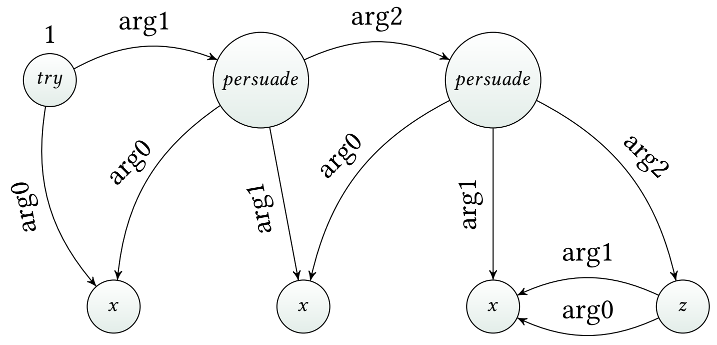


Figure 12: The resulting DAG after deriving the regular tree $trys(persuadec(persuadec(zc1(s'x))))$. The resulting DAG is created by applying the graph operation *trys* on the graph shown on the right side in Figure 11(b).

Figure 11 represent step three and four of the derivation. In the third step, the node *persuadec* is removed from the tree and the graph operation with the same name is applied. The operation creates two new nodes called *persuade* and *x*. Persuade has three outgoing edges, the first edge with the label *arg0* connects to the newly created node *x*. The second edge with label *arg1* connects to node *x* since the port number matches the dock number. Lastly, the third edge with label *arg2* connects to node *z* and since the port number match with the dock number. The operation *persuadec* introduces two ports, a port with port number two at the node *persuade* and a port with port number one at the newly created node *x*. The resulting graph is shown in Figure 11(a). The fourth derivation step removes another node with the name *persuadec*, this means that the same operation is applied again but it is connected to a different node since the port placement has changed. *persuadec* creates a new node called *persuade* and *x*, where persuade has three outgoing edges. The node persuade creates an edge to the newly created node *x* in the same way as the earlier usage of the operation *persuadec*. The edge with label *arg1* connects to the port with port number one in the underlying graph and the edge with label *arg2* connects to the port with port number two in the underlying graph. Once again *persuadec* introduces two ports for the newly created nodes, port number one at node *x* and port number two at node *persuade*.

The final graph is shown in Figure 12. This graph is the result of removing the last node in the tree and applying the graph operation with the same name as the node to the graph in Figure 11(b). The operation creates a new node called *try* which has two docks. These docks create edges by fusing with the matching ports. This creates two outgoing edges, one to the node *persuade* and one to node *x*. The operation also introduces a new port at the node *try* with node number one.

4.3.2 Problems when applying disjoint union operations

Since a derivation of a tree including a disjoint union operation is already exemplified in Section 3.2 we instead address important implementation details. One problem that has to be addressed is how the renaming of nodes is handled when using the disjoint union operation which was discussed in Section 3. The need to rename nodes is based on the idea that the nodes do not have a unique id [2], however in the implementation of *Lovelace* every node created from a graph operation is given a unique id. This means that there is no need for renaming since every node is unique. Although the renaming problem does not affect *Lovelace* there is another problem that the disjoint union creates. In Björklund et al. [2] the concatenation automatically change the numbering of the ports in an incremental order.

Since each operation in the graph grammar depicted in Figure 3 has internal port numbers a problem arises when applying the disjoint union operation over two argument graphs with the same port numbers. The solution that is used in *Lovelace* is to increment the port numbers generated by the right-hand child of the union node in the tree. Let $|childport_{lhs}|$ represent the number of ports generated by the left-hand child of the union node in the tree. Given $|childport_{lhs}|$ the new port numbers for the ports generated by the right-hand child are calculated by $portnumber_{rhs} - |childport_{lhs}|$ for each port in the right-hand side operation.

To make the DAG in Figure 11(b) understandable there is a need to define the variable x, z, and the unused variable y.

4.4 Instantiating the generated corpus

There are two types of graphs described in this section, the graphs where the definitions have not been applied are called uninstantiated graphs and the graphs where the definitions have been applied are called instantiated graphs. Each variable needs one or more definitions. Definitions are a set of words (preferably in the same word class) that replace the variable.

Before describing the way these variables are instantiated it is necessary to discuss why they exist. The idea is that each variable is able to represent a word class in a semantic word bank. An example of such a semantic word bank is *Unified Verb Index* [9] which is a combination of the word banks *VerbNet* [10], *PropBank* [11], *FrameNet* [12] and *OntoNotes* [13]. These word banks are able to identify different properties of words. The behaviour of a word class is defined in a multitude of ways depending on how the word class is used. Take the word class *want-32.1-1-1* (defined in verbNet [10]) which includes the words {`desire, like, need, prefer, want`}. It should be noted that a word can belong to multiple word classes based on its usage. There are two entities used in the word class. The first one is *agent* which represents the entity that has an impact on an entity, the second one is patent which is the impacted entity. The word class *want-32.1-1-1* is defined using four arguments [11]:

Arg0-PAG Defines the agent.

Arg1-PPT Defines the patient.

Arg2-GOL used to represent the *beneficiary* in the word class *want-32.1-1-1.*

Arg3-PPT *in-exchange-for*, this argument explains that there is an exchange that the agent has to do.

Arg4-DIR represents the argument *from.*

Given the defined arguments of a specific word or word class, one can build graph operations adapted for the specific behaviour of word classes. This leads to the ability to create semantic graphs without ambiguity since a word can appear in multiple word classes depending on how it is used.

One thing to keep in mind is that *Lovelace* is not strictly bound by the English language. If one would like to represent another language it would require the usage of semantic word banks which are able to represent that specific language.

The way that Lovelace handles definitions is by the use of a text file that includes definitions for each variable. The definition file for the *GEG* in Figure 5 is represented in Figure 13.

```
z = want believe
x = boy girl children boys girls they he she child
y = boy girl children boys girls they he she child want believe persuade try
```

Figure 13: Definitions of the variables z, x and y

These definitions are used to iterate over the *DAG* and create a new graph for each possible combination of node definitions. Given the definitions in Figure 13 and an uninstantiated *DAG* with the variables z, z, x and y generates $|z| \cdot |z| \cdot |x| \cdot |y| = 2 \cdot 2 \cdot 9 \cdot 13 = 468$ instantiated graphs.

One might think that these variables represent random letters but the idea is that each variable represents different word classes.

4.5 Graph visualization

```
digraph G  {
        1 -> 0 [label="arg1"]
        1 -> 0 [label="arg0"]
        2 -> 0 [label="arg1"]
        2 -> 1 [label="arg2"]
        2 -> 0 [label="arg0"]
        3 -> 0 [label="arg0"]
        3 -> 2 [label="arg1"]

        1 [label="want"]
        0 [label="boy"]
        2 [label="persuade"]
        3 [label="try"]
}
```

Figure 14: A text file representing a *DAG* on a format that is able to generate a visual representation of that DAG through the usage of *Graphviz Online*.

Each graph is represented as a text file, the text file has a specific format called Digraph (depicted in Figure 14) which makes it possible to create an image of the graph through the use of the tool *Graphviz Online* [14] [2]. In the depiction, each node is represented by its unique node number combined with a label. An edge is represented by the node number of the outgoing node followed by an −> pointing at the node number of the receiving node. Thereafter the label indicates which argument the edge represents.

When using the text file in Figure 14 to generate a visualization of the DAG the tool *Graphviz Online* is used. The resulting graph can be seen in Figure 15. The reason for using *Graphviz Online* to visualize the semantic graphs is quite simple. It is an easy-to-use tool that

[2]To use *Graphviz Online* one can follow the link https://dreampuf.github.io/GraphvizOnline/ and insert the graph file on the left-hand side. After that *Graphviz Online* generates a visualization on the right-hand side.

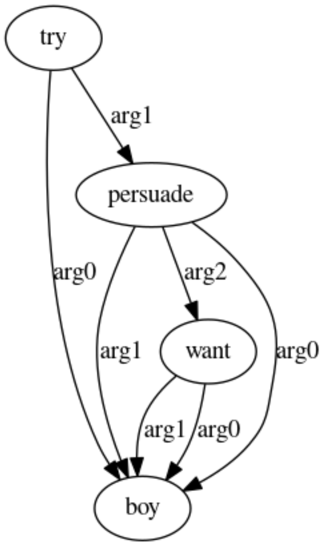


Figure 15: A visualisation of the file depicted in Figure 14. The visualization is a semantic graph created using the tool *Graphviz Online*.

is able to generate every required attribute of the semantic graphs [14].

4.6 Parameters

Lovelace has two obligatory parameters which are the flags `-g` and `-t`. The flag `-g` is used to provide a grammar file with graph operations. The operations are in the same format as the operations depicted in Figure 4.1. The flag -t provides the program with a file that includes one or more regular trees which are used to derive *DAG*s. In addition to the obligatory flags, there exist optional flags, these flags are defined in the following way:

`-L <min num nodes>` Set the minimum number of nodes in the tree. If L is set to 4 the program is going to skip every tree that has less than 4 nodes.

`-H <max num nodes>` Set the maximum number of nodes in the tree. Similar functionality as L but sets an upper node bound instead of a lower one.

`-d <definition file>` This file include definitions of symbols such as x,y,z. An example of a definition could be `x = {boy, girl, them}`.

`-k <operation name>` generates every graph which is based on a tree that includes that specific operation name. An example of a key operation could be the *persuadec*.

These optional flags make it possible to filter out interesting trees from a large tree corpus. One thing to keep in mind is that the usage of the d-flag drastically increases the workload of the program since every generated graph creates a multitude of instantiated graphs depending on how many undefined nodes are used in the graph. If the d-flag is not used the program generates an uninstantiated graph with the variables instead of the definitions of the variables. A problem with the usage of the d-flag is that the user is given every possible definition even if the user is just interested in a few definitions. It is a problem since it leads to unnecessary computations and an overflow of information which makes it difficult for the user to find relevant information.

4.7 Execution of Lovelace

Given the input of the operation file called `operations.txt` shown in 5, a tree file `tree.txt` with a single tree *persuades*$(sz1(sx))$, and a definition file `definitions.txt` with the definitions `x = {girl}, z = {believe}` *Lovelace* is executed by the following command: `java lovelace.java -t tree.txt -g operations.txt -d definitions.txt`. The corpus created by the input is a single semantic graph shown in Figure 16 and can be translated into the English sentence ”the girl persuaded herself to believe herself”.

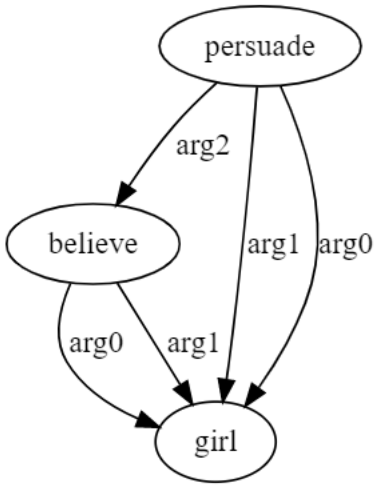


Figure 16: The semantic graph created by the example run of Lovelace.

5 Discussion

The aim of this thesis was to identify aspects that are important when designing a corpus generator based on *GEG* as well as identify important functionalities of a semantic graph corpus generator. This chapter is partitioned into Section 5.1 and 5.2, where Section 5.1 discusses the vital design- and implementation aspects in a semantic graph corpus generator and Section 5.2 discusses the functionality of *Lovelace*.

5.1 Design and implementation aspects

Starting with the grammar creation, since *GEG* consists of both trees and graph operations both had to be specified. The choice to create an *RTG* to let *BETTY* [8] generate the trees for the tree file that is used in the tree file enables the option to combine *Lovelace* and *BETTY* in the future.

Since the generated trees include the name of graph operations (and their nonterminals) an *RTG* and a file specifying the graph operations can be seen as a pair. The *GEG* would not be able to translate the tree nodes into operations if either the *RTG* which was used when generating trees or the graph operation file would be exchanged for another file (given that the new *RTG* or operation file does not include the same operations). An idea to remove the pairing of *RTG* and graph operations was explored. The idea included the addition of nonterminals in the operation file which makes it possible to generate an *RTG* based on the operation file, thus removing the requirement of an *RTG* file in the *GEG*. Although it is very much possible to implement, it makes the operation file more complex since every pair of nonterminals needs to be specified. In addition, one could argue that it is more intuitive with a pairing of input files since *GEG* is defined as a pair of tree grammar and graph operations according to the Definition 4.

The second aspect of implementing *Lovelace* is the parsing of input files. Since *GEG* every tree in the tree file used as input is based on a specific *RTG* which match the graph operations used in the operation file the operation file only has to be parsed once while each tree has to be parsed into a tree object. Although this solution might seem logical it is inefficient in some cases. An example of inefficiency is if a user of *Lovelace* wants to generate a semantic graph based on a single tree including 3 or 4 graph operations it is inefficient to parse every single graph operation. In such a scenario one could argue that it would be more efficient to only parse graph operations included in the tree since that would lead to no unnecessary parsing. On the other hand, a solution of only parsing necessary operations introduces the requirement of checking if an operation has been parsed or not. This leads to an unnecessary amount of checking if the tree file includes trees with overlapping graph operations. Since the idea of *Lovelace* is to generate corpora independent of the number of trees in the input file the file is most likely going to include trees with overlapping graph operations. Therefore the solution of parsing every operation at once is the most suitable solution for the corpus generator.

As described in Section 3, the edges of a node in *GEG* are instantly introduced when the node is created, this allows for *GEG* to decide its non-uniform membership problem[1] in polynomial time. It does however present the problem of not being able to create nodes with an arbitrary number of edges which limit the amount of relations a node can have with other nodes. Björklund et al. [2] solved this by adding a technique introduced by [15] known as cloning to *GEG*. This technique enables the cloning of nodes which leads to nodes in a graph generated by *GEG* being able to represent an arbitrary number of edges from a node. This aspect is not yet implemented in *Lovelace* which means that the extension operations are bound by the number of edges declared in each operation. This reduces the expressiveness of the graph operations in *Lovelace*, thus implementing cloning would be of interest to researchers.

Another limitation of *Lovelace* is that the only way for an extension operation to fuse with a contextual node is by searching for a node with a specific node name. This means that there is no way to base the contextual node on the semantic (other than the name of the node) in the underlying graph. Björklund et al. [2] introduce logical expression in the extension operation to enable the fusing with contextual nodes to be based on a logical expression representing a specific semantic condition in the underlying graph. Much like cloning the introduction of a logical expression for contextual nodes would increase the expressiveness of the graph operations in *Lovelace*. Therefore it is an interesting subject for future work.

5.2 Important functionality of a semantic graph corpus generator

To be able to create a corpus of semantic graphs the nodes need meaning. One way of making meaning is by using semantic word banks which are described in Section 4.4. Given that each variable is defined as a word class it gives meaning to the node represented by the variable and the outgoing edges from that specific node. These word classes enable the semantic graphs generated by *Lovelace* to be correct for each word in a given word class since all words in a word class have the same outgoing edge arguments (Section 4.4 for an example). It should be noted that in order to construct an accurate semantic graph for each word within a word class, it is imperative for the word class to be defined correctly. The usage of word classes leads to fewer word definitions, thus fewer graph operations. Being able to generate more semantic graphs with fewer graph operations (compared to if each word had its own graph operation) is an advantage of using word classes. However, this is not the only advantage of using word classes. The combination of well-defined word classes and semantic graphs means that the semantic graphs generated by *Lovelace* are unambiguous since if a word has a different semantic meaning the word is represented in more multiple word classes, therefore, the semantic graphs are only representing one specific semantic behaviour. Escaping the ambiguous nature of natural language is an important aspect when translating and processing natural language.

Given that a corpus generator is expected to generate semantic graphs for an extensive set of trees, the set may include trees that are not of relevance or interest. Hence, a crucial function of a corpus generator lies in its capability to selectively exclude undesirable trees from the set. An undesired tree refers to a tree that lacks a particular graph operation or employs an insufficient or excessive number of graph operations *Lovelace* handle the filtering of trees that lack a specific graph operation, has an insufficient or excessive number of graph operations through the use of the optional parameters explained in Section 4.6 through the use of the flags L, H, and k.

The only filtering capability that *Lovelace* lacks is the ability to filter based on specific words. This form of filtering assumes significance when it is necessary for each semantic

[1] given that the graph is in the graph language

graph in a corpus to incorporate a specific keyword. Some ideas for graph filtering are the following:

1. Graph has to include a specific word.
2. Just allow words from a definition file to appear once in each graph.
3. Require each word from a word set to appear at least once in each graph.
4. Only allowing one word from a word set to define every variable representing that specific word set.

All of these examples of filtering could be made into flags, using the same principle as in tree filtering. The filtering of graphs would allow for further specification of which corpus a user wants to generate which means that the relevance of the generated corpus increases.

The impact that the development of *Lovelace* has on the field of Computing Science is that it gives information about which design and implementation aspects are important when implementing a corpus generator. From a linguistic point of view *Lovelace* enables the generation of corpora that are of interest when translating or processing natural language given that the user is able to create a *GEG* suited for their linguistic interest.

6 Conclusion

The purpose of this thesis is to answer what functionalities are useful in a semantic graph corpus generator and which design considerations are vital for a successful implementation of a corpus generator based on *GEG*. To answer these questions the corpus generator *Lovelace* was implemented.

When it comes to the implementation and design aspects it was realized that dividing the *GEG* into two input files is the more intuitive option since it enables the user to understand that a *GEG* is a pairing of an *RTG* and graph operations. Furthermore, it was argued that the best way of parsing the graph operations is by parsing all of them at the same time since it means that there is no need to check whether an operation is parsed or not. It was also argued that most graph operations are used since the corpus generator is supposed to handle an unbound number of trees. The operations in *Lovelace* could be made more expressive if cloning of nodes and a logical expression to connect to contextual nodes were implemented. Therefore it is an interesting subject for future research as well as the combination of *BETTY* and *Lovelace*.

When evaluating the useful functionalities of a corpus generator it is noted that the combination of semantic graphs and well-defined word classes not only leads to fewer operations (since a variable can represent a word class) but also generates unambiguous corpora. This is because of the unambiguous nature of well-defined word classes. Another functionality that has a linguistic implication is the optional variables which enable the filtering of trees in *Lovelace*. This enables a user to more precisely specify the corpus that the user wants to generate. However, there is no filtering of semantic graphs. An implementation of such filtering would enable the user of *Lovelace* to further specify the corpus which leads to their corpus being even more relevant for their interest area. This suggests that corpora generated by *Lovelace* has the potential to serve as a valuable tool for natural language processing and translation tasks.

References

[1] I. Langkilde and K. Knight, “Generation that exploits corpus-based statistical knowledge,” in *COLING 1998 Volume 1: The 17th International Conference on Computational Linguistics*, 1998.

[2] J. Björklund, F. Drewes, and A. Jonsson, “Generation and polynomial parsing of graph languages with non-structural reentrancies,” *Computational Linguistics*, vol. 1, no. 1, 2023. [Online]. Available: https://arxiv.org/pdf/2105.02033.pdf

[3] 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.

[4] 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.

[5] A. Jonsson, “Generation of abstract meaning representations by hyperedge replacement grammars–a case study,” 2016.

[6] 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.

[7] Y. Stade, “Language theoretic properties of graph extension languages: An investigation of graph extension grammars with context matching and logic,” 2022.

[8] J. Björklund, B. Björklund, F. Drewes, and A. Jonsson, “Improved n-best extraction with an evaluation on language data,” *Computational Linguistics*, vol. 48, no. 1, 2022. [Online]. Available: https://doi.org/10.1162/COLI

[9] M. Palmer, “Semlink: Linking propbank, verbnet and framenet,” in *Proceedings of the generative lexicon conference*. GenLex-09, Pisa, Italy, 2009, pp. 9–15.

[10] K. Kipper, A. Korhonen, N. Ryant, and M. Palmer, “A large-scale classification of english verbs,” *Language Resources and Evaluation*, vol. 42, pp. 21–40, 2008.

[11] D. Gildea and M. Palmer, “The necessity of parsing for predicate argument recognition,” in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 2002, pp. 239–246.

[12] J. Ruppenhofer, M. Ellsworth, M. Schwarzer-Petruck, C. R. Johnson, and J. Scheffczyk, “Framenet ii: Extended theory and practice,” International Computer Science Institute, Tech. Rep., 2016.

[13] R. Weischedel, S. Pradhan, L. Ramshaw, M. Palmer, N. Xue, M. Marcus, A. Taylor, C. Greenberg, E. Hovy, R. Belvin *et al.*, “Ontonotes release 4.0,” *LDC2011T03, Philadelphia, Penn.: Linguistic Data Consortium*, 2011.

[14] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, “Graphviz and dynagraph—static and dynamic graph drawing tools,” *Graph drawing software*, pp. 127–148, 2004.

[15] F. Drewes, B. Hoffmann, D. Janssens, and M. Minas, “Adaptive star grammars and their languages,” *Theoretical Computer Science*, vol. 411, no. 34-36, pp. 3090–3109, 2010.

Appendices

The appendix includes one listing which is the graph operation file used in *Lovelace* depicted in Listing 6. The operations are the operation in Figure 5 on a format parsable by *Lovelace*.

```
OPERATION: sx {
    0[label="x", port=1]
}

OPERATION: s'x {
    0[label="x", port=1]
}

OPERATION: zs1 {
    0[label="z", port=1]
    1[dock=1]

    0 -> 1 [label="arg0"]
    0 -> 1 [label="arg1"]
}

OPERATION: zs2 {
    0[label="z", port=1]
    1[label="y"]
    2[dock=1]

    0 -> 1 [label="arg1"]
    0 -> 2 [label="arg0"]
}


OPERATION: zs3 {
    0[label="z", port=1]
    1[label="x"]
    2[dock=1]

    0 -> 1 [label="arg0"]
    0 -> 2 [label="arg1"]
}

OPERATION: zs4 {
    0[label="z", port=1]
    1[dock=1]
    2[dock=2]

    0 -> 1 [label="arg0"]
    0 -> 2 [label="arg1"]
}

OPERATION: trys {
    0[label="try", port=1]
    1[dock=1]
    2[dock=2]

    0 -> 1 [label="arg0"]
    0 -> 2 [label="arg1"]
}

OPERATION: persuades {
    0[label="persuade", port=1]
    1[label="x"]
    2[dock=1]
    3[dock=2]

    0 -> 2 [label="arg1"]
    0 -> 3 [label="arg2"]
    0 -> 1 [label="arg0"]
}

OPERATION: persuadess {
    0[label="persuade", port=1]
    1[dock=1]
    2[dock=2]
    3[dock=3]

    0 -> 1 [label="arg0"]
    0 -> 2 [label="arg1"]
    0 -> 3 [label="arg2"]
}


OPERATION: zc1 {
    0[label="z", port=2]
    1[label="undef", port=1]
    2[dock=1]

    0 -> 2 [label="arg0"]
    0 -> 2 [label="arg1"]
}

OPERATION: zc2 {
    0[label="z", port=2]
    1[label="undef", port=1]
    2[label="y"]
    3[dock=1]

    0 -> 3 [label="arg0"]
    0 -> 2 [label="arg1"]
}

OPERATION: zc3 {
    0[label="z", port=2]
    1[label="x", port=1]
    2[dock=1]

    0 -> 1 [label="arg0"]
    0 -> 2 [label="arg1"]

}

OPERATION: zc4 {
    0[label="z", port=2]
    1[label="undef", port=1]
    2[dock=1]
    3[dock=2]

    0 -> 2 [label="arg0"]
    0 -> 3 [label="arg1"]

}

OPERATION: tryc {
    0[label="try", port=2]
    1[label="undef", port=1]
    2[dock=1]
    3[dock=2]

    0 -> 2 [label="arg0"]
    0 -> 3 [label="arg1"]
}

OPERATION: persuadec {
    0[label="persuade", port=2]
    1[label="x", port=1]
    2[dock=1]
    3[dock=2]


    0 -> 1 [label="arg0"]
    0 -> 2 [label="arg1"]
    0 -> 3 [label="arg2"]
}

OPERATION: persuadecc {
    0[label="persuade", port=2]
    1[label="undef", port=1]
    2[dock=1]
    3[dock=2]
    4[dock=3]

    0 -> 2 [label="arg0"]
    0 -> 3 [label="arg1"]
    0 -> 4 [label="arg2"]
}


OPERATION: u {
    1 1
}

OPERATION: u' {
    1 2
}
```

Listing 1. A complete textual representation of the graph extension grammar operations depicted in Figure 5.

UMEÅ UNIVERSITY