Graph Algebraic Grammars for Semantic Parsing

Dissertation an der Universität Hamburg

Sebastian Beschke

2022

Fakultät für Mathematik, Informatik und Naturwissenschaften,
Fachbereich Informatik

Supplementary material is available under `https://gitlab.com/nats/gramr-thesis-experiments`.

Gutachter:

Prof. Dr.-Ing. Wolfgang Menzel (Universität Hamburg)
Prof. Dr. Alexander Koller (Universität des Saarlandes)
Hamburg, den 7. 11. 2022 (Tag der Disputation)

Acknowledgements

This dissertation project has spanned nine years and leaves me indebted to a lot of people, academically and otherwise, who all have in some way supported me during this time. I will try my best to thank as many of them as I can, even if there are too many to name individually.

I thank my supervisor, Wolfgang Menzel, for giving me the chance to pursue a project in computational linguistics back when I had little knowledge of the field, for giving me the room to grow into it, and for sticking with me and supporting me throughout the project even when I was making little progress.

I am glad to have shared this time with all the friendly people at the University of Hamburg, from the NATS, WSV and LT groups, and have all the inspiring exchanges ranging from paper writing advice to lunch banter. It was a pleasure!

The same goes for everyone I met at LangTec. I feel privileged to have been part of such a talented and inspiring crew. In particular, I thank Patrick McCrae for guidance and mentorship.

In 2013 I had the unique opportunity to work at Tsinghua University's Natural Language Processing lab under the supervision of Prof. Yang Liu and Prof. Maosong Sun. This was an intense and productive time, and next to the professors, I thank all lab members, who welcomed me with open arms.

Dissertation writing is not only demanding of the writer, and not just intellectually. Agnes, you bore the emotional load that came with this project and helped me cope. Without you, I do not believe that I would have seen it through. Thank you for being with me.

Submitting this thesis feels like a milestone on a journey I have been on for a long time. Since my childhood, my parents have gone out of their way to put all possibilities within my reach, and even if my goals must not always have made sense to them, they trusted me and made sure I could pursue them. For that, and everything else, I am deeply grateful.

Contents

Chapter 1.

Introduction

In 2021, we rely more than ever on computers to access an increasing wealth of information. Crucial scientific or business insights may be hidden inside vast databases or large quantities of unstructured text data. Making unstructured information available to automated processing (also known as *natural language understanding*) is therefore of increasing importance, as is the efficient access to large amounts of structured data.

Semantic parsing is a technology that can help address these challenges by transforming unstructured text into formal representations. Such meaning representations can then be used to perform downstream tasks, such as querying a database (effectively providing a *natural language database interface*) or extracting the facts expressed in the meaning representations and storing them in a knowledge base. In both cases, semantic parsers can be part of a toolchain that bridges the gap between human and computer-based processing of information.

In our approach, English sentences such as *The programmer wanted to find seven bugs.* are transformed into graphs that capture some aspects of its meaning, like the graph in Figure 1.1:

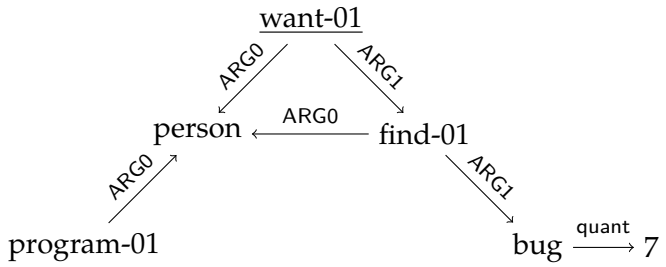


Figure 1.1.: A graph representing the meaning of the sentence *The programmer wanted to find seven bugs.*

- The main events of *wanting* and *finding* as well as the involved entities, a *person* and some *bugs*, are represented by nodes. The person is the agent (ARG0) of several events at once.
- Edge labels describe the roles taken by the entities: the programmer *finds* and *wants*, and the bugs *are found*.
- The number of *bugs* is expressed through a special role (quant) of the *bug* entity.
- The programmer is represented as a *person who programs*, showing how the meaning representation employs concepts to abstract away from concrete words.

This meaning representation is *abstract* in the sense that it does not refer to the sentence from which it was derived. The concepts referenced in the meaning representation are thought to be more like logical predicates and do not need to match any words present in the sentence.

Still, the information necessary to construct the meaning representation as shown above is encoded in the sentence. The relationships expressed in the graph result from those found in the syntactic structure of the sentence. The process that derives a meaning representation from a sentence can be encoded in the form of a grammar.

In this thesis, we extend the well-established *Combinatory Categorial Grammar* with operations for graph construction. Our semantic parsing system automatically creates a lexicon of building blocks for meaning representations while keeping the lexicon small and interpretable. As a result, our system works with limited computational resources, shows strong performance even with small amounts of training data, and its output is interpretable with regards to linguistic theory.

1.1. Research Questions

Implementors of semantic parsers face a number of problems that make semantic parsing a challenging task:

- **Large structured output spaces:** There is an infinite number of meaning representation graphs, and an exponential number of syntactic derivations for a sentence. A semantic parser must be able to find an appropriate output in this infinite space.

- **Working with latent structure:** No derivations for meaning representation graphs are provided in the training data, and thus, such structure must be inferred by a grammar-based semantic parser.

- **Limited training data availability:** The manual annotation of meaning representations is costly because meaning representations are complex and annotators require training and experience with meaning representation formalisms. Data sets therefore tend to be considerably smaller than those available for tasks where data collection is easier.

Our strategy for addressing these challenges is to aim for a system whose core mechanisms are *simple* and *transparent* and which is strongly biased towards *linguistic plausibility*.

Simple mechanics help address the computational issues raised by exponential search and output spaces. We therefore design a grammar based on a few simple graph construction operations in order to reduce ambiguity. This design choice also leads to transparency by allowing derivations to be represented in a readable, uncluttered manner.

A linguistic footing allows the system to make the most of the available training data. We therefore base our grammar on Combinatory Categorial Grammar, which allows us to use existing tools and biases our system towards plausible interpretations.

This strategy leads to the following research questions:

1. How can Combinatory Categorial Grammar be adapted to use graph-based semantic construction operators while achieving good coverage of a natural language corpus?
   a) Which is the smallest set of fundamental grammar rules to achieve good coverage and strong end-to-end parsing performance?
   b) What are the limitations of this approach in relation to the Abstract Meaning Representation corpus and the linguistic properties of Combinatory Categorial Grammar?

2. How can a semantic lexicon be induced that has broad coverage, but is small enough to be inspected manually?

3. How can a semantic parser be built that makes use of an induced semantic lexicon and overcomes the computational challenges of semantic parsing?

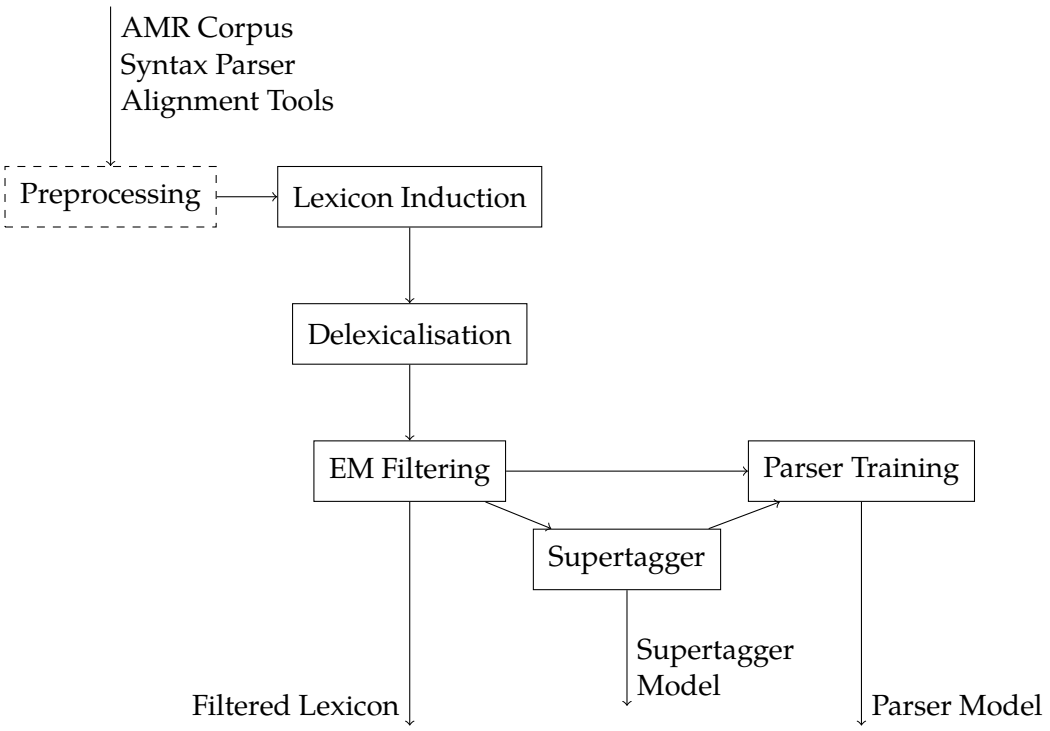


Figure 1.2.: An overview of the software pipeline developed in this thesis. All solid boxes are new software components developed for this research. Arrows indicate the data flow between components. Arrows to the bottom show the system's outputs.

a) How does this parser's performance compare to that of other approaches?

b) What is the parser's error profile?

c) How does the parser perform when the amount of training data is artificially limited?

1.2. Implementation

The present work is empirical in nature. Throughout the thesis, our focus is on broadening the scope of the practically feasible. All concepts discussed in the following chapters have therefore been implemented as part of a new semantic

parsing system which is based on the principles of Graph Algebraic Combinatory Categorial Grammar and allows us to examine practical issues of their application.

Our implementation consists of a pipeline of several components. Through the pipeline, information extracted from an annotated corpus is propagated to produce a functioning semantic parser which is capable of producing meaning representations for new sentences. The steps of our parser's training pipeline are visualised in Figure 1.2. Below, we give a summary of each step along with a brief description of its design considerations.

First, in the preprocessing step, the input corpus, which is annotated with Abstract Meaning Representations (AMRs), is enriched using several external tools. This includes standard natural language processing techniques such as tokenisation and lemmatisation, as well as more specific tools for syntactic Combinatory Categorial Grammar (CCG) parsing and AMR alignment.

The preprocessed training corpus is fed into a lexicon induction algorithm, which creates a large number of lexical items that can be used to build semantic analyses for new sentences. However, these lexical items are tied to concrete word forms, and thus not applicable to words not seen in the training corpus. The delexicalisation step therefore generates generic templates from the concrete lexical items, each of which describes the semantic structure associated with a word without its lexical information.

Both the delexicalisation step and the lexical induction algorithm itself overgenerate, which makes a filtering step necessary. Using a probabilistic model of syntactic-semantic derivations, an expectation maximisation (EM) algorithm is used to assign probabilities to the delexicalised lexicon entries and reduce it to the set of items with the highest explanatory power.

In principle, the resulting filtered, delexicalised lexicon is well-suited for parsing new sentences. However, in practice a parser struggles to select from the large number of templates that are still in the lexicon. We therefore train a neural network-based supertagger to limit the number of templates the parser has to consider.

Finally, a structured learning algorithm is used to train a machine learning model which can drive a bottom-up parsing algorithm towards plausible analyses. After training, the resulting model, along with the filtered, delexicalised lexicon and the supertagging model, can be used to parse new text.

1.3. Scope and Contributions

In this thesis we describe a new approach to the semantic parsing of graph meaning representations. The approach is discussed both on a theoretical level and in the context of a concrete implementation of a semantic parsing system.

Due to the complexity of the implementation, a large part of this thesis is dedicated to describing and validating the individual algorithms that are part of the semantic parsing pipeline. We develop a viable proof-of-concept which covers common linguistic phenomena at a reasonable level of performance. Throughout the thesis, we therefore forego some opportunities for algorithmic and linguistic improvements in favour of presenting a "minimal working configuration".

The remainder of this thesis is structured as follows:

- **Chapter 2: Abstract Meaning Representation Parsing** – We introduce the task of semantic parsing, as well as the Abstract Meaning Representation language. We also examine the state of the art of Abstract Meaning Representation parsing and other important related work.
- **Chapter 3: Foundations** – In this chapter, we explain the most important concepts upon which this work is based. This includes CCG, graph algebras for semantic construction, and several machine learning algorithms required for building the semantic parsing pipeline.
- **Chapter 4: Graph Algebraic Combinatory Categorial Grammar** – In the first part of this chapter, a graph algebra is designed whose operations are tailored towards the construction of Abstract Meaning Representations. Then, a modification of Combinatory Categorial Grammar is defined which makes use of this algebra. Finally, some limitations of the grammar are discussed.

  Contributions:

  - A definition of Graph Algebraic Combinatory Categorial Grammar (GA-CCG), a novel grammar for semantic parsing
  - A theoretical discussion of GA-CCG's limitations
- **Chapter 5: Induction of Graph Algebraic CCG Lexica** – In this chapter, we discuss the automatic induction of GA-CCG lexical items from an AMR corpus. We introduce an algorithm which recursively splits the annotated meaning representations to end up with reusable graph fragments. Heuristics that are necessary to control the computational complexity of the algorithm are

described. We evaluate the coverage of the algorithm using several instances of GA-CCG grammars.

Contributions:

- An algorithm for GA-CCG lexical induction
- An evaluation of the coverage of GA-CCGs on an AMR corpus

- **Chapter 6: Post-Processing the Lexicon: Delexicalisation, Filtering, and Supertagging** – This chapter is dedicated to pipeline steps necessary to make the induced GA-CCG lexicon usable for a semantic parser. This includes the steps of delexicalisation (to improve coverage), filtering (to reduce ambiguity), and supertagging (to reduce the amount of computation required by the parser).

  Contributions:

  - An expectation maximisation algorithm over CCG derivations which estimates the probabilities of lexical entries and which can be used to filter a highly ambiguous GA-CCG lexicon
  - A neural network-based supertagger which predicts lexical templates from an induced GA-CCG lexicon and is able to deal with gaps in its training data

- **Chapter 7: Parsing with Graph Algebraic Combinatory Categorial Grammars** – We describe the algorithms necessary for parsing with an induced GA-CCG. This includes a cost-sensitive perceptron algorithm for training a linear model to score parsing hypotheses, as well as a bottom-up parsing algorithm to decode from the model. To bootstrap the model, we also define an oracle heuristic.

  Contributions:

  - An adaptation of a cost-sensitive perceptron algorithm to semantic parsing
  - A training schedule for the semantic parser including oracle-based bootstrapping

- **Chapter 8: Evaluation of Graph Algebraic Combinatory Categorial Grammars for Semantic Parsing** – In this chapter, we describe end-to-end experiments encompassing the entire semantic parsing pipeline. This includes an

analysis of the influence of various hyperparameters, features, and grammar variants, as well as an error analysis of parser output.

Contributions:

- A comparison of results to other grammar-based semantic parsers
- A qualitative analysis of the semantic parser's output and error profile

- **Chapter 9: Conclusion** – We draw conclusions from experimental results and point out further research directions.

Chapter 2.

Abstract Meaning Representation Parsing

A *semantic parser* is a program that translates natural language into formal meaning representations. Within the scope of this broad definition, there exist a wide range of approaches, including both rule-based and learned systems. The field of meaning representation languages is equally diverse ranging from narrowly-defined, task specific to generic, broad coverage representations.

The ideas and tools used in today's semantic parsing systems can be traced back to their origins in linguistics and human-computer interfaces. Following Evang (2016), we distinguish narrow-coverage and broad-coverage semantic parsing systems and give a brief history of both.

One early impulse for research on narrow-coverage semantic parsing has been the desire for natural language human-computer interfaces, where a system communicates with the user in natural language in order to perform some task. An example is the famous SHRDLU system (Winograd 1971), where the user instructs a computer to perform actions in a virtual block world. While SHRDLU used a grammar that was specified manually in a laborious effort, later efforts have moved towards automatically inducing grammars from examples. This has been enabled by the creation of corpora such as ATIS (Price 1990) and Geoquery (Zelle and Mooney 1996), which contain natural language queries to information systems on flight connections and geography, respectively. For instance, Wong and Mooney (2007) induce a synchronous context-free grammar that builds meaning representations in parallel to syntactic derivations using λ-calculus.

Broad-coverage semantic parsing has emerged out of efforts in linguistics to model the mechanisms by which meaning is created in language. In this context, meaning is often represented in terms of model-theoretic semantics, which identify the semantics of an utterance with its propositional content. The pioneering work of Montague (1973) has been very influential and inspired further compositional and logic-based approaches to semantic derivation. One of them is Discourse Representation Theory (DRT; Kamp and Reyle 1993), which uses meaning representations

that are an extension of first-order logic. Another is Combinatory Categorial Grammar (CCG; Steedman 2000), which describes syntax as a process that links natural language utterances to the semantic structures via a combinatory interface.

Both theoretical frameworks have been combined in the implementation of Boxer (Bos 2008), which constructs discource representation structures of DRT by applying a set of hand-written rules to syntactic CCG derivations. This tool has in turn be used to construct a large-scale broad-coverage corpus annotated with DRT, the Groningen Meaning Bank (Basile et al. 2012). Based on this corpus, broad-coverage parsers have been trained which do not rely on hand-written rules (Le and Zuidema 2012, among others). As an alternative to the hierarchical, detailed representations of DRT, the AMR language has been created to provide an easier target for both parsers and annotators (Banarescu et al. 2013). As it is the focus of this thesis, we discuss AMR parsing in more depth in the following section.

While both approaches to semantic parsing may differ in their motivations, nowadays there is significant technological overlap between both worlds. For instance, CCG-based approaches first matured in narrow-coverage settings (Kwiatkowski et al. 2010; L. S. Zettlemoyer and Collins 2005, among others) before being successfully applied to AMR parsing (Artzi, K. Lee and L. S. Zettlemoyer 2015).

The remainder of this chapter focuses on AMR parsing. In Section 2.1, the AMR language and its features are introduced. Section 2.2 gives an overview of the most highly-performing AMR parsers and the variety of approaches that appear in the recent literature.

2.1. Abstract Meaning Representation

Abstract Meaning Representation (AMR) is a formal language that expresses the semantics of natural language sentences in the form of rooted, directed graphs (Banarescu et al. 2019). The nodes in an AMR represent entities, events, or concepts, and edges define role relationships between the nodes. Edges are labelled with the role they represent. Nodes are only labelled if they are leaves, in which case they are assigned a concept. Non-leaf, or variable, nodes (that is, entities and events) are unlabelled, but always instantiate some concept, which is expressed with a special instance role. Exactly one node of an AMR is identified as the *root* of the meaning representation and specifies the semantic focus of the utterance.

2.1.1. Definition

To capture the formal aspects of this definition and make it available to build upon in later chapters, we first define semantic graphs. We will extend this definition in Chapter 4 to define GA-CCG s*-graphs.

Definition 2.1 (Semantic Graph)**.** Let

- $V = V_c \cup V_v$ a set of vertices with $V_c \cap V_v = \{\}$,
- $v_{\text{root}} \in V$,
- $E \subseteq V \times V$ a set of directed edges,
- L a set of labels,
- $l : V_c \cup E \to L$ a labeling function.

Then $G = (V, E, v_{root}, l)$ is a directed, labelled, and rooted graph. We call G a *semantic graph* with root v_{root}, constant nodes V_c and variable nodes V_v.

In the definition, the set of nodes V is divided into a set of variable nodes, V_v, and a set of constant nodes, V_c. Only constant nodes are assigned a label. In an AMR, every variable node has an outgoing edge labelled instance which connects it to a constant node and marks the variable as an instantiation of the abstract concept represented by the constant node. Constant nodes are always leaves. These aspects of an AMR are captured in the following definition:

Definition 2.2 (Abstract Meaning Representation)**.** Let $G = (V, E, v_{\text{root}}, l)$ a semantic graph with constant nodes V_c and variable nodes V_v.
G is called an *Abstract Meaning Representation* (*AMR*) iff all of the following hold:

- $l(v) \in L_{\text{concepts}}$ for $v \in V_c$ (all constant nodes are labelled with concepts),
- $l(e) \in L_{\text{relations}}$ for $e \in E$ (all edges are labelled with relations),
- For all variable nodes $v \in V_v$, there is exactly one edge $e = (v, v') \in E$ with $v' \in V_c$ and $l(e) =$ `instance` (all variable nodes have an outgoing edge labelled `instance` which targets a constant node),
- No constant node $v \in V_c$ has an outgoing edge.

Node and edge labels are drawn from separate sets, which we call L_{concepts} and $L_{\text{relations}}$, respectively. This distinction is sufficient for our formal definition, but in AMR, these sets can be further subdivided.

Concepts can be OntoNotes frames (Hovy et al. 2006), lemmas, or constants. An OntoNotes frame is used whereever the event in question can be represented as one. In OntoNotes, frames are represented by a verb lemma and a sense tag, such as `want-01`. Frames can therefore be looked up in the OntoNotes corpus to obtain a definition. In cases where no frame is applicable, the concept label is a simple lemma such as `boy`. Additionally, there is a number of concepts used in specific cases, such as quoted literals (`"Korea"`), numerals (`20`), and `-` to indicate negation.

Relations can be subdivided into core and non-core roles. Core roles take the form ARGn. The available core roles and their semantics are defined by OntoNotes; this implies that core roles are only available for instances of OntoNotes frames. There is also an inventory of universal non-core roles such as mod (modification), polarity (negation), time, location, etc.[1] Additional non-core roles can be formed from prepositions, such as prep-along-with, prep-toward, etc. Finally, relations of the form op$n$ are used to represent ordered lists, such as lists of conjuncts, or the tokens making up a named entity.

2.1.2. Scope

In comparison to other broad-coverage meaning representations, such as the Discourse Representation Structures used in the Groningen Meaning Bank (Basile et al. 2012), AMR is a simplified meaning representation which excludes some aspects of semantics. It has been designed this way to improve human readability and ease automated processing, and to allow large-scale annotation from scratch (Banarescu et al. 2013).

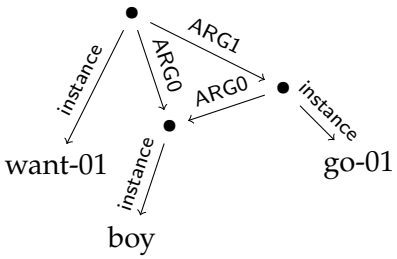
Possibly the most noticeable omission in AMR when compared to more conventional logic-based representation are scoped quantifiers. Variables in AMR are implicitly quantified existentially, and a logic-based interpretation of AMR can accomodate at most one universal quantifier per sentence (Bos 2016). In addition, grammatical features such as tense, number and aspect are not represented in a systematic way. AMR annotation is performed on the sentence level with no information on coreferences between sentences. The authors of the AMR specification also point out that the design and vocabulary are biased towards English (Banarescu et al. 2019), although AMRs for other languages have also been created (Li et al.

[1] The full list of AMR roles can be found in the AMR specification, Banarescu et al. 2019.

2016).

2.1.3. Visual Representation

According to the definition, all internal (variable) nodes of AMRs are unlabelled, but always instantiate a concept. For instance, the AMR paper (Banarescu et al. 2013) contains the following AMR graph, representing the sentence *The boy wants to go*:



While this visualisation explicitly represents all edges and nodes in the AMR, it is somewhat cluttered. In this thesis, we therefore employ a simplified notation, where the concept instantiated by a variable node is placed into the variable node itself. The root node of a graph is underlined, and in cases where constant nodes are represented explicitly (whenever they are not connected with an instance edge), the node label is written in `monospace` font. In our simplified visualisation, the AMR shown above is represented as follows:

This simplified visualisation allows for a compact and intuitive representation of AMRs while still emphasizing their graph nature.

2.1.4. AMR Corpora

Three broad-coverage English corpora with annotated AMRs have been made available so far. The AMR Annotation Release 1.0 (Knight, Laura Baranescu et al. 2014)

consists of newswire and discussion forum texts with sentence-wise, manually created AMR annotations. In total, the corpus contains 13,051 meaning representations from various sources including newswire text and discussion forums. Its `PROXY` section, containing 8,252 sentences of newswire text, is frequently used on its own for the evaluation of AMR parsers. The `consensus` section contains 200 sentences taken from the Wall Street Journal corpus which is part of the Penn Treebank and other syntactic corpora such as CCGBank, which allows the various representations to be compared. Sentences in the AMR 1.0 corpus are separated into a training, dev, and test set according to a rough 80/10/10 ratio, and the final evaluation of parsers is performed on the test set.

Two further releases of broad-coverage AMR corpora followed: The AMR Annotation Release 2.0 (Knight, Bianca Badarau et al. 2017) with a total of 39,260 AMRs, additionally including a large amount of discussion forum texts, and the AMR Annotation Release 3.0 (Knight, Bianca Badarau et al. 2020), with additional discussion forum data as well as sentences from other sources, totalling 59,255 AMRs.

A number of other corpora annotated with AMRs also exist, among them an annotated version of the novel *The Little Prince* by Antoine de Saint-Exupéry[2], as well as a corpus of cancer-related biomedical research papers[3]. The *Little Prince* corpus has additionally been annotated with AMRs in Chinese (Li et al. 2016) and Brazilian Portuguese (Sobrevilla Cabezudo and Pardo 2019). Also, Damonte and Cohen (2018) have created a multilingual corpus of AMRs across parallel corpora in Italian, Spanish, German, and Chinese.

2.1.5. Evaluation of AMR Parsers

Evaluating the output of an AMR parser is not trivial given the abstract nature of AMRs. While it is desirable to compare the parser's output graphs element-by-element to the gold-standard graph to produce a granular evaluation, it is unclear which element of one graph corresponds to which element of the other, since the AMRs' nodes are not anchored in the sentence or some other common structure.

The established metric for AMR evaluation, Smatch (Cai and Knight 2013), tackles this problem by including an optimization procedure in the calculation of the metric. Both AMRs are decomposed into triples of either of two forms:

[2]Provided at `https://amr.isi.edu/download/amr-bank-struct-v3.0.txt` without authorship information. Retrieved 11 November 2021.

[3]Provided at `https://amr.isi.edu/download/2018-01-25/amr-release-bio-v3.0.txt` without authorship information. Retrieved 11 November 2021.

- *relation*(*variable*, *concept*)
- *relation*(*variable1*, *variable2*)

An additional "triple" *root*(*variable*) is included to allow the focus assigned by the parser to be judged by the evaluation metric.

The Smatch algorithm then computes precision (P), recall (R), and F1 (F) score on these triples extracted from AMRs G_{parsed} and G_{gold}.

$$P = \frac{|\text{triples}(G_{\text{parsed}}) \cap \text{triples}(G_{\text{gold}})|}{|\text{triples}(G_{\text{parsed}})|} \tag{2.1}$$

$$R = \frac{|\text{triples}(G_{\text{parsed}}) \cap \text{triples}(G_{\text{gold}})|}{|\text{triples}(G_{\text{gold}})|} \tag{2.2}$$

$$F = 2\frac{PR}{P+R} \tag{2.3}$$

Since the variables of both AMRs are abstract symbols, they need to be mapped onto one another so that overlapping triples can be properly identified. The Smatch F1 score is the highest score achieved by any variable mapping for a given AMR pair. To obtain this mapping, a software package is provided which runs a hillclimbing optimisation algorithm [4].

Although this optimisation algorithm is a local search and therefore not guaranteed to find the optimal mapping, this issue appears negligible in practice as multiple restarts can be used to obtain converging scores. Commonly, AMR parsers are evaluated by running the Smatch algorithm four times, where one iteration starts from a mapping initialised by a heuristic as described by Cai and Knight (ibid.) and the remaining iterations start from randomly assigned mappings. We follow this practice in our experiments.

2.2. Approaches to AMR Parsing

Approaches to semantic parsing of AMRs are so varied that it is difficult to do the whole field justice in a single section. Instead, this section will highlight some of the milestones of AMR parsing, following a basic categorisation scheme of approaches.

Although AMR parsers are complex systems with many components and features, parsing algorithms can be classified into broad categories. The abstract of

[4] Accessible at `https://github.com/snowblink14/smatch`, retrieved 4 July 2021.

Koller, Oepen and Sun (2019) distinguishes composition-based, factorisation-based, transition-based, and translation-based methods. We follow this categorisation in the following paragraphs

The first published AMR parser, JAMR, falls into the category of factorisation-based methods (Flanigan et al. 2014). Using a heuristic alignment algorithm, it decomposes AMR graphs into concepts. During parsing, it then predicts the concepts invoked by a sentence and connects them using a maximum spanning tree algorithm. Its alignment algorithm is also used by several other systems, including this thesis. More recently in the factorisation category, Zhang et al. (2019) approached AMR parsing using a novel neural architecture, the attention-based neural transducer.

Translation-based methods, which treat AMRs in a serialized format as the target language for a neural machine translation system, have also had some successes. For example, Noord and Bos (2017) showed that using appropriate pre- and post-processing and synthetic training data, a character-based translation algorithm can accurately produce AMRs. More recently, Xu et al. (2020) showed the effectiveness of pre-training a translation-based AMR parser on other sequence-to-sequence tasks, and achieved state-of-the-art performance.

Transition-based methods, starting with Wang, Xue and Pradhan (2015a), are another highly productive class of AMR parsing approaches. These systems tackle AMR parsing by predicting a sequence of actions, such as pulling entries from a lexicon, or combining two previously created parsing states. Notably, AMR parsing has served as an early area of application for pioneering deep learning architectures, such as Stack LSTM (Ballesteros and Al-Onaizan 2017) or Stack Transformer (Y.-S. Lee et al. 2020).

Composition-based approaches are based on grammars which describe how AMR fragments are put together to describe a sentence's meaning. As such, they are the most closely related to the linguistic field of computational semantics. This is most clearly reflected in CCG-based systems (Artzi, K. Lee and L. S. Zettlemoyer 2015; Misra and Artzi 2016) which encode AMR construction operations in λ-calculus and use CCG parsing algorithms to construct syntactic-semantic derivations. Another line of work uses interpreted regular tree grammars to describe the translation from sentences to AMR graphs (Groschwitz, Fowlie et al. 2017; Groschwitz, Lindemann et al. 2018; Lindemann, Groschwitz and Koller 2019). In this approach, a supertagger selects graph fragments from a lexicon for each input token. Then, a dependency parsing algorithm is used to predict graph composition operations which can be carried out on these fragments to produce the final meaning representation.

At present, there is a gap in performance between the strongest-performing

composition-based system (Lindemann, Groschwitz and Koller (2019) with 75.3 F1 on the AMR 3.0 corpus) and the overall strongest system (Xu et al. (2020) with 81.4 F1). The added linguistic depth of composition-based analyses therefore does not currently translate to an advantage in end-to-end performance.

2.2.1. Positioning of this Thesis

The semantic parsing system developed in this thesis follows a composition-based approach as it is based around an adaptation of CCG. It combines important aspects of both lines of work introduced above: Like Artzi, K. Lee and L. S. Zettlemoyer (2015), we use CCG to build syntactic-semantic analyses for AMR, and a CKY-style bottom-up parsing algorithm to construct them. And like Groschwitz, Lindemann et al. (2018), we define AMR construction operations in graph algebraic terms, and employ a neural supertagger to predict graph fragments.

While we draw heavily on these ideas, many aspects of our implementation have been adapted significantly, and some features have been added. For example, the graph-algebraic operators used in this thesis are tailored to model the semantic behaviour of CCG operators, leading to a larger number of rules compared to the work of Groschwitz, Lindemann et al. (ibid.). Another innovation is the dedicated lexicon filtering step, which leads to reduced computational requirements compared to the system of Artzi, K. Lee and L. S. Zettlemoyer (2015).

Apart from keeping the required amount of computation manageable, in this thesis we emphasize transparency as a fundamental aspect of grammar-based processing through the use of a compact lexicon and a simple graph-based construction mechanism.

Chapter 3.

Foundations

Like every scientific work, this thesis builds upon numerous concepts introduced by previous work. Some are important theoretical models which we use and extend. Others are algorithms which are needed to implement those ideas in practice. In this chapter, we introduce the concepts and algorithms upon which the rest of this thesis relies.

In Section 3.1, we discuss *Combinatory Categorial Grammar* (CCG), a grammar formalism which treats both syntax and semantics. CCG is the basis for the GA-CCG grammar developed in Chapter 4.

Section 3.2 introduces some mathematical background and notation on *graph algebras*. We use this notation in Chapter 4 to define the semantic construction mechanism for GA-CCG. The section also includes some straightforward extensions developed for this thesis.

In Section 3.3, basic algorithms for training linear models with structured output spaces are discussed. The prediction of structures, such as AMR graphs, is associated with certain challenges, which can be addressed with specific algorithms. Such algorithms are needed to train the GA-CCG parser developed in Chapter 7.

The induction process for GA-CCG, described in Chapter 5, produces a large amount of lexical items. Expectation maximisation (EM) is an unsupervised method for assigning probabilities to the lexical items, allowing useless items to be filtered out. In Section 3.4, we introduce EM, as well as its application to context-free grammars, the inside-outside algorithm. The latter algorithm is adapted to GA-CCG in Chapter 6.

Due to the large number of items in induced CCG lexicons, it is common to employ a supertagging step to predict the most likely items in a given context and reduce the burden of the parsing algorithm. Such supertaggers are often implemented using recurrent neural network models, which we introduce in Section 3.5. The supertagger developed for this thesis is described in Chapter 6.

3.1. Combinatory Categorial Grammar

Formal grammars are rule systems that describe how sentences of a language are formed. Through the recursive application of their rules, derivation structures are created which reflect the syntactic structure of a given sentence, providing an analysis of the (syntactic) relationships among its words and phrases.

Combinatory Categorial Grammar (CCG) is a type of formal grammar that models both natural language syntax and semantics. The nodes in its derivation carry information on both the syntactic type and the semantic content of the word or phrase represented by the node. The rules of CCG allow for the content of neighbouring nodes to be combined if their syntactic types allow it.

CCG embodies a computational and strictly compositional view of natural language semantics. Additionally, it is well-studied both from a linguistic and a computational perspective. It is general enough to use it for annotation of a broad-coverage treebank (Hockenmaier and Steedman 2007), and it can be parsed using reasonably efficient and well-understood algorithms (see Section 3.1.6 below). Its properties make CCG an attractive foundation for implementing a grammar-based semantic parsing system.

In this section, we give a minimal, but workable definition of CCG. A similar presentation of CCG is given in more detail in Chapter 3 of Steedman (2000). Some of the examples in this section were also taken from that chapter.

3.1.1. Fundamental Concepts

A CCG is made up of two inventories:

1. A *lexicon* describing the words included in the language and their syntactic and semantic properties.

2. A set of *combinatory rules* which define how derivations can be built from a list of lexical entries, and how the semantic interpretation of a derivation is constructed.

CCG is a *lexicalised* grammar, since the lexicon entries contain most of the information about the properties of individual words. In contrast, there are only a few generic combinatory rules, which act upon the categories drawn from the lexicon. The choice of lexical entries for every word thus determines a large amount of the structure of a sentence's derivation and its semantic representation.

Each entry of a CCG lexicon is a triple consisting of a word, a syntactic category, and a semantic category. The entry assigns syntactic and semantic properties to the word, but several entries may exist for any given word.

The *syntactic category* describes how the word interacts with other words on the syntactic level. It specifies which words can be combined into constituents, and therefore defines the space of the possible syntactic structures of the sentence. The *semantic category* specifies the semantic content of the word. In CCG, every syntactic operation is mirrored on the semantic level, so that the same computation builds up the syntactic structure of the sentence, and its meaning representation.

When a sentence is parsed with CCG, a lexical entry must be selected for every word in the sentence. Combinatory rules can then be applied to a pair of neighbouring words and their categories wherever there is a matching rule. As its output, a combinatory rule produces another pair of syntactic and semantic categories: this is a *derivation node*, which has the original words as its children.

Derivation nodes can in turn be combined with neighbouring words or nodes by combinatory rules, so that they form a tree structure. Once a derivation node has been constructed that encompasses the entire sentence, the entire structure is a *derivation* of the sentence, and its semantic category is a representation of the sentence's meaning.

In the following sections, we briefly define each of these concepts to give a fundamental understanding of the mechanics of CCG.

3.1.2. Syntactic Categories

Syntactic categories describe the syntactic type of a word or constituent. They are constructed as hierarchical structures of atoms and slashes.

Atomic categories do not take any arguments. Examples for atomic syntactic categories are noun phrases (NP), prepositional phrases (PP), and sentences (S).

Function categories are made up of a result category, a slash, and an argument category. Slashes exist as forward (/) and backward (\) slashes. The direction of the slash indicates the position of the expected argument: For example, the syntactic category S\NP indicates a result type of S and an argument type of NP, where the argument is expected to the left.

Example 3.1. In the sentence *I slept*, the syntactic categories can be assigned as follows:

$$\text{I} := \text{NP}$$
$$\text{slept} := \text{S}\backslash\text{NP}$$

I can be the argument for *slept* since it matches the argument type NP and is located to the left of *slept*, as required by the backward slash. The resulting syntactic category is S.

Slash categories can themselves be result and argument types, allowing complex categories such as (S\NP)/NP to be constructed. This is the category of transitive verbs, which accept a noun phrase to the right (the object) and one to the left (the subject). The resulting category is S, since the sequence of subject, verb, and object forms a full sentence.

3.1.3. Semantic Categories

A *semantic category* represents the semantic content of a lexical entry or derivation node. Like syntactic categories, they can be functions which receive the semantic categories of neighbouring derivation nodes as their arguments.

In the CCG literature, semantic categories are usually given as logical expressions embedded in a typed λ-calculus. In particular, they are required to have a semantic type that is compatible with the lexical entry's syntactic category, in the sense that any syntactically permissible rule can also be applied to the semantic category.

Example 3.2. The sentence *I slept* from Example 3.1 can be extended with semantic categories, as follows:

$$\text{I} := \text{NP} : \text{i}'$$
$$\text{slept} := \text{S}\backslash\text{NP} : \lambda x.\text{sleep}'\ x$$

Like the syntactic category S\NP is applied to the argument NP, so the semantic category $\lambda x.\text{sleep}'\ x$ is a function taking i' as an argument. Through β-reduction, the meaning representation for the whole sentence can be computed:

$$\begin{aligned} & (\lambda x.\text{sleep}'\ x)\text{i}' \\ \Rightarrow_\beta \quad & \text{sleep}'\ \text{i}' \end{aligned}$$

3.1.4. Combinators

In CCG, combinators define how categories interact with one another. Combinators are functions transforming one or several input categories into a result category. They are defined both over syntactic and over semantic categories, and are always applied on both levels in parallel. For example, the *forward application* combinator $>$ has the following definition:

$$X/Y : f \quad Y : g \quad \Rightarrow_{>} \quad X : fg \tag{3.1}$$

Here, the combinator $>$ acts in parallel on two pairs of inputs: The syntactic categories X/Y and Y, and the semantic categories f and g. In the definition of the combinatory rule, these categories are all patterns. When a sentence is processed, these patterns are matched onto the actual categories of neighbouring derivation nodes to determine the outcome of the rule.

Example 3.3. For example, this rule could be used to analyse the phrase *the dog*, given the following lexicon:

$$\text{the} := \text{NP/N} : \lambda x.x$$
$$\text{dog} := \text{N} : \text{dog}'$$

Given these lexical entries, the rule can be applied as follows:

$$\text{NP/N} : \lambda x.x \quad \text{N} : \text{dog}' \quad \Rightarrow_{>} \quad \text{NP} : \text{dog}'$$

The semantic category of *the* is the identity function: it simply returns its argument, adding no semantic content of its own. However, the syntactic category of *the* specifies that it transforms a noun (N) into a noun phrase (NP). Using the forward application combinator, we can thus derive the following *phrasal item*:

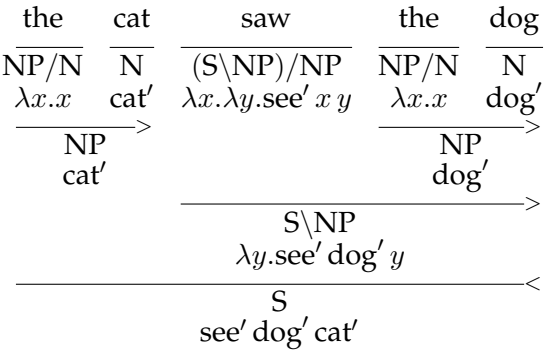
$$\text{the dog} := \text{NP} : \text{dog}'$$

CCG combinators observe the word order of the sentence. For the forward application combinator, the word occurring first in the sentence is always its first argument, and therefore always plays the role of the function. CCG also uses a

Lexicon

cat := N : cat
dog := N : dog
saw := (S\NP)/NP : $\lambda x.\lambda y.\,\text{see}\,x\,y$
the := NP/N : $\lambda x.\,x$

(a) A set of lexical entries.



(b) A CCG derivation.

Figure 3.1.: A CCG-based analysis of the sentence *The cat saw the dog*.

backward application combinator, but some other combinators are only used in one direction. Table 3.1 shows the definitions for the CCG combinators that are used in this thesis, which are taken from the grammar of the EasyCCG parser (Lewis and Steedman 2014) and slightly extended. In Chapter 4, we give examples for the linguistic constructions associated with each combinator.

3.1.5. Derivations

Given a set of combinators and a lexicon, we can analyse the syntactic structure and semantic content of sentences. As part of this analysis, each word of a sentence is assigned a lexical entry, and then combinators are used to recursively combine the lexical entries' categories, until a syntactic and semantic category for the entire sentence have been derived. The resulting structure is called a *derivation*.

Symbol	Name	Definition
$>$	Forward Application	$X/Y : f \quad Y : g \quad \Rightarrow \quad X : fg$
$<$	Backward Application	$Y : g \quad X\backslash Y : f \quad \Rightarrow \quad X : fg$
$>\mathbf{B}$	Forward Composition	$X/Y : f \quad Y/Z : g \quad \Rightarrow \quad X/Z : \lambda z.f(gz)$
$<\mathbf{B}_\times$	Backward Crossed Composition	$Y/Z : g \quad X\backslash Y : f \quad \Rightarrow \quad X/Z : \lambda z.f(gz)$
$>\mathbf{B}^2$	Generalised Forward Composition (2nd degree)	$X/Y : f \quad (Y/Z_1)/Z_2 : \lambda z_1 \lambda z_2.gz_1z_2 \quad \Rightarrow \quad (X/Z_1)/Z_2 : \lambda z_1 \lambda z_2.fz_1z_2$
$<\mathbf{B}^2_\times$	Generalised Backward Crossed Composition (2nd degree)	$(Y/Z_1)/Z_2 : \lambda z_1 \lambda z_2.gz_1z_2 \quad X\backslash Y : f \quad \Rightarrow \quad (X/Z_1)/Z_2 : \lambda z_1 \lambda z_2.fz_1z_2$
tc-rel	Binary Conversion / Type Changing	, S[to|adj|pss|ng]\NP $\Rightarrow$ $(NP\backslash NP)$
tc-nmod	Binary Conversion / Type Changing	$X/X \quad (X/X)\backslash(X/X) \Rightarrow (X/X)\backslash(X/X)$
tr-np2	Type Raising	NP $\Rightarrow$ S/(S\NP)
tr-np2	Type Raising	NP $\Rightarrow$ (S\NP)\((S\NP)/NP)
tr-pp	Type Raising	PP $\Rightarrow$ (S\NP)\((S\NP)/PP)
lex-n-np	Conversion	N $\Rightarrow$ NP
lex-rel-np	Conversion	S[pss|ng|adj|to]\NP $\Rightarrow$ NP\NP
lex-rel-n	Conversion	S[to]\NP $\Rightarrow$ N\N
lex-rel-dcl	Conversion	S[dcl]/NP $\Rightarrow$ NP\NP
lex-mod	Conversion	S[pss|ng|to]\NP $\Rightarrow$ S/S
lex-dcl-n	Conversion	S[dcl]/NP $\Rightarrow$ N\N
lex-pp-fn-np	Conversion	PP $\Rightarrow$ NP\NP
lex-pp-fn-vp	Conversion	PP $\Rightarrow$ (S\NP)\(S\NP)

Table 3.1.: Definitions of the combinators used in this thesis. Definitions follow Lewis and Steedman (2014); the semantic definitions are taken from Steedman (2000). For conversions, we give no semantic definition since Lewis and Steedman (2014) only define them syntactically. The conversions **tc-rel**, **tc-nmod**, **lex-dcl-n**, **lex-pp-fn-np**, and **lex-pp-fn-vp** have been added for this thesis.

Example 3.4. Figure 3.1 shows a derivation of the phrase *the cat saw the dog*. Lexical entries such as "the := N : $\lambda x.\, x$" are the leaves of the derivation. Their parent nodes are created by applying a matching combinator to the syntactic and semantic categories of neighbouring nodes.

We call the elements of a derivation *nodes*, since the derivation can be viewed as a tree with the lexical entries at the leaves. Therefore, each node in the derivation represents a choice: the leaves correspond to the choice of lexical entry to represent a word, and the nodes above them each represent the choice of a combinator. Likewise, there are two types of derivation node: leaves and combinatory nodes.

Definition 3.5 (Derivation Node)**.** The set of *derivation nodes* is made up of *leaf nodes* and *combinatory nodes*, defined as follows:

- Let w be a word, c_{syn} a syntactic category, and c_{sem} a semantic category. Then the tuple (w, c_{syn}, c_{sem}) is a *leaf node*.
- Let C be a combinator, c_{syn} a syntactic category, c_{sem} a semantic category, and $n_1, \ldots, n_k$ a list of derivation nodes, where k is the number of arguments accepted by C. Then the tuple $(C, c_{\text{syn}}, c_{\text{sem}}, (d_1, \ldots, d_k))$ is a *combinatory node*.

For a derivation node d, we define the following accessor functions:

$$\textsc{tok}(d) = \begin{cases} w & \text{if } d \text{ is a leaf node} \\ \textsc{tok}(d_1) \ldots \textsc{tok}(d_k) & \text{if } d \text{ is a combinatory node} \end{cases}$$
$$\textsc{syn}(d) = c_{\text{syn}}$$
$$\textsc{sem}(d) = c_{\text{sem}}$$

The following accessors are only defined for combinatory nodes:

$$\textsc{cmb}(d) = C$$
$$\textsc{children}(d) = d_1, \ldots, d_k$$

3.1.6. Parsing CCG

In terms of its generative power, CCG is a *mildly context-sensitive* language, and thus more expressive than context-free languages (Vijay-Shanker and Weir 1994).

Algorithm 3.1 A variant of the CKY algorithm for CCG parsing.

Inputs:

- A sequence of words $w_1, \ldots, w_n$
- A lexicon L
- A set of combinators $\mathcal{C}$

Output: A parse chart Y containing the derivation nodes

$Y[i, j] \leftarrow \{\}$ for $1 \leq i \leq j \leq n$
for $j \leftarrow 1, \ldots, n$ **do**
  for $(w, c_{\text{syn}}, c_{\text{sem}}) \in L$ with $w = w_j$ **do**
    $Y[j, j] \leftarrow Y[j, j] \cup \{(w, c_{\text{syn}}, c_{\text{sem}})\}$ ▷ Insert leaf nodes
  end for
  for $i \leftarrow j - 1, \ldots, 1$ **do** ▷ Iterate over start index
    for $k \leftarrow i + 1, \ldots, j$ **do** ▷ Iterate over split index
      for $e_1, e_2 \in Y[i, k - 1] \times Y[k, j]$ **do**
        for $C \in \mathcal{C}$ **do**
          if $C(\textsc{syn}(e_1), \textsc{syn}(e_2))$ is defined **then**
            $e \leftarrow (C,$
              $\textsc{tok}(e_1)\textsc{tok}(e_2),$
              $C(\textsc{syn}(e_1), \textsc{syn}(e_2)),$ ▷ Create derivation node
              $C(\textsc{sem}(e_1), \textsc{sem}(e_2)),$
              $(e_1, e_2))$
            $Y[i, j] \leftarrow Y[i, j] \cup e$
          end if
        end for
      end for
    end for
  end for
end for

Nevertheless, parsing algorithms designed for context-free grammars are commonly used to parse CCG, including chart-based bottom-up parsing algorithms derived from the well-known CKY algorithm (Cocke and Schwartz 1970; Kasami 1965; Younger 1967).

There is also literature on parsing algorithms which do not map CCG parsing onto a context-free parsing algorithm, but model the structure of CCG derivations directly (for an example, see Kuhlmann and Satta (2014)). These algorithms are however significantly more complex than the other two classes of algorithms and are rarely used in statistical parsing research.

The experiments performed in this thesis use a CKY-style parsing algorithm. Algorithm 3.1 shows an adaptation of CKY to CCG. The algorithm constructs a *chart* Y to collect derivation nodes. Y is a two-dimensional array that, for every pair of word indices, contains the set of derivation nodes that can be inferred for the subsequence of the input sentence indicated by the indices.

The content of the cells that are located on the chart's diagonal (corresponding to individual words) is made up of leaf nodes, which are drawn directly from the lexicon. Other chart cells are filled by applying a combinator to the entries located in neighbouring chart cells. To iterate over all pairs of adjacent chart cells, three indices are used: a start index i, a split index k, and an end index j. For every combination of indices, pairs of derivation nodes from $Y[i, k-1]$ and $Y[k, j]$ are drawn from the chart. The syntactic categories of both entries are matched to the patterns specified by the grammar's combinatory rules. If an applicable combinator is found, an inner derivation node is created and written to the chart.

After the algorithm terminates, all possible derivations for the entire sentence are found in the chart cell $Y[1, j]$. The algorithm is slightly simplified in that only binary combinators are considered, but unary combinators such as type raising can be trivially applied on the level of each cell. It is extended for GA-CCG parsing in Chapter 7.

3.2. Graph Algebras for Semantic Construction

Semantic construction is to the process of combining meaning fragments to form a complete meaning representation. On the most abstract level, a *calculus* or *algebra* is required which operates on meaning fragments. The most prominent example might be the use of λ-calculus to build expressions of first order logic, which goes back to Montague (1973) and continues to be widely used, including in CCG. Meaning fragments are represented as functions which take other fragments as

arguments. Since any computation can be performed using λ-calculus, it is an extremely powerful construction mechanism.

The power of λ-calculus has been very useful in the study of formal and computational semantics, allowing complex and creative solutions to linguistic problems. However, this complexity can be a liability when designing machine learning systems: A semantic construction mechanism forms part of the problem statement that a semantic parser is optimised to solve, and the degrees of freedom it permits can put an additional burden on the learning algorithm. In contrast, a well-designed, carefully restricted mechanism can impose a useful prior that allows learning to proceed more efficiently and effectively by limiting the degrees of freedom and thus the size of the solution space that has to be searched.

The task of semantic construction can be formalised by defining an algebra that operates on meaning fragments. The operations of such an algebra take meaning fragments as arguments and produce either new fragments or complete meaning representations. By repeatedly applying these operations, a set of initial meaning fragments can thus be combined into a single, complete meaning representation. In the following section, we examine such an algebra.

3.2.1. The HR Algebra

The *HR algebra* is an algebra for the construction of graphs, named for its relationship with hyperedge replacement grammars[1]. It has first been applied to the construction of AMRs by Koller (2015), who employed the HR algebra in the context of Interpreted Regular Tree Grammars (IRTGs) and showed that phenomena such as complements and modifiers can be elegantly modeled in this way. The IRTG approach has been extended to a full statistical AMR parser (Groschwitz, Fowlie et al. 2017).

The HR algebra defines operations over *graphs with sources* or *s-graphs*. s-graphs are graphs augmented with a labelling function which assigns a *source label* to some nodes, marking them as *sources*. The most important operation of the HR algebra, *parallel composition*, combines two s-graphs by merging the nodes of both graphs that share the same source label. Additional operations permit the renaming and deletion of source labels. Source labels are drawn from a set $\mathcal{A}$, and each source label may appear at most once in an s-graph.

In the case of AMR parsing, AMR graphs are augmented with source labels to represent meaning fragments. Relatively small graphs, representing the meaning

[1]There is an equivalence between equational sets of the HR algebra and hyperedge replacement grammars. For more details, see Courcelle and Engelfriet (2012).

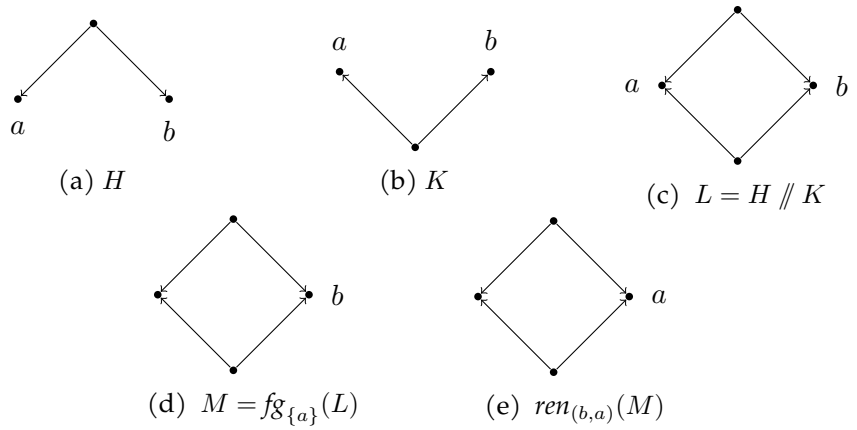


Figure 3.2.: Examples for basic operations on s-graphs. H and K are merged (c), then source a is forgotten (d), and finally, source b is renamed to a (e).

of individual words, are drawn from a lexicon and recursively combined into full AMRs according to the rules of a grammar, such as the GA-CCG described in Chapter 4.

Example 3.6 (HR Algebra)**.** The HR algebra's operations are illustrated in Figure 3.2. The example starts with the s-graphs H and K. Both graphs contain an a-source and a b-source. In the next step, both graphs are merged using parallel composition, which unites the a-source of H with the a-source of K and the b-source of H with the b-source of K, yielding the new graph L. Then, the a-source is forgotten, and the b-source renamed to an a-source.

3.2.2. Formal Definition

In this section, the HR algebra is defined in mathematical terms. Our definitions follow those given by Courcelle and Engelfriet (2012), although some have been slightly simplified.

Definition 3.7 (s-graph)**.** Let $G^\circ = (V_{G^\circ}, E_{G^\circ})$ be a graph and $\mathcal{A}$ a countable set of labels. The tuple $G = (G^\circ, \mathit{slab}_G)$ is called an *s-graph*, where $\mathit{slab}_G : V_{G^\circ} \to \mathcal{A}$ is a partial injective function from vertices to labels.

The domain of slab_G is called $\mathit{Src}(G)$, the *set of sources of* G.

For simplicity, we will write V_G for V_{G° when referring to an s-graph's vertices, and E_G instead of E_{G° for its edges.

Given two s-graphs G, H, we write $H \subseteq G$ if $H^\circ \subseteq G^\circ$ and $\mathit{slab}_H = \mathit{slab}_G|V_H$ (that is, slab_H agrees with slab_G but its domain is limited to the nodes in V_H).

Parallel composition is the operation of taking the union of two graphs that share a set of source nodes. This definition may seem counterintuitive, as we are interested in composing not only graphs that already share some nodes, but arbitrary pairs of s-graphs. However, we can always substitute an s-graph for an isomorphic one which fulfils the required sharing property. In the following, we treat equally labelled sources in separate graphs as if they refer to the same node and assume that such a substitution to take place implicitly.[2]

Definition 3.8 (Parallel composition)**.** Let G, H, K be s-graphs. G is the *parallel composition* of H and K (written: $G = H /\!\!/ K$) iff. all of the following hold:

1. $H \subseteq G, K \subseteq G, G^\circ = H^\circ \cup K^\circ$ (G is the union of H and K, and the source labels of H and K agree with G)
2. $V_H \cap V_K = \mathit{Src}(H) \cap \mathit{Src}(K)$ (H and K share no vertices except for the sources they have in common)
3. $E_H \cap E_K = \emptyset$ (H and K share no edges)
4. $\mathit{slab}_G = \mathit{slab}_H \cup \mathit{slab}_K$ (G contains no sources other than those that are also in H and K)

The remaining two operations, renaming and forgetting, act on a single s-graph's source labels without changing its structure.

Definition 3.9 (Forgetting sources)**.** Let $G = (G^\circ, \mathit{slab}_G)$ be an s-graph, $\mathcal{A}$ a set of source labels, and $A \subseteq \mathcal{A}$. The *forget* operator $f\!g_A$ is defined as follows:

$$f\!g_A(G) = (G^\circ, \mathit{slab}'_G) \text{ where } \mathit{slab}'_G(n) = h \circ \mathit{slab}_G$$
$$h(l) = l \text{ if } l \notin A$$

The partial function h limits the domain of slab_G to the sources outside of A. In effect, all source labels in A are thus removed from the s-graph. Thus, the a-sources of G cease to be sources for $a \in A$, but all other sources remain unchanged.

[2] This substitution is formalised by Courcelle and Engelfriet (2012), who introduce the concept of *abstract s-graphs*, each of which represents a class of isomorphic graphs.

Definition 3.10 (Renaming sources)**.** Let $f : \mathcal{A} \to \mathcal{A}$ be a permutation of $\mathcal{A}$. The *rename* operator ren_f is defined as follows:

$$ren_f(G) = (G^\circ, slab'_G) \text{ where } slab'_G = f \circ slab_G$$

By applying a permutation f to the output of the source labelling function, ren_f allows source labels to be modified, but does not introduce or remove sources.

3.2.3. Multiple Source Labels per Node

Because *slab* has been defined as a function, every node in an s-graph can be assigned at most one source label. However, in the context of semantic construction, a node could play several roles, for example both as the root of a meaning representation and an argument slot. We therefore extend the HR algebra by relaxing this requirement and define s*-graphs, where a node may be assigned any number of source labels. The source labelling function *slab* therefore becomes set-valued, and the definitions require a slightly different notation.

Definition 3.11 (s*-graph)**.** Let $G^\circ = (V_{G^\circ}, E_{G^\circ})$ be a graph and $\mathcal{A}$ a countable set of labels. The tuple $G = (G^\circ, slab_G)$ is called an *s*-graph*, where $slab_G : V_{G^\circ} \to \mathcal{P}(\mathcal{A})$ is a function assigning a set of source labels to every node, fulfilling the property: $|\{v | a \in slab_G(v)\}| \leq 1$ for all $a \in \mathcal{A}$ (*every label is assigned to at most one node*).

For brevity, we write V_G for V_{G° and V_H for V_{H°.

The set $\{v \in V_G \mid |slab_G(v)| > 0\}$ is called $Src(G)$, the *set of sources of* G.

Given two s*-graphs G, H, we write $H \subseteq G$ if $H^\circ \subseteq G^\circ$ and $slab_H(v) \subseteq slab_G(v)$ for all $v \in V_H$.

In place of the injectivity constraint that is required for the *slab*-function of an s-graph, a constraint is introduced that requires every source label to be assigned to at most one node in the graph.

Definition 3.12 (Parallel composition of s*-graphs)**.** Let G, H, K be s*-graphs. G is the *parallel composition* of H and K (written: $G = H \mathbin{/\!\!/} K$) iff. all of the following hold:

1. $H \subseteq G, K \subseteq G, G^\circ = H^\circ \cup K^\circ$ (G is the union of H and K, and the source labels of H and K agree with G)
2. $V_H \cap V_K = Src(H) \cap Src(K)$ (H and K share no vertices except for the sources they have in common)

3. $E_H \cap E_K = \emptyset$ (H and K share no edges)

4. $slab_G(v) = \begin{cases} slab_H(v) \cup slab_K(v) & \text{if } v \in V_{H^\circ} \cap V_{K^\circ} \\ slab_H(v) & \text{if } v \in V_{H^\circ} \backslash V_{K^\circ} \\ slab_K(v) & \text{if } v \in V_{K^\circ} \backslash V_{H^\circ} \end{cases}$
(G contains no sources other than those also in H and K)

Definition 3.13 (Forgetting sources of s*-graphs)**.** Let $G = (G^\circ, slab_G)$ be an s*-graph, $\mathcal{A}$ a set of source labels, and $A \subseteq \mathcal{A}$. The *forget* operator fg_A is defined as follows:

$$fg_A(G) = (G^\circ, slab'_G) \text{ where } slab'_G(v) = slab_G(v) \backslash A$$

Definition 3.14 (Renaming sources of s*-graphs)**.** Let $f : \mathcal{A} \to \mathcal{A}$ be a permutation of $\mathcal{A}$. The *rename* operator ren_f is defined as follows:

$$ren_f(G) = (G^\circ, slab'_G) \text{ where } slab'_G(v) = \{f(l) | l \in slab_G(v)\}$$

While not described by Courcelle and Engelfriet (2012), an operator to invent sources in an s*-graph can also straightforwardly be defined.

Definition 3.15 (Adding sources to s*-graphs)**.** Let $G = (G^\circ, slab_G)$ be an s*-graph with $G^\circ = (V_G^\circ, E_G^\circ)$. Let $f : V_G^\circ \to \mathcal{A}$ be a partial injective source labelling function. The *add* operator add_f is defined as follows:

$$add_f(G) = (G^\circ, slab'_G)$$

where

$$slab'_G(v) = \begin{cases} slab_G(v) \cup f(v) & \text{if } v \in Dom(f) \\ slab_G(v) & \text{otherwise} \end{cases}$$

s*-graphs are the fundamental semantic structures used by the semantic parsing system developed in this thesis. In 4, we describe how they are embedded into a syntactic-semantic grammar.

3.3. Linear Models of Linguistic Structures

Machine learning tasks are commonly categorised into classification tasks, where labels are drawn from a closed set of classes, and regression tasks, where labels are real-valued. Many tasks involving natural language, however, deal with *structure prediction*, where labels have a complex structure. Structure prediction encompasses tasks such as sequence tagging, where not just individual labels but sequences thereof form the output of the model. In addition to sequence tagging, parsing is another example of structure prediction, where labels take the form of parse trees. In AMR parsing, sentences are labelled with AMR graphs.

Whereas classification algorithms usually work by assigning every class a score and then selecting the class with the highest scores, this is not feasible in structure prediction as there are usually too many labels to enumerate. For instance, the number of possible parse trees of a sentence may be exponential, and there is an infinite number of valid AMR graphs.

Instead of predicting a label in an atomic step, in structure prediction the task of producing an output label is broken down into decisions. For example, a parser is commonly said to be in a certain state, based on its past decisions. At every state, there is a number of actions the parser can take, and a binary classification algorithm can be used to score each action. The parser then performs the most highly-scored action to advance to the next state according to its parsing algorithm. Linguistic structure prediction therefore involves two parts: a *learning* component, and a *decoding* component.

For instance, when parsing a sentence with the CKY algorithm for CCG parsing (see Algorithm 3.1), the first decisions to be made involve selecting a syntactic category for each token. Further decisions involve whether or not to combine two neighbouring constituents using a given combinator. Each of these decisions can be scored, finally producing the parse tree with the highest combined score. To perform this decoding task efficiently, a beam search algorithm may be used (Section 3.3.1).

As an implementation for the learning algorithm, we describe the *structured perceptron*, which we use to score decisions in our GA-CCG parser. We start by introducing the perceptron algorithm for classification (Section 3.3.2), before describing its use for structured prediction (Section 3.3.3).

In this section, we introduce the fundamental concepts and algorithms needed for training a GA-CCG parser, but do not yet unify them into a complete parsing system. A full implementation of a GA-CCG parser is presented in Chapter 7.

3.3.1. CKY Parsing with Beam-Search

An algorithm for constructing CCG derivations has been introduced in Section 3.1.6 (Algorithm 3.1). This algorithm is complete: it enumerates all derivations of the input sentence that are possible under the given grammar.

The total number of derivations of a sentence may be too large to enumerate in practice. At the same time, it is usually desirable to not just enumerate derivations, but also express a preference among them. *Beam search* can be applied to perform inexact search while considering a scoring function of parses. It works by limiting the number of hypotheses considered at each step of the algorithm (Lowerre 1976). In CCG parsing, this means that each cell $Y[i, j]$ of the parse chart is pruned to a maximum of k entries, where k is the *beam size*. Which entries are kept is determined by a scoring function, which usually is the output of a machine learning model.

An adaptation of the CKY algorithm for CCG with beam search is shown in Algorithm 3.2. After all available hypotheses for a given chart cell $C[i, j]$ have been enumerated, the function max$k$ is applied, which limits that cell's contents to those k entries that are assigned the highest scores by the scoring function.

In the remainder of this section, we examine how the *structured perceptron* algorithm can be used to train a scoring function which can be used with the beam-search CKY algorithm.

3.3.2. The Perceptron Algorithm

The perceptron algorithm is a discriminative machine learning algorithm which searches for a hyperplane separating positive and negative examples. It is one of the simplest and oldest classification algorithms (Rosenblatt 1958).

In classification as well as structure prediction, predictions $y \in \mathcal{Y}_x$ for examples x are represented by feature vectors $\Phi(x, y)$. The perceptron algorithm trains a weight vector $\mathbf{w}$ of the same dimension as the feature vector, which is used to produce a *score*:

$$\textsc{Score}(x, y) = \mathbf{w}^\top \Phi(x, y) \tag{3.2}$$

The model makes a prediction for x by choosing y such as to maximise $\textsc{Score}(x, y)$.

To obtain w, the perceptron algorithm uses an *online learning* procedure, which repeatedly iterates over the data set. For every example, a prediction is made. If the prediction is incorrect, the weights are updated by adding the features for the annotated label and subtracting the features for the incorrect prediction, as shown

Algorithm 3.2 An algorithm for CCG parsing with beam search.

Inputs:

- A sequence of words $w_1, \ldots, w_n$
- A beam size k
- A lexicon L
- A set of combinators $\mathcal{C}$
- A scoring function for derivation nodes SCORE

Output: A parse chart Y containing the derivation nodes

$Y[i, j] \leftarrow \{\}$ for $1 \leq i \leq j \leq n$
for $j \leftarrow 1, \ldots, n$ **do**
  for $(w, c_{\text{syn}}, c_{\text{sem}}) \in L$ with $w = w_j$ **do**
    $Y[j, j] \leftarrow Y[j, j] \cup \{(w, c_{\text{syn}}, c_{\text{sem}})\}$ ▷ Insert leaf nodes
  end for
  $Y[j, j] \leftarrow \max k_{\text{SCORE}(x)}\{x \in Y[j, j]\}$ ▷ Enforce beam limitation
  for $i \leftarrow j - 1, \ldots, 1$ **do** ▷ Iterate over start index
    for $k \leftarrow i + 1, \ldots, j$ **do** ▷ Iterate over split index
      for $e_1, e_2 \in Y[i, k - 1] \times Y[k, j]$ **do**
        for $C \in \mathcal{C}$ **do**
          if $C(\text{SYN}(e_1), \text{SYN}(e_2))$ is defined **then**
            $e \leftarrow (C,$
              $\text{TOK}(e_1)\text{TOK}(e_2),$
              $C(\text{SYN}(e_1), \text{SYN}(e_2)),$ ▷ Create derivation node
              $C(\text{SEM}(e_1), \text{SEM}(e_2)),$
              $(e_1, e_2))$
            $Y[i, j] \leftarrow Y[i, j] \cup e$
          end if
        end for
      end for
    end for
    $Y[i, j] \leftarrow \max k_{\text{SCORE}(x)}\{x \in Y[i, j]\}$ ▷ Enforce beam limitation
  end for
end for

in Algorithm 3.3. This pushes the weight vector towards achieving the correct classification of x.

Algorithm 3.3 The basic perceptron algorithm.

Inputs:
- A data set $(\tilde{x}_1, \tilde{y}_1), \ldots, (\tilde{x}_N, \tilde{y}_N)$
- An iteration count T
- An initial weight vector $\mathbf{w}_{\text{init}}$

Output: A weight vector $\mathbf{w}_t$

$\mathbf{w}_N^0 \leftarrow \mathbf{w}_{\text{init}}$
for $t \leftarrow 1 \ldots T$ **do**
  $\mathbf{w}_0^t \leftarrow \mathbf{w}_N^{t-1}$
  for $i \leftarrow 1 \ldots N$ **do**
    $y \leftarrow \text{argmax}_{y' \in \mathcal{Y}} {\mathbf{w}_{i-1}^t}^\top \Phi(\tilde{x}_i, y')$
    if $y \neq \tilde{y}_i$ **then**
      $\mathbf{w}_i^t \leftarrow \mathbf{w}_{i-1}^t + \Phi(\tilde{x}_i, \tilde{y}_i) - \Phi(\tilde{x}_i, y)$
    else
      $\mathbf{w}_i^t \leftarrow \mathbf{w}_{i-1}^t$
    end if
  end for
end for
return $\mathbf{w}_N^T$

Since the decision function is a linear function of the feature vectors, the decision boundary of the perceptron is a hyperplane that separates positive and negative examples. The perceptron algorithm can be shown to find a separating hyperplane if the training data are linearly separable, and to make a bounded number of errors if the training data are not separable (Collins 2002). Its convergence can be further improved by using an adaptive learning rate.

Adadelta Updates

While the standard perceptron update is theoretically guaranteed to converge, it may take a long time to do so. One reason is that the parameters for rare features are updated less frequently, which is not accounted for by the perceptron algorithm and leads to a low rate of convergence for the corresponding parameters. To present the

Adadelta method, we first define the gradient at update step t as follows (compare Algorithm 3.3 line 7):

$$g_t = \Phi\tilde{x}_t, \tilde{y}_t - \Phi\tilde{x}_t, y \tag{3.3}$$

The Adadelta algorithm speeds up convergence by applying individual, dynamic learning rates to every parameter (Zeiler 2012). It keeps track of an accumulated, decaying average of squares of gradients $E[g^2]$.

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2 \tag{3.4}$$

In the same manner, all updates Δx made by the algorithms are accumulated.

$$E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2 \tag{3.5}$$

The update is calculated by scaling the perceptron update by the ratio of these quantities:

$$\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t} g_t \tag{3.6}$$

where

$$\text{RMS}[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{3.7}$$

Since the accumulated gradients and updates are squared, the RMS function effectively computes a root of mean squares.

The hyperparameter $\epsilon > 0$ should be a small positive value, while $0 < \rho < 1$ is set to 0.95 in Zeiler (ibid.).

3.3.3. Structured Perceptron

The perceptron requires evaluating all possible labels to decide which label to predict (Algorithm 3.3 line 5). In the case of structured prediction, this is infeasible since the set of available labels $\mathcal{Y}(x)$ depends on x and may be exponential in the length of x (for instance, for a syntax parser, $\mathcal{Y}(x)$ contains the valid syntax trees for x).

In the structured perceptron, the argmax of Algorithm 3.3 is therefore implemented as a search in the space of labels, often implemented using dynamic programming algorithms such as the CKY algorithm (see Algorithm 3.1). This is possible if

the feature function Φ can be decomposed over the internal structure of the label into local feature vectors ϕ_i such that $\Phi(x) = \sum_{i=1}^{n} \phi_i(x)$ for some decomposition of the label into n parts (Collins 2002).[3]

For example, in syntax parsing, features can be extracted not only from complete derivations, but also for sub-trees of a derivation corresponding to a part of the sentence. By multiplying these feature vectors with the weight vector, sub-derivations can be scored and the CKY algorithm can recursively construct a complete derivation from the highest-scoring parts.

3.3.4. Cost-Sensitive Perceptron

The standard perceptron uses only two outputs to compute an update. The underlying assumption is that the correct output is known and the computed update will therefore improve the model. However, with structured output spaces, the correct output is not necessarily accessible to the model, for example because no derivation of the correct output can be found. In *cost-sensitive learning*, we accept that the model may not be able to produce a perfect result, and instead update towards the best available output, as determined by a cost function.

A simple instance of a cost-sensitive perceptron algorithm has been presented by Singh-Miller and Collins (2007), which is shown in Algorithm 3.4. It enforces a margin between the best output and all other outputs, where the size of the margin of each output depends on the cost difference.

At the beginning of each iteration, the algorithm obtains a set of predictions $\mathcal{Y}(\tilde{x}_i)$. These predictions are separated into a *good set* G which consists of the predictions with minimal cost, and a *bad set* B comprising all others.

The goal of the algorithm is to enforce a margin (a minimum score difference) between the predictions in G and the predictions in B. The required margin is defined as $\lambda\Delta(y)$ for any prediction y, where $\Delta(y)$ is the cost difference between y and any good state, and λ is a scaling parameter.

Two sets of margin-violating predictions are defined: C is the set of good predictions which are not sufficiently separated from some bad prediction, and E is the set of bad predictions which violate the margin regarding some good prediction. To increase the margin for the violating predictions, a positive update is computed from C and a negative update from E.

Both the positive and negative updates are weighted averages of the respective predictions' feature vectors. All predictions in C are weighted equally. In contrast,

[3] In practice, the decomposition of the feature function need not be perfect, as beam search allows for a margin of error.

Algorithm 3.4 The cost-sensitive perceptron algorithm by Singh-Miller and Collins (2007).

Inputs:

- A data set $(\tilde{x}_1, \tilde{y}_1), \ldots, (\tilde{x}_N, \tilde{y}_N)$
- An iteration count T
- An initial weight vector $\mathbf{w}_{\text{init}}$
- A feature extractor $\Phi(y)$
- A cost function $\text{cost}(y)$
- A margin scaling factor λ

Output: A weight vector $\mathbf{w}_t$

$\mathbf{w}_N^0 \leftarrow \mathbf{w}_{\text{init}}$
for $t \leftarrow 1 \ldots T$ **do**
  $\mathbf{w}_0^t \leftarrow \mathbf{w}_N^{t-1}$
  for $i \leftarrow 1 \ldots N$ **do**
    $\mathit{cost}_{\text{min}} \leftarrow \min_{y \in \mathcal{Y}(\tilde{x}_i)} \text{cost}(y)$
    $\Delta(y) \leftarrow \text{cost}(y) - \mathit{cost}_{\text{min}}$ for all $y \in \mathcal{Y}(\tilde{x}_i)$
    $G \leftarrow \{g \mid g \in \mathcal{Y}(\tilde{x}_i), \Delta(g) = 0\}$
    $B \leftarrow \mathcal{Y}(\tilde{x}_i) \backslash G$
    $C \leftarrow \{c \mid c \in G; \exists z : z \in B, \mathbf{w}_{i-1}^t(\Phi(c) - \Phi(z)) < \lambda\Delta(z)\}$
    $E \leftarrow \{e \mid e \in B; \exists y : y \in G, \mathbf{w}_{i-1}^t(\Phi(y) - \Phi(e)) < \lambda\Delta(e)\}$
    $\tau(e) \leftarrow \sum_{c \in C} \frac{v_c(e)}{|C| \sum_{e' \in E} v_c(e)}$ for all $e \in E$
      where $v_c(e) = \begin{cases} 1 & \text{if } \mathbf{w}_{i-1}^t(\Phi(c) - \Phi(e)) < \lambda\Delta(e) \\ 0 & \text{otherwise} \end{cases}$
    $w_i^t \leftarrow w_{i-1}^t + \sum_{c \in C} \frac{\Phi(c)}{|C|} - \sum_{e \in E} \tau(e)\Phi(e)$
  end for
end for
return $\mathbf{w}_N^T$

every prediction in E is weighted by the number of C-predictions regarding which it is in violation. Consequently, the computed update is added to the weight vector.

An important difference to the perceptron approaches discussed so far is that in the cost-sensitive perceptron, several outputs of the model are considered in a weighted-average update. In contrast, both the standard perceptron and the early update perceptron compute the update from a single pair of outputs. Arguably, this allows the algorithm to more effectively use the information contained within the prediction list.

3.3.5. Minibatch Training

As an online learning algorithm, the perceptron algorithm processes examples sequentially. To derive an update for an example, it performs inference – for instance, using a CKY parsing algorithm, which can require a large amount of computation time. It is desirable to parallelise these computations to make use of parallel and distributed computer architectures.

Fortunately, it has been shown that *minibatching*, a simple method for parallelising the training of online learners, yields good speed-ups while preserving the convergence behaviour of perceptrons (Zhao and Huang 2013). In minibatching, updates are collected for several examples at a time and then summed and applied collectively. Inference for these examples can thus be performed in parallel. While this method slightly alters the learning behaviour since the same weight vector is used for all predictions of the same minibatch, this does not harm the algorithm's convergence behaviour for moderate batch sizes.

3.4. Expectation Maximisation and the Inside-Outside Algorithm

Expectation Maximisation (*EM*) is a class of local optimisation algorithms. The name is derived from the common structure of these algorithms, which alternate between two steps. In the estimation step, expected counts of model events are computed across the data set based on the model parameters. In the maximisation step, the model parameters are optimised to maximise the probability of the expected model events. These two steps are alternated until the parameters converge.

EM is usually applied when a hidden process – such as a probabilistic grammar – is assumed to have generated the data, but the process cannot be observed. One application for EM is parameter estimation for probabilistic context-free grammars

(PCFGs). Given a corpus of text and a context-free grammar (CFG), the task is to assign each CFG rule a probability such that the probability of the training data is maximised. The grammar is thus tuned to represent the linguistic structures found in the training data.

When used for PCFG estimation, the expectation step of the EM algorithm is implemented using the *inside-outside algorithm* (Baker 1979; Lari and Young 1990). It calculates expectations for the number of occurrences of each CFG rule in a given example.

EM for PCFG estimation can be interpreted as a grammar induction method. In the beginning, a large number of PCFG rules are added to the grammar. During the expectation step, rules that are applicable frequently across the corpus are assigned a high expected count. The maximisation step then assigns these rules a higher probability, whereas rare rules are assigned low probabilities. If a rule is not necessary, for instance because it can always be replaced by a more general rule, its probability converges to zero. The EM algorithm can therefore be used to weed out unnecessary rules from an overgenerating grammar.

3.4.1. PCFG Parameter Estimation

Context-free grammars (CFGs) are rule systems that produce sequences of symbols according to rewriting rules which translate non-terminals – starting with a special symbol S – into other non-terminals or terminal symbols. In the application to natural language grammar, terminal symbols are words and the context-free production rules define the syntax of the natural language.

Definition 3.16 (Context-Free Grammar)**.** A *context-free grammar* is a tuple (N, Σ, R, S) where N is a set of nonterminals, Σ is a set of terminals, R is a set of rules, and $S \in N$ is a start symbol.

Without loss of generality, we assume that CFGs are in Chomsky normal form, so that rules take one of the following forms:

- $A \to BC$ where $A, B, C \in N$
- $A \to t$ where $t \in \Sigma$

As practical context-free grammars for natural languages are ambiguous, parsers require a mechanism to prefer one parse over the other and thus output the most plausible parse for a sentence. Probabilistic context-free grammars are an extension where every rule of a CFG is associated with a probability.

Definition 3.17 (Probabilistic Context-Free Grammar)**.** Let $G = (N, \Sigma, R, S)$ be a context-free grammar and $q : R \to [0, 1]$ a probability distribution over R with $\sum_{r \in R} q(r) = 1$. The tuple (G, q) is called a *probabilistic context-free grammar* (PCFG).

To estimate PCFG parameters, the inside-outside algorithm can be used in combination with expectation maximisation. The high-level iterative structure of the corresponding EM algorithm is shown in Figure 3.5. For a given number of iterations, the algorithm alternates between an expectation and a maximisation step. In the expectation step, the algorithm collects expected counts for each PCFG rule across all examples. The expected rule counts express how frequently a rule is expected to occur in the derivation for an example given the current model parameters. In the maximisation step, new model parameters are computed from the expected counts. Since the model is probabilistic, this step can be implemented by maximum likelihood estimation, that is, normalising the counts to form a probability distribution.

The computation of the expected counts is performed by the inside-outside algorithm, which is shown in Figure 3.6. The algorithm computes two types of probabilities for a sentence $x_1, \ldots, x_n$:

- The inside probability *inside*(A, i, j) represents the probability that tokens $x_i, \ldots, x_j$ are derived from the non-terminal A.
- The outside probability *outside*(A, i, j) represents the combined probability of all partial derivations which lead to the non-terminal A spanning tokens $x_i, \ldots, x_j$, but do not include the sub-tree of A.

The values of inside probabilities on the token level are known: They correspond to the parameters of the terminal rules which produce the tokens x_1 through x_n. The remaining inside probabilities can be computed bottom-up by considering the parameters of non-terminal rules which might have produced a given non-terminal.

Outside probabilities are known on the sentence level, since there is no outside context to be included. Since S is defined as the start symbol, the probability for its occurrence at the root of the derivation is 1, while it is 0 for all other non-terminals. Outside probabilities are then propagated downwards by probabilistically modelling the application of binary rules: The outside score of a non-terminal is the sum of the outside scores of all potential parents, multiplied by the inside scores of all potential siblings and the scores of the corresponding rules.

To calculate the expected count for each rule, rule probabilities are derived from the inside and outside scores by summing over all possible rule applications: for

Algorithm 3.5 The EM algorithm for estimating PCFG parameters.

Inputs:
- A CFG (N, Σ, R, S)
- A Corpus $X = (x_1 \dots x_n)$
- An iteration count T
- An initial probability distribution $q^0 : R \to [0, 1]$

Output: A probability distribution $q^T : R \to [0, 1]$

for $k \leftarrow 1 \dots T$ **do**
  for all $r \in R$ **do** ▷ Initialise counts with zero
    $\mathit{count}(r) \leftarrow 0$
  end for
  for $i \leftarrow 1 \dots n$ **do** ▷ Expectation step: aggregate rule counts
    $c \leftarrow \textsc{Counts}(x_i, q^{k-1})$
    for all $r \in R$ **do**
      $\mathit{count}(r) \leftarrow \mathit{count}(r) + c(r)$
    end for
  end for
  for all $r \in R$ **do** ▷ Maximisation step: normalise rule counts
    $q^k(r) \leftarrow \frac{\mathit{count}(r)}{\sum_{r' \in R} \mathit{count}(r')}$
  end for
end for
return q^T

terminal rules, these are all tokens which match the token produced by the rule, and for non-terminal rules, they are all sub-sequences of the sentence of length two or greater. The probability is then scaled by the inverse probability assigned to the entire sentence to obtain a count.

The EM algorithm iteratively optimises the total probability assigned to the data set by shifting probability mass towards the rules which are the most useful in the explanation of the corpus. While Algorithm 3.5 uses a fixed number of iterations, it can also be stopped when the total probability of the corpus converges.

In chapter 6, we derive an EM algorithm for the filtering of GA-CCG lexica, using EM with the inside-outside algorithm as a starting point. While the EM algorithm is almost identical, inside and outside probabilities are computed differently due to the different generative process underlying GA-CCG derivations. The details of the algorithm are explained in Section 6.2.

3.5. Neural Network Models for Sequence Tagging

Sequence taggers are tools which assign a label from a closed inventory to each element of a sequence. They are a part of many natural language processing pipelines. For instance, part-of-speech taggers assign labels such as "noun" or "transitive verb" to tokens from a natural language corpus.

Tags are often used as features by downstream natural language processing components. For example, feeding tags into a parser allows it to abstract over concrete tokens and take the more abstract part-of-speech tags into account when making decisions. This abstraction improves the parser's ability to generalize.

In lexicalised grammar formalisms such as CCG (Clark and Curran 2004) or lexicalised tree-adjoining grammar (Bangalore and Joshi 1999), tagging is employed to improve efficiency. In such grammars, almost all decisions are shifted to the lexicon, making the choice of lexicon entries important but also challenging, since lexical categories are complex and there are many categories to choose from. Since parsing algorithms such as CKY (see Section 3.1.6) have a high polynomial complexity, this results in severe computational overhead. Sequence taggers can be used to reduce the search space at the lexical level by pre-selecting the lexical categories made available to the parser. Such taggers are called *supertaggers*.

Concretely, a supertagger for CCG is trained on a corpus such as CCGbank (Hockenmaier and Steedman 2007) to predict the syntactic category associated with each token. Its tag set is therefore the set of syntactic categories that were observed in the corpus, including both atomic categories such as NP and complex

Algorithm 3.6 Inside-outside algorithm calculating expected PCFG rule counts.

Inputs:
- A CFG (N, Σ, R, S)
- A sentence $x = (t_1 \dots t_n)$
- A probability distribution $q : R \to [0, 1]$

Output: A mapping $\mathit{count} : R \to \mathbb{R}^+$

function COUNTS($x = (t_1, \dots, t_n), q$)

for $i \leftarrow 1 \dots n$ **do** ▷ Initialise inside scores for terminal rules

$$\mathit{inside}(A, i, i) \leftarrow \begin{cases} q(A \to t_i) & \text{if } A \to t_i \in R \\ 0 & \text{otherwise} \end{cases}$$

end for

for $j \leftarrow 1 \dots n, i \leftarrow j \dots 1, A \in N$ **do** ▷ Compute inside scores

$$\mathit{inside}(A, i, j) \leftarrow \sum_{A \to BC \in R} \sum_{k=i}^{i-1} q(A \to BC)\, \mathit{inside}(i, k)\, \mathit{inside}(k+1, j)$$

end for

for all $A \in N$ **do** ▷ Initialise outside scores for root of derivation

$$\mathit{outside}(A, 1, n) = \begin{cases} 1 & \text{if } A = S \\ 0 & \text{otherwise} \end{cases}$$

end for

for $j \leftarrow n \dots 1, i \leftarrow 1 \dots j, A \in N$ **do** ▷ Compute outside scores

$$\mathit{outside}(A, i, j) \leftarrow \sum_{B \to CA \in R} \sum_{k=1}^{i-1} q(B \to CA)\, \mathit{inside}(C, k, i-1)\, \mathit{outside}(B, k, j) + \sum_{B \to AC \in R} \sum_{k=1}^{i-1} q(B \to AC)\, \mathit{inside}(C, j+1, k)\, \mathit{outside}(B, i, k)$$

end for

for all $A \to BC \in R$ **do** ▷ Compute counts for non-terminal rules

$$\mathit{count}(A \to BC) \leftarrow \sum_{1 \le i < k < j < n} \frac{\mathit{outside}(A,i,j)\, q(A \to BC)\, \mathit{inside}(B,i,k)\, \mathit{inside}(C,k+1,j)}{\mathit{inside}(\mathrm{S},1,n)}$$

end for

for all $A \to x \in R$ **do** ▷ Compute counts for terminal rules

$$\mathit{count}(A \to t) \leftarrow \sum_{i : t_i = t} \frac{\mathit{inside}(A,i,i)\, \mathit{outside}(A,i,i)}{\mathit{inside}(\mathrm{S},1,n)}$$

end for

return *count*

end function

categories such as (S\NP)/NP. For each token, it outputs a probability distribution over all tags, the top items of which are forwarded to the parser. Instead of choosing from the whole space of syntactic categories, the parser can therefore focus on a limited number of tags per token, greatly improving its ability to search for complete derivations.

A supertag sequence already contains a large amount of information about a CCG derivation. This is demonstrated by the fact that the EasyCCG parser is able to use a purely deterministic parsing component while achieving respectable accuracy (Lewis and Steedman 2014), relying on a supertagger as the only learned component.

Semantic parsers also face the problem of lexical selection, albeit in an even more severe form: they need to choose not just a syntactic category, but also the semantic content of any given token. For composition-based semantic parsers, supertagging is therefore an attractive option to limit the parser's search space (Groschwitz, Fowlie et al. 2017).

In recent research, Bidirectional LSTM (BiLSTM) models are frequently used for supertagging (Lewis, K. Lee and L. Zettlemoyer 2016; Vaswani et al. 2016). The supertagger introduced in Section 6.3 also follows this approach. In the rest of this section, we therefore discuss the technical foundations for BiLSTM supertagging.

3.5.1. Long Short-Term Memory

The Long Short-Term Memory (LSTM) is a type of neural network commonly used for sequence processing (Hochreiter and Schmidhuber 1997). It is recurrent, meaning that information is transferred not just from input to output, but also along the *time dimension*, for instance, between the tokens of a sentence. The state of the LSTM when processing a token depends not just on the token itself, but also on the previous LSTM state, which allows the network to "remember" contextual information.

For every input element x_j, an LSTM computes two quantities: a memory vector c_j and a hidden state h_j.[4] Both are vectors of length d, where the dimension d is a hyperparameter which can be chosen when implementing an LSTM network. The hidden state h_j is also considered the *output* of the LSTM at time step j.

The distinctive feature of LSTMs are its three gates: the forget gate f, input gate i, and output gate o. All three depend on the current sequence element x_j and the previous hidden state h_{j-1} to compute a vector of weights to apply to the elements

[4]See Goldberg (2017) for a more detailed introduction. In this section, we adopt their notation.

of an LSTM state vector:

- The forget gate is applied to the previous cell's memory vector c_{j-1}, which allows the LSTM to bring some components close to zero in preparation for overwriting them.
- The input gate is applied to an update candidate z to control which of its components are added to the memory vector.
- The output gate is used to control which parts of the memory vector c_j are represented in the hidden state h_j.

Each gate is represented by a pair of weight matrices which can be trained to optimise the LSTM's operation.

The full calculations performed by an LSTM are as follows:

$$c_j = f \odot c_{j-1} + i \odot z \tag{3.8}$$
$$h_j = o \odot \tanh(c_j) \tag{3.9}$$

where

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi}) \tag{3.10}$$
$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf}) \tag{3.11}$$
$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho}) \tag{3.12}$$
$$z = \tanh(x_j W^{xz} + h_{j-1} W^{hz}) \tag{3.13}$$

In this description, $\odot$ is the element-wise product, $x_j \in \mathbb{R}^{d_x}$ and $c_j, h_j, i, f, o, z \in \mathbb{R}^d$ are vectors, and $W^{xi}, W^{xf}, W^{xo}, W^{xz} \in \mathbb{R}^{d_x \times d}$ as well as $W^{hi}, W^{hf}, W^{ho}, W^{hz} \in \mathbb{R}^{d \times d}$ are weight matrices.

3.5.2. Stacked and Bidirectional LSTM

On a high level, an LSTM transforms an input sequence $x_1, \ldots, x_n$ into an output sequence $h_1, \ldots, h_n$. It is therefore possible to stack another LSTM on top of the first one using $h_1, \ldots, h_n$ as the input sequence. This adds additional parameters and computing steps to the model, enabling it to deal with more complex inputs.

In natural language processing, it is also common to encounter phenomena where the output generated for a token depends on its right context. LSTMs are therefore

commonly used bidirectionally: two LSTMs are instantiated, where the first LSTM processes the sentence in left-to-right order and the second LSTM in reverse. The outputs of both LSTMs are concatenated for each token to form the output of the bidirectional LSTM.

3.5.3. Using an LSTM to Predict Tags

A stacked BiLSTM outputs a sequence of vectors $h_1, \ldots, h_n$ with of dimensionality $2d$. To use these vectors for the prediction of tags, they must be mapped onto the tag inventory. This is achieved by passing the output vectors through an additional fully connected neural network layer, followed by a softmax activation function which maps the layer's output into the output range $[0, 1]$. The resulting vector assigns a score to every possible tag, and can be decoded by choosing the single highest-scored tag. More complex decoding strategies can also be applied, such as emitting all tags above a certain score threshold.

A fully connected layer is composed of a weight matrix $W \in \mathbb{R}^{d \times k}$ and a bias term $b \in \mathbb{R}^k$, where k is the output dimensionality, that is, the number of tags. The output of the tagger is computed as follows:

$$o_j = \text{softmax}(h_j W + b)$$

The softmax function applies the exponential function to every element of the output vector and then normalises the vector so that the elements sum to one.

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Here, z_i and z_j represent individual entries of the k-dimensional vector z.

3.5.4. Word Vectors

Whereas the input of a tagger is a sequence of discrete tokens, the computations of neural networks are performed upon continuous vectors. Furthermore, since the size of an LSTM's weight matrices scales with the dimensionality of the input, it is not feasible to represent the input using one-hot vectors which are zero except for the single position indicating the input word, as these vectors can have millions of entries.

Instead, words are represented as dense vectors of relatively low dimension. The simplest implementation is to keep a lookup table (also called an embedding matrix)

which contains a word vector for every word in the vocabulary. Rare words are represented using a special entry often called `UNK` (for *unknown*). Word embeddings can be obtained by training an embedding matrix on the fly, using a precomputed embedding, or applying a pre-trained language model to the input:

- To train word embeddings on the fly, the embedding matrix is initialised randomly and then updated along with the other layers during training. This allows the word embeddings to represent properties that are relevant to the training task.
- Pre-trained word embeddings are derived from large corpora using generic training tasks, such as predicting words from their context (Mikolov et al. 2013), or predicting the co-occurrence probabilities of words (Pennington, Socher and Manning 2014).

  After a word embedding model has been trained, its embedding vectors are saved to a table and can be re-used to encode the input for a tagger. In this application scenario, each input word is replaced by the word embedding vector taken from the table. This implies that while the words' context is considered during training, the resulting word embeddings are applied independent of context and conflate various forms and senses of words that share the same surface form.
- Large pre-trained language models are sequence-processing models that, like word embedding models, are trained on large, broad-coverage corpora. The difference lies not just in the models' size, but also in how they are applied: The tagger's input sentence is not encoded word-by-word, but the sentence is passed to the model as a whole, which allows the model to encode the words' context-dependent semantics in its word representations. Word representations from large pre-trained language models have been shown to improve performance on a range of natural language processing tasks (Devlin et al. 2019; Howard and Ruder 2018; Peters et al. 2018).

  Pre-trained language models can also be used in a transfer learning setting by embedding them within another model and thus having them participate in the training for the specific task.

In practice, more than one embedding method can be used. For instance, the input for the tagging model can be constructed by concatenating an embedding trained on the fly with a pre-trained embedding. In Section 6.3, we describe our supertagger for GA-CCG and experiment with various sources of embeddings.

Chapter 4.

Graph Algebraic Combinatory Categorial Grammar

In this chapter, we introduce Graph Algebraic Combinatory Categorial Grammar (GA-CCG), a grammar which describes how AMRs can be derived from natural language sentences. GA-CCG is the theoretic basis for the implementation of a GA-CCG semantic parsing pipeline that is described in the following chapters.

This chapter is divided into two main sections. In Section 4.1, the the semantic construction operations used in GA-CCG (Section 4.1) are defined. Section 4.2 describes how they can be integrated with CCG to achieve form a syntactic-semantic grammar.

4.1. Semantic Construction of AMRs

Semantic construction is the process of composing semantic fragments into a full meaning representation. In Section 3.2, the HR algebra has been introduced as a mechanism for the composition of graphs. However, the HR algebra is very general and allows the construction of all kinds of graphs, whereas we deal only with AMR graphs. We will therefore define a restricted set of operations which better captures the composition of AMRs. At the same time, there is no need to define our operations on all possible s-graphs. We therefore define GA-CCG s*-graphs, which are the objects of our algebra and embody specific rules for the assignment of source labels.

4.1.1. Placeholder AMRs

AMRs are directed graphs. To recapitulate Definition 2.1, an AMR contains two types of vertices:

- Constant nodes, which are labelled with a concept.
- Variable nodes, which are unlabelled.

In addition, AMR edges are labelled with a role.

In a complete AMR, every variable node is required to have an outgoing instance edge, which assigns a concept to the event or entity represented by the variable node. In a *placeholder AMR*, this requirement is lifted: variable nodes may be defined without specifying a concept they instantiate. We call such nodes *placeholders*.

Definition 4.1 (Placeholder AMR)**.** Let $G = (V, E, v_{root}, l)$ be a semantic graph[1] with variable nodes V_v and constant nodes V_c.

G is called a *placeholder AMR* iff. all of the following hold:

- $l(v) \in L_{concepts} \cup L_{constants}$ for $v \in V_c$ (all constant nodes are labelled with concepts)
- $l(e) \in L_{relations}$ for $e \in E$ (all edges are labelled with relations)

In contrast to the definition of AMRs (Definition 2.2), it is not required that variable nodes have outgoing instance edges. Nodes that are not edges are called placeholders. $V_p = \{v \in V_v : \neg\exists e = (v, v') \in E : l(e) = \mathsf{instance}\}$ is the set of *placeholders* of G.

The motivation for introducing placeholders is that we wish to represent fragments of AMRs which can be composed into a complete AMR. Placeholders can be filled by merging them with nodes from another graph.

4.1.2. GA-CCG s*-Graphs

It is straightforward to extend placeholder AMRs with source labels by defining a function *slab* as introduced in Section 3.2. Interpreting placeholder AMRs as s*-graphs will allow us to define graph-algebraic operations over meaning representations. These GA-CCG s*-graphs contain the information necessary to combine elementary meaning fragments to form full AMRs. In this work, they are used to express the meanings of lexical entries, phrases, and full sentences.

Source labels used in GA-CCG s*-graphs take one of the following forms:

- $\langle$root$\rangle$ marks a node as the *root* of the graph.

[1] See Definition 2.1.

- $\langle i \rangle$ with $i \in \mathbb{N}$ is used to label placeholders with an index i.
- $\langle \mathsf{s} \rangle$ is used temporarily to label a placeholder-argument pair.

Root source In every non-empty AS s*-graph, exactly one node is marked as the root of the graph. The significance of this label is twofold. First, as in AMR, it serves as an indication of the semantic focus of the expressed meaning. Second, it defines the node to which semantic operations, such as modification, are usually applied.

Placeholder sources Every placeholder must be labelled as an $\langle i \rangle$-source. The indices i that occur in a GA-CCG s*-graph are required to be unique and consecutive starting from 0. Therefore, there exists a total ordering among the placeholders, and we call the one with the highest index the *outermost placeholder* p_{max}.

By indexing placeholders, we allow GA-CCG s*-graphs to play the role of functions: each placeholder represents an argument position. A placeholder may be filled by merging it with a non-placeholder node. The outermost placeholder represents the function's first argument and is filled first. While numbering placeholders from the inside out may seem unusual, it provides the convenience of avoiding renumbering placeholders after every operation.

Definition 4.2 (GA-CCG s*-graph)**.** Let the set of GA-CCG source labels $\mathcal{S}$ be defined as follows: $\mathcal{S} = \{\langle \mathsf{root} \rangle, \langle \mathsf{s} \rangle\} \cup \{\langle i \rangle : i \in \mathbb{N}\}$.

Furthermore, let $G = (V, E, v_{root}, l)$ be a placeholder AMR with placeholders $V_p \subseteq V$, and $slab : V \to \mathcal{P}(\mathcal{S})$ a source labelling function.

A tuple $(V, E, v_{root}, l, slab)$ is called a *GA-CCG s*-graph* iff. the following hold:

1. $\langle \mathsf{root} \rangle \in slab(v_{root})$ (the root of the graph is labelled as a root source)
2. $|\{v : v \in V, \langle \mathsf{s} \rangle \in slab(v)\}| = 0$ (there is no s-source[2])
3. $[\exists v \in V : \langle i \rangle \in slab(v)] \Rightarrow [\exists v' \in V : \langle i-1 \rangle \in slab(v')]$ for all $i \geq 1$ (i-sources are numbered consecutively starting from 0)
4. $\langle i \rangle \in slab(v) \Leftrightarrow v \in V_p$ for all $i \in \mathbb{N}$ (the i-sources are exactly the placeholders)

The set of all GA-CCG s*-graphs is denoted $\mathcal{G}^*$.

Definition 4.3 (Outermost Placeholder)**.** Let G be a GA-CCG s*-graph with placeholders V_p and $|V_p| \geq 1$. The *outermost placeholder index* $p_{max}(G)$ is the highest index of any i-source occurring in the graph:

$p_{max}(G) = \max\{i : i \in \mathbb{N}, \exists v \in V_p : \langle i \rangle \in slab(v)\}$

[2] $\langle \mathsf{s} \rangle$ is assigned as a source label only temporarily during the processing of a semantic operator.

We now define a number of operators that combine pairs of GA-CCG s*-graphs.

4.1.3. The Apply Operator

The *apply operator* **A** is used to model basic function application, such as the filling of a verb's argument slots. Its first argument must therefore have at least one placeholder. **A** *fills* the outermost placeholder and *merges* the remaining placeholders of the function graph. Both of these aspects are implemented using the more fundamental parallel composition operator $/\!\!/$, as defined in Section 3.2.3.

Definition 4.4 (apply operator)**.** Let G, H be GA-CCG s*-graphs where G has the set of placeholders V_p^G and $|V_p^G| \geq 1$. The *apply operator* **A** is defined as follows:

$$\mathbf{A}(G, H) = G' /\!\!/ H'$$

where

$$G' = ren_{\{\langle p_{max}(G)\rangle \to \langle \mathsf{s}\rangle\}}(G)$$
$$H' = ren_{\{\langle \mathsf{root}\rangle \to \langle \mathsf{s}\rangle\}}(H)$$

Example 4.5. Consider the sentence *I slept*. Its meaning can be represented by the following AMR graph:

$$\underline{\text{sleep-01}} \xrightarrow{\text{ARG0}} \text{i} \qquad = G_{I\ slept}$$

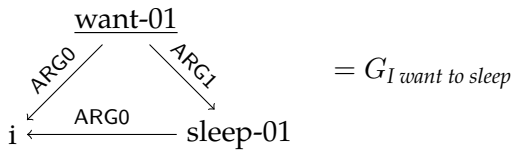
Suppose that the lexicon assigns the following meaning representations to the two individual words:

$$I\text{:} \quad \underline{\text{i}} \qquad = G_I$$
$$slept\text{:} \quad \underline{\text{sleep-01}} \xrightarrow{\text{ARG0}} \langle 0 \rangle \qquad = G_{slept}$$

The **A** operator can be used to derive the sentence meaning from the lexical meanings:

$$G_{I\ slept} = \mathbf{A}(G_{slept}, G_I)$$

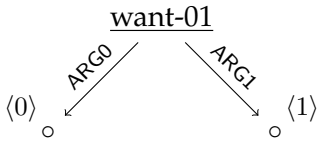
Example 4.6. This example illustrates how sources are merged by the **A** operator. Consider the sentence *I want to sleep*, represented by the following AMR:



$= G_{I\ want\ to\ sleep}$

In addition to the lexical entries from Example 4.5, assume that *want* is represented in the lexicon as follows:

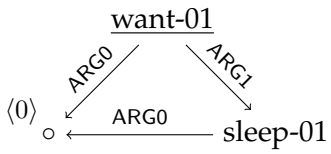
want:



$= G_{want}$

To construct the meaning of the phrase *want to sleep*, $\mathbf{A}(G_{want}, G_{sleep})$ may be evaluated[3]. Since G_{want} and G_{sleep} both contain a placeholder labelled $\langle 0 \rangle$, both placeholders are merged, creating the triangular control structure that can be observed in the sentence meaning representation.

$\boldsymbol{A}(G_{want}, G_{sleep}) =$



The sentence meaning *I want to sleep* is therefore represented by the following expression:

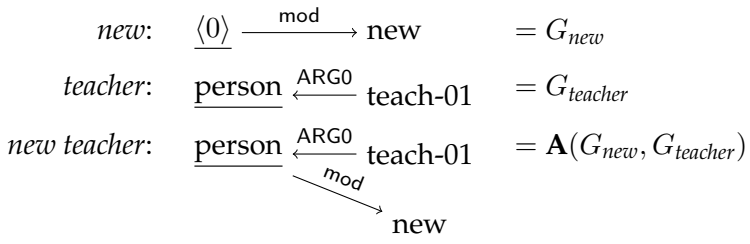
$$G_{I\ want\ to\ sleep} = \boldsymbol{A}(\boldsymbol{A}(G_{want}, G_{sleep}), G_I)$$

Additional examples for the use of $\boldsymbol{A}$ and other semantic operators are provided in Section 4.2 in the context of GA-CCG derivations for these sentences.

[3] For simplicity, the semantics of *to* are ignored here. A more syntactically complete account of this sentence is given in Section 4.2.

4.1.4. The Modify Operator

Apart from the filling of verb argument slots, application also models modification. Modifiers act as functions and are represented by graphs whose only placeholder is the root. For example, a representation for the phrase *new teacher* can be constructed using the **A** operator as follows:



As the example shows, the *apply* operator is suitable for modifications where the entity being modified is represented by the root of the graph.

In this example, the semantic representation for *teacher* contains two nodes, where one node represents an (abstract) event of teaching and the other node the person who teaches. This interpretation results from the AMR rule stating that OntoNotes frames are used to represent noun semantics whenever possible[4].

A decompositional interpretation of nouns such as *teacher* enhances the expressiveness of AMR, as both the *person* and *teaching* aspects of *teacher* can be targeted by modifiers. However, this behaviour is not supported by the *apply* operator, which always targets the root node. We therefore define a *modify* operator which allows the targeting of nodes other than the root of the argument.

Definition 4.7 (modify operator)**.** Let G, H be GA-CCG s*-graphs where G has the set of placeholders V_p^G and $|V_p^G| \geq 1$.

For all $i \in \mathbb{N}$ such that $|\mathit{level}_i(H)| = 1$, the *modify operator* $\mathbf{M}_i$ is defined as follows:

$$\mathbf{M}_i(G, H) = G' /\!\!/ H'$$

where

$$G' = \mathit{ren}_{\{\langle p_{max}(G)\rangle \to \langle \mathsf{s}\rangle\}}(G)$$
$$H' = \mathit{add}_{\{v \to \langle \mathsf{s}\rangle : v \in \mathit{level}_i(H)\}}(H)$$

[4] See Banarescu et al. (2019), Part III, Nouns that invoke predicates.

Example 4.8. In practice, modification at level 1 is sufficient for handling nouns such as *teacher*. Such a modification occurs in the phrase *school teacher*, as shown below.

$$\begin{aligned}
\textit{school}: &\quad \underline{\langle 0 \rangle} \xrightarrow{\text{location}} \text{school} && = G_{\textit{school}} \\
\textit{teacher}: &\quad \underline{\text{person}} \xleftarrow{\text{ARG0}} \text{teach-01} && = G_{\textit{teacher}} \\
\textit{school teacher}: &\quad \underline{\text{person}} \xleftarrow{\text{ARG0}} \text{teach-01} \xrightarrow{\text{location}} \text{school} && = \mathbf{M}_1(G_{\textit{school}}, G_{\textit{teacher}})
\end{aligned}$$

4.1.5. The Compose Operator

Sometimes, the need arises to construct meaning representations for unconventional constituents. In sentences such as *I read the paper the scientist wrote*[5], the phrase *the scientist wrote* forms a constituent, although it is missing an object.

Usually, the representations for transitive verbs are constructed such that the object is represented with the outer placeholder, following the convention that verb phrases (that is, sentences missing a subject) are considered constituents and thus the verb is applied first to the object and then to the subject. In the case of extracted objects, the argument slots need to be filled in the opposite order.

This situation is modelled using the *compose operator* **B**, which is defined as follows:

Definition 4.9 (compose operator)**.** Let G, H be GA-CCG s*-graphs where G has the set of placeholders V_p^G with $|V_p^G| \geq 2$, and H has the set of placeholders V_p^H $|V_p^H| = 0$. The *compose operator* $\mathbf{B} : \mathcal{G}^* \times \mathcal{G}^* \to \mathcal{G}^*$ is defined as follows:

$$\mathbf{B}(G, H) = G' /\!\!/ H'$$

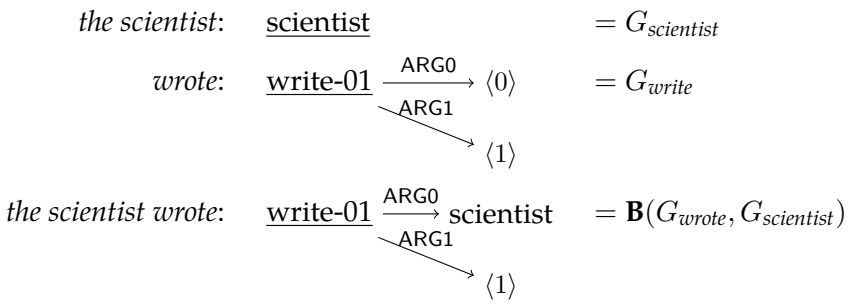
where

$$\begin{aligned}
G' &= \mathit{ren}_{\{\langle p_{\mathit{max}}(G)-1\rangle \to \langle \mathsf{s}\rangle\}, \langle p_{\mathit{max}}(G)\rangle \to \langle p_{\mathit{max}}(G)-1\rangle}(G) \\
H' &= \mathit{ren}_{\{\langle \mathsf{root}\rangle \to \langle \mathsf{s}\rangle\}}(H)
\end{aligned}$$

To avoid clashes between placeholders, the compose operator is only defined if the argument has no placeholders. Also, the function graph needs to have at least two placeholders for the second-to-outermost placeholder to be filled.

[5] See Figure 4.6 for a CCG analysis of this example.

Example 4.10. The compose operator allows the constituent *the scientist wrote* to be constructed as follows:



4.1.6. The Substitute Operator

The *substitute operator* **S** is similar to **A** in that it fills the outermost placeholder of its first argument. However, it first renames its first argument's remaining placeholders so as not to collide with the second argument's placeholders: it shifts the first argument's placeholders outward. That is, while **A** *merges* the two graphs' placeholders, **S** keeps them separate.

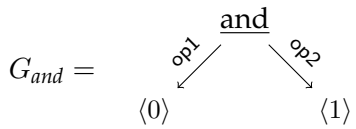
Definition 4.11 (substitute operator)**.** Let G, H be GA-CCG s*-graphs where G has the set of placeholders V_p^G and $|V_p^G| \geq 1$. The *substitute operator* **S** is a function defined as follows:

$$\mathbf{S}(G, H) = G' /\!\!/ H'$$

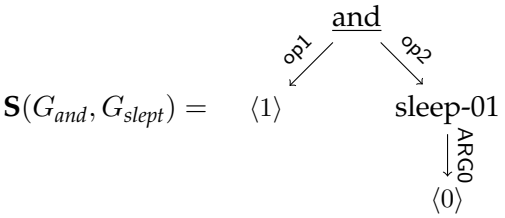
where

$$\begin{aligned}
G' &= ren_{f_1}(G) \\
H' &= ren_{f_2}(H) \\
f_1(s) &= \begin{cases} \langle \mathsf{s} \rangle & \text{if } s = \langle p_{\mathsf{max}}(G) \rangle \\ \langle i + p_{\mathsf{max}}(H) \rangle & \text{if } s = \langle i \rangle, i < p_{\mathsf{max}}(G) \\ s & \text{otherwise} \end{cases} \\
f_2(s) &= \begin{cases} \langle \mathsf{s} \rangle & \text{if } s = \langle \mathsf{root} \rangle \\ s & \text{otherwise} \end{cases}
\end{aligned}$$

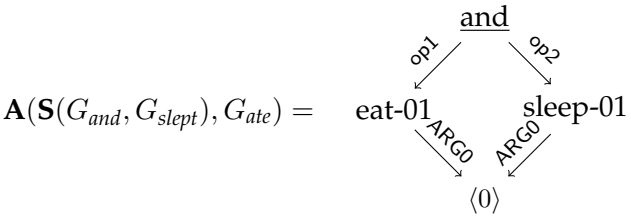
Example 4.12. An important use case for the **S** operator is the modelling of conjunctions. Assume that the conjunction *and* is represented in the lexicon as follows:



In the sentence *I ate and slept*, two intransitive verbs are conjoined. The substitution operator is necessary to prevent the merging of the conjunction's argument slot with the verb's argument slot.



The op1 placeholder is renamed to $\langle 1 \rangle$, which allows both verbs' argument slots to merge in a further application step.



4.1.7. The Ignore Operator

The semantic operator **Ki** does not perform any graph operation at all, but always returns its second argument unchanged. It is used for dealing with punctuation, which is assumed here to have no influence on a sentence's semantic representation.[6]

[6]The name **Ki** originates in combinatory logic, where KI represents the combinator that always returns its second argument.

Definition 4.13 (ignore operator)**.** Let G, H be s*-graphs. The *ignore operator* **Ki** is defined as follows:

$$\mathbf{Ki}(G, H) = H$$

4.2. Graph Algebraic Combinatory Categorial Grammar

In the previous section, the semantic operators **A**, $\mathbf{M}_1$, **B**, **S**, and **Ki** were introduced. Together, they form an algebra that describes the construction of abstract meaning representations. However, so far this algebra is not connected to the surface form of natural language sentences as we have not yet defined a syntax-semantic interface.

In this section, we adapt the syntax-semantics interface of CCG to make use of graph-algebraic semantic operators. By mapping CCG combinators to semantic operators, we obtain a family of fully syntactic-semantic grammars which we collectively call Graph Algebraic Combinatory Categorial Grammar (GA-CCG).

In this section, we describe the inventory of rules for GA-CCG and present the concrete GA-CCG rule sets used for evaluations in the remainder of this thesis. Following the presentation of each rule, we provide motivating examples from the AMR corpus.

4.2.1. Definition of GA-CCG

Traditional syntactic-semantic CCG derivations employ combinators that have a defined effect on both syntactic and semantic categories (see Section 3.1 for details). By first selecting lexical entries for each word of a sentence and then recursively applying unary and binary combinators, the tree structure of a CCG derivation is created, each node of which is annotated with both types of categories.

To adapt this framework to the graph algebraic construction of AMRs, we must define the effect of each derivation step in terms of GA-CCG s*-graphs.

Whereas in CCG, the applicability of a combinator depends solely on the syntactic categories of its argument nodes and the computation represented by the combinator, we choose to equip GA-CCG rules with additional predicates over the syntactic context of the derivation node to allow fine-grained control over the context within which a combinator applies. Specifically, this syntactic context consists of the following elements

1. the CCG combinator applied at the step,

2. the syntactic categories of the step's child nodes.

Furthermore, each rule is associated with a semantic operator. If the rule's predicate holds, the semantic operator is applicable at the given step.

Definition 4.14 (GA-CCG)**.** Let c^l_{syn} and c^r_{syn} be syntactic categories. The tuple $c = (c^l_{\text{syn}}, c^r_{\text{syn}})$ is called a *binary rule context*.

Let C be a binary CCG combinator, p be a predicate over rule contexts and o a semantic operator. Then the tuple (C, p, o) is called a *binary GA-CCG rule*.

A binary rule $r = (C, p, o)$ is applicable to a binary rule context $(c^l_{\text{syn}}, c^r_{\text{syn}})$ if $p(c^l_{\text{syn}}, c^r_{\text{syn}})$ holds and C is applicable to c^l_{syn} and c^r_{syn}. In this case, we write IsApplicable$(r, c^l_{\text{syn}}, c^r_{\text{syn}})$.

Let R_2 be a set of binary GA-CCG rules and R_1 be a set of unary CCG combinators. The pair (R_1, R_2) is called a *GA-CCG rule set*.

For brevity, we will express the predicates of GA-CCG rules in the form of a simple pattern matching language. Each of the elements C, c^l_{syn}, and c^r_{syn} is matched according to one of the following expressions:

- $*$: matches anything
- **X** (for a combinator **X**): matches exactly this combinator
- X (for a syntactic category X): matches exactly this syntactic category
- $\sim X$ (for a syntactic category X): matches anything but this syntactic category
- α: matches any syntactic category, but requires that all instances of the symbol refer to the same category

A rule is applicable if all three patterns for combinator, left and right syntactic category match.

4.2.2. Directionality of Operators

In CCG, combinators are directional: Some can be applied in forward or backward direction, others are only permitted in one direction. This directionality needs to be reflected on the semantic level. This is achieved by simply reversing the order of the semantic operator's arguments.

Definition 4.15 (Backward Semantic Operators)**.** Let $f \in \{\mathbf{A}, \mathbf{M}_1, \mathbf{B}, \mathbf{S}, \mathbf{Ki}\}$ be a semantic operator. We define the backward version of f as follows:

$$f^{\leftarrow}(G, H) = f(H, G)$$

4.2.3. Unary Rules

The grammars defined in the preceding section neglect to define a semantic interpretation for unary rules such as type raising and type changing rules. Particularly type raising, which adds argument slots to a given semantic representation, is a challenge in the GA-CCG framework: since edge labels are defined as part of the GA-CCG s*-graph, proper relations would have to be invented during type raising. One possible solution is to leave edge labels underspecified until later in the derivation process, which has been explored by Blodgett and Schneider (2019). Within the scope of this dissertation, we leave this gap unfilled and define all unary rules to return the semantic category unchanged.

4.2.4. GA-CCG Rule Sets

We are now ready to compile sets of GA-CCG rules. Since we employ the EasyCCG parser (Lewis and Steedman 2014) for syntactic parsing during the lexicon induction step, the grammars must handle the set of combinators output by the EasyCCG parser.

We define one full grammar which employs all semantic operators, named **all**. This grammar contains specialized rules for specific syntactic contexts, namely type raising, noun modification, and syntactic expletives. To test the effect of these rules, we also define the three simplified grammars **no-fa-modify**, **no-ba-ignore**, and **no-tr**, in which the respective specialised rules are omitted. All of these omissions are combined in the **base** grammar, which can be considered a baseline consisting only of the most broadly applicable rules. Table 4.1 lists the rules for each of the grammars.

The following is an overview of the basic rules of GA-CCG (examples will be given in the rest of this section):

- All application and composition combinators are associated with the apply operator, observing directionality.
- The **conj** combinator is associated with both the substitute and the apply operator, as not all syntactic conjunctions are represented as conjunction nodes in AMR.
- Punctuation combinators including **tc-rel** are associated with the apply operator, observing directionality.

The following additional rules cover special syntactic constructions:

- Modifiers and determiners may attach to a level-1 node with the modify operator.
- Composition is interpreted by the backward compose operator if the left constituent is a type-raised noun phrase.
- In backward application and backward crossed composition, the direction of the apply operator is reversed if the right constituent is a type-raised verb phrase.
- In application, semantically empty arguments such as the expletive *it* may be consumed by the ignore operator.

These rules form a fine-grained and thus restrictive grammar. Hypothetically, a parser might benefit from being able to apply semantic operators more freely. To test this hypothesis, we also define the relaxed grammars **unrestricted-a** and **unrestricted-all**. Respectively, they allow the apply operator, or all semantic operators, to be applied at each derivation step in any direction. These grammars are defined in Table 4.2.

4.2.5. Application

Application is the most common combinator in CCG derivations, and used in a variety of situations. In GA-CCG, the semantics of this combinator are modeled using the **A** operator with the direction of the operator following that of the combinator.

CCG uses the specialised **rp** and **lp** combinators to deal with punctuation. They absorb punctuation to the right or left of a word, respectively. In the context of GA-CCG, punctuation can be interpreted as a special case of application: since punctuation usually does not contribute any semantic content, its lexical meaning is the identity function, that is, the graph consisting of a single placeholder. Since we can assume this meaning to be fixed, punctuation combinators can be interpreted using the **Ki** operator, which reduces the amount of superfluous lexical items produced during lexicon induction.

Not all punctuation is semantically empty: in particular, commas can play the role of conjunctions and are assigned the syntactic category *conj*. This case is demonstrated in Example 4.24

The following example contains both simple cases of application and punctuation. For brevity, sentence-ending punctuation is omitted in all remaining examples.

Context			Interpretation				
C	c^l_{syn}	c^r_{syn}	**all**	**no-modify**	**no-ignore**	**no-tr**	**base**
$>$	$*$	$*$	**A**	**A**	**A**	**A**	**A**
	α/α	$*$	$\mathbf{M}_1$	–	$\mathbf{M}_1$	$\mathbf{M}_1$	–
	NP/N	$*$	$\mathbf{M}_1$	–	$\mathbf{M}_1$	$\mathbf{M}_1$	–
	$*$	NP	$\mathbf{Ki}^{\leftarrow}$	$\mathbf{Ki}^{\leftarrow}$	–	$\mathbf{Ki}^{\leftarrow}$	–
$<$	$*$	$\sim X_1$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$
	$*$	X_1	**A**	**A**	**A**	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$
	$*$	$\alpha\backslash\alpha$	$\mathbf{M}^{\leftarrow}{}_1$	–	$\mathbf{M}^{\leftarrow}{}_1$	$\mathbf{M}^{\leftarrow}{}_1$	–
	NP	$*$	**Ki**	**Ki**	–	**Ki**	–
$>\mathbf{B}$	$\sim X_2$	$*$	**A**	**A**	**A**	**A**	**A**
	X_2	$*$	$\mathbf{B}^{\leftarrow}$	$\mathbf{B}^{\leftarrow}$	$\mathbf{B}^{\leftarrow}$	**A**	**A**
$<\mathbf{B}_\times$	$*$	$\sim X_1$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$
	$*$	X_1	**A**	**A**	**A**	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$
$>\mathbf{B}^2$	$*$	$*$	**A**	**A**	**A**	**A**	**A**
$<\mathbf{B}^2_\times$	$*$	$*$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$
conj	$*$	$*$	**S**	**S**	**S**	**S**	**S**
	$*$	$*$	**A**	**A**	**A**	**A**	**A**
lp	$*$	$*$	**A**	**A**	**A**	**A**	**A**
rp	$*$	$*$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$	$\mathbf{A}^{\leftarrow}$
tc-rel	$*$	$*$	**A**	**A**	**A**	**A**	**A**

Table 4.1.: Rules for the main grammars examined in this thesis. The abbreviated syntactic categories X_1 and X_2 are defined as follows: $X_1 = (S\backslash NP)/((S\backslash NP)/NP)$; $X_2 = S/(S\backslash NP)$.

Context			Interpretation	
CMB	SYN_l	SYN_r	**unrestricted-a**	**unrestricted-all**
$*$	$*$	$*$	**A**, $\mathbf{A}^{\leftarrow}$	**A**, $\mathbf{A}^{\leftarrow}$, **B**, $\mathbf{B}^{\leftarrow}$, $\mathbf{M}_1$, $\mathbf{M}^{\leftarrow}{}_1$, **S**, $\mathbf{S}^{\leftarrow}$, **Ki**, $\mathbf{Ki}^{\leftarrow}$

Table 4.2.: Rules for the two unrestricted grammars examined in this thesis.

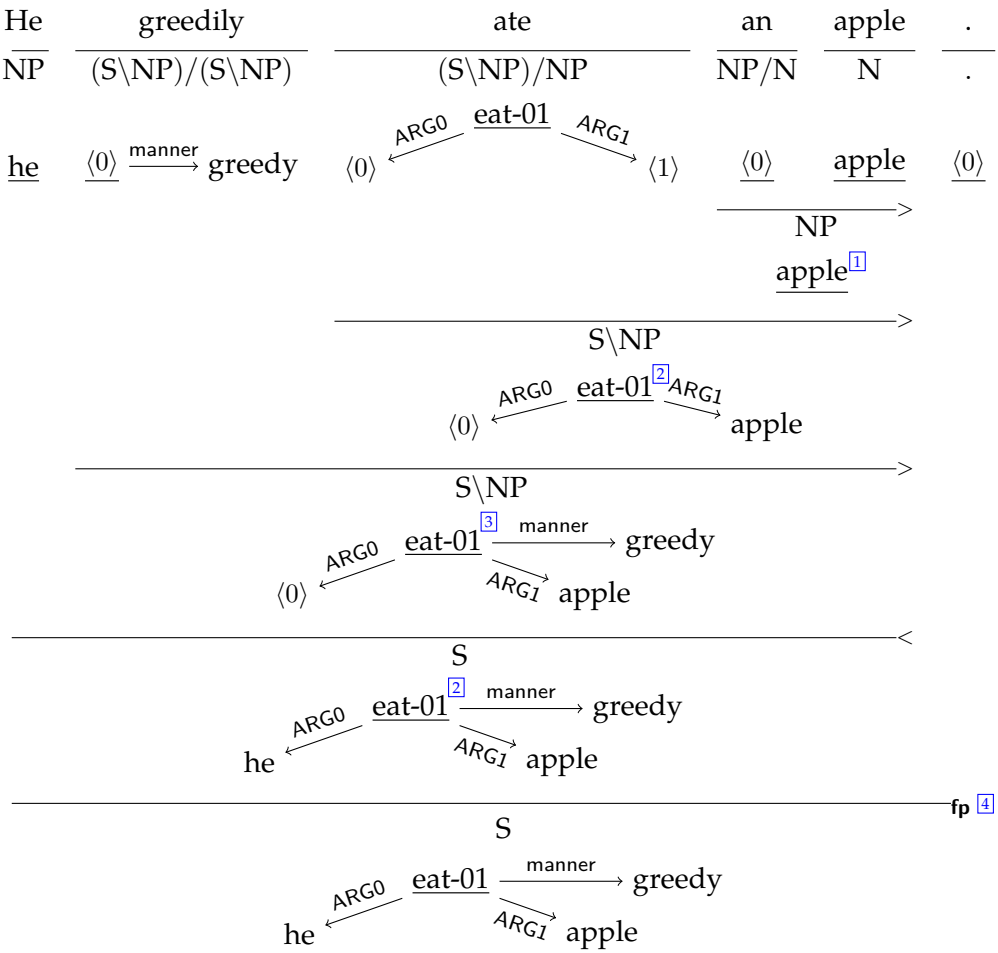


Figure 4.1.: A GA-CCG derivation for the sentence *He greedily ate an apple*. The use of application is demonstrated in three common contexts: application of a (semantically empty) determiner [1], the filling of a verb's argument slots [2], and modification by an adverb [3]. The example also demonstrates how punctuation is absorbed with the *forward punctuation* combinator and either the **Ki** or the **A** operator [4].

Example 4.16 (Application in GA-CCG)**.** Some of the most common uses of application in CCG are the filling of argument slots, modification, and the application of determiners. These three contexts are illustrated in the example given in Figure 4.1.

Punctuation can be viewed as a special case of application, where punctuation is always assigned identity semantics. However, the directionality is reversed: The *forward punctuation* combinator, which consumes punctuation to the right of a constituent, is associated with backward application. An example for this is also given in Figure 4.1.

A slightly more complex use of applications occurs in subject control constructions. These constructions illustrate why any remaining placeholders must be merged after filling an argument slot: the merging allows the control verb to take the same subject as the controlled verb.

Example 4.17 (Subject control)**.** In the sentence *She wanted to sleep*, the subject is shared between the verbs *want* and *sleep*. On the semantic level, this sharing emerges naturally from the merging rules of GA-CCG, as is demonstrated in Figure 4.2.

As mentioned in Section 4.1.4, modifiers sometimes need to attach below the root of the modified graph, so our grammars use a special rule for modifiers of the form X/X and $X\backslash X$, as well as for determiners (NP/N).

Example 4.18 (Noun modification)**.** Nouns are sometimes represented by two-node graphs. One example is the word *teacher*, where the lexical meaning is split into a *person* node and a *teach* node. That is, the AMR representation for *teacher* can be expressed as *a person who teaches*. Either of the two nodes can be targeted by a modifier such as *new* or *school*. Figure 4.3 shows a CCG derivation for the phrase *school teacher*.

Syntactic arguments can sometimes be semantically empty. This is the case with syntactic expletives such as *it* or *there*: These arguments are required syntactically but do not appear in the meaning representation. We therefore provide a rule that allows expletives and other noun phrases to be ignored.

Example 4.19 (Expletives)**.** In phrases that start with *it is* or *there are*, the words *it* and *they* are frequently expletive, that is, semantically empty. At the same time, they assume the syntactic role of a noun phrase. An example is the phrase *there are seven bugs in the code*, which is analysed in Figure 4.4: While *are* requires a noun phrase argument to the left, this argument contributes no semantic content. The ignore operator can be applied in such contexts.

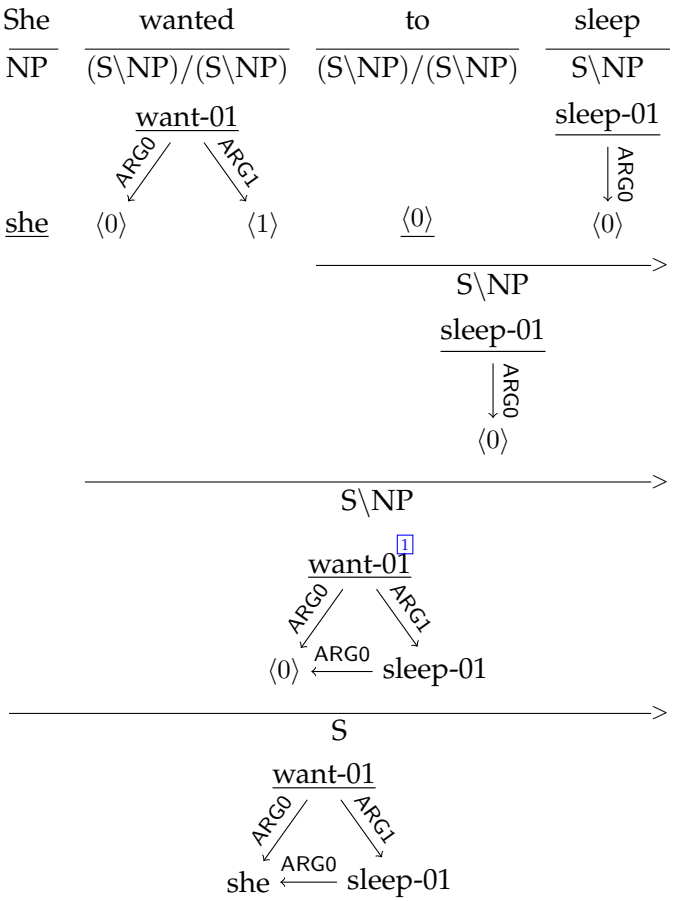


Figure 4.2.: A GA-CCG derivation for the sentence *She wanted to sleep*. The subject control structure is set up through the merging of the ⟨0⟩ placeholders during the application of *wanted* [1].

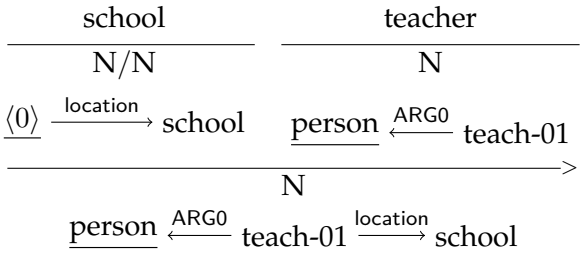


Figure 4.3.: The GA-CCG derivation for the phrase *school teacher* shows how the modification operator is employed as an interpretation for noun modification. It allows the modifier *school* to attach to the level-1 node *teach-01*.

4.2.6. Conjunctions

Conjunctions are extremely frequent, and therefore their proper treatment is important. For this purpose, the *substitute operator* **S** has been introduced in Section 4.1.6. This operator allows binary conjunctions of function categories with an arbitrary number of arguments.

Example 4.20 (Conjunction of verbs)**.** Employing the *substitute* operator for conjunctions ensures that function categories such as transitive verbs can be properly conjoined. The argument slots of the conjunction are kept separate from those of the conjuncts. The derivation in Figure 4.5 shows how this interpretation causes the correct dependencies to be established.

4.2.7. Composition

The most frequent composition combinators are $>\mathbf{B}$ and $<\mathbf{B}_{\times}$. In contrast, we observed no occurrences of the generalised composition combinators in the syntactic parser output for our training set, and therefore we do not include examples for these combinators (see Table 5.1).

A canonical use case for forward composition occurs when an extracted object is type-raised. Type raising reverses the roles of function and argument. The operator $\mathbf{B}^{\leftarrow}$ is used to assign the correct argument slot to type raised noun phrases.

Example 4.21 (Type raising and forward composition)**.** Forward composition allows two function types to combine. It typically occurs in combination with type raising and allows unconventional constituent types such as S/NP to be constructed. In other words, it allows argument slots to be filled in a nonstandard order. This is

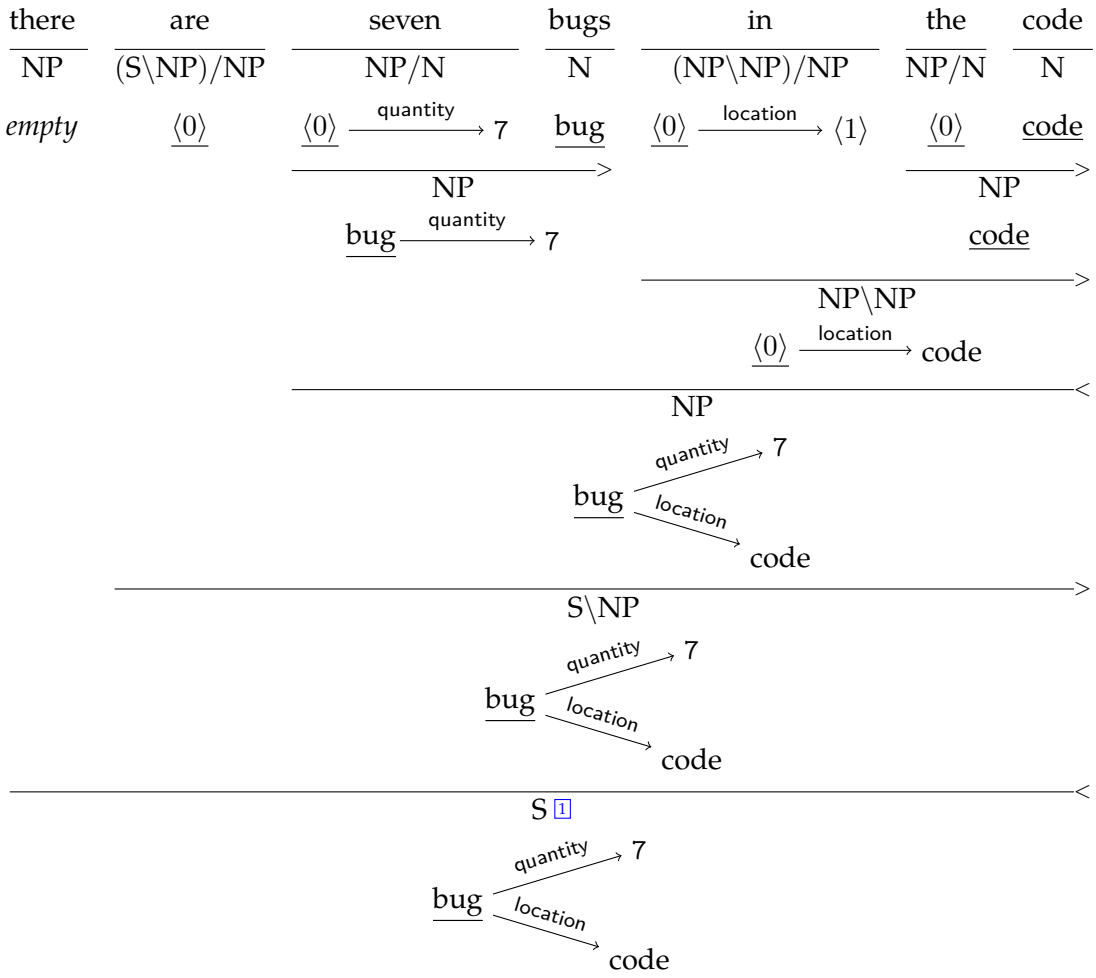


Figure 4.4.: Derivation of the phrase *there are seven bugs in the code*. In this phrase, *there* is expletive and represented by a dummy AMR. In the final derivation step, the dummy argument is ignored 1 to fill the syntactic argument without adding semantic content.

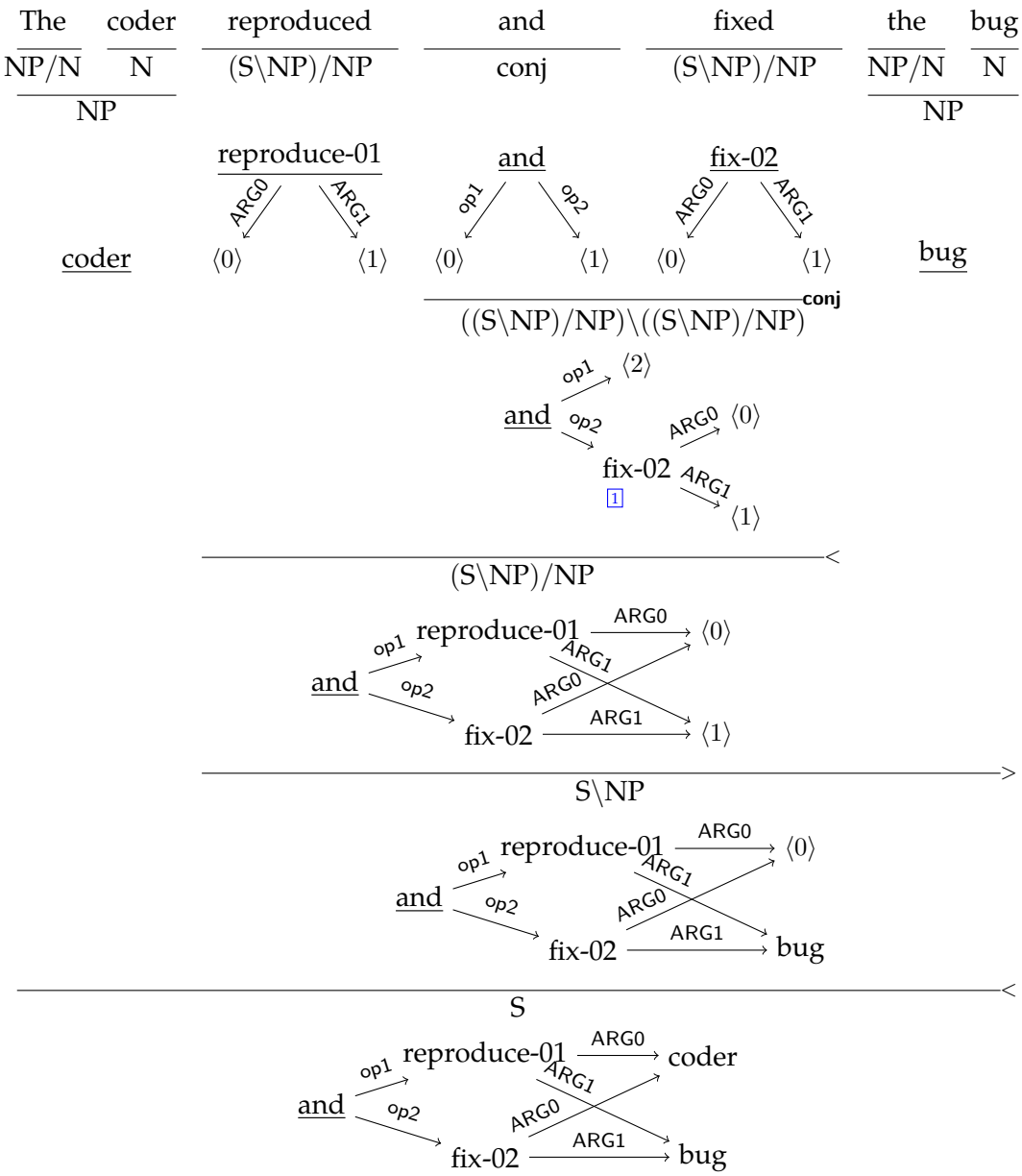


Figure 4.5.: This example demonstrates a conjunction of two transitive verbs. The **conj** combinator invokes the semantic operator **S**, keeping both arguments of *fix* free to merge with those of *reproduce* [1].

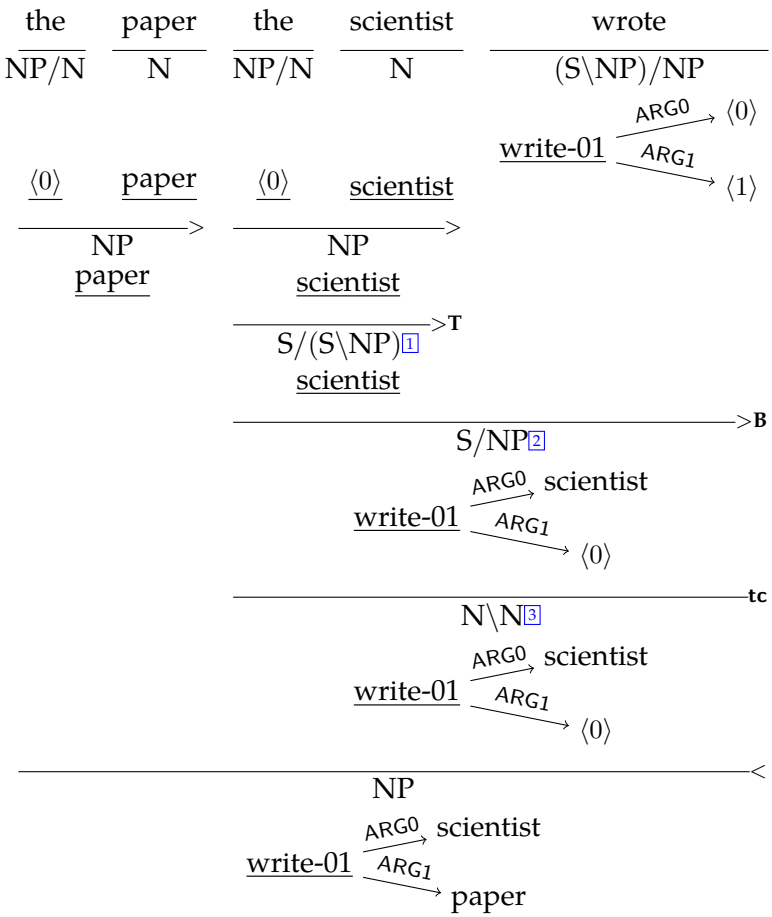


Figure 4.6.: Since the object is extracted from the relative clause *the scientist wrote*, type raising [1] and forward composition [2] are employed to allow the subject to combine with the verb. A type changing rule from EasyCCG's grammar allows the resulting phrase to act as a relative clause. [3]

evident in Figure 4.6, where a missing object in a relative clause means that the phrase *the scientist wrote* has to be constructed as a constituent.

Backward crossed composition is necessary to allow verbs to be affected by modifiers placed between the verb and its arguments. This can be the case in modal verb constructions (such as "What could *possibly* go wrong?") and reported speech, as shown in the following example.

Example 4.22 (Backward crossed composition)**.** In indirect speech, adverbials can be placed between the verb and the argument (the reported content). The $<\mathbf{B}_{\times}$ combinator allows this word order. Figure 4.7 shows an example.

4.3. Non-Compositional Operations

Not all constructions of AMR are easily represented in a strictly compositional manner, and while this work is focused on compositional phenomena, our implementation of GA-CCG contains two non-compositional additions to the pure CCG-based derivation process: coreferences and nested conjunctions.

4.3.1. Coreferences

Coreferences occur when an entity is mentioned more than once within a sentence. In such cases, a single sub-graph in the AMR represents two separate tokens or constituents. This makes it difficult to correctly establish semantic role relationships because the semantic representation is unavailable in one of the constituents in question.

Coreference nodes are a simple non-deterministic solution to this problem. Instead of instantiating the referenced concept twice, one of its instantiations creates a specially labelled coreference node instead. It can later be merged with an appropriate node to resolve the coreference.

Example 4.23 (Coreferences)**.** Reflexive pronouns such as *themself* invoke coreference nodes since their meaning cannot be represented in such a way that the proper dependencies are established compositionally, as shown in Figure 4.8. Instead, the coreference mechanism relies on a statistical model to merge the coreference node as appropriate after parsing has completed.

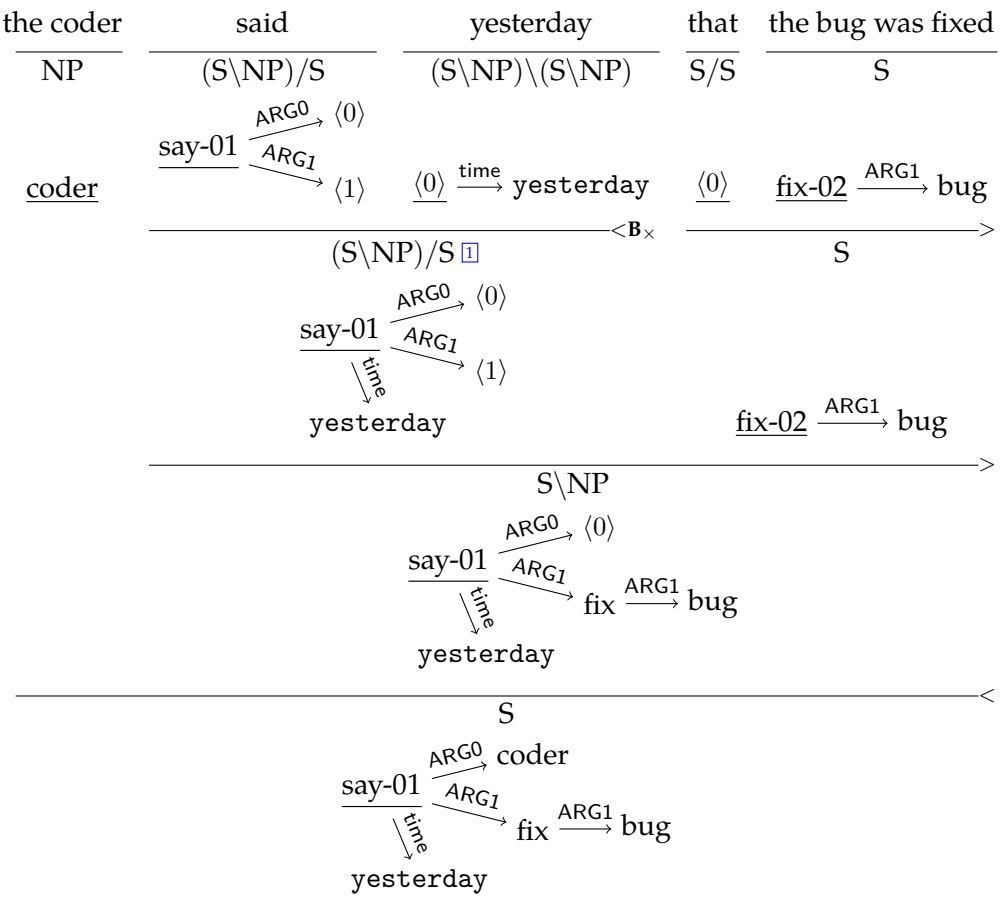


Figure 4.7.: One use case for backward crossed composition is to allow adverbials on verbs expecting additional arguments. Here, *said* is modified by *yesterday* [1].

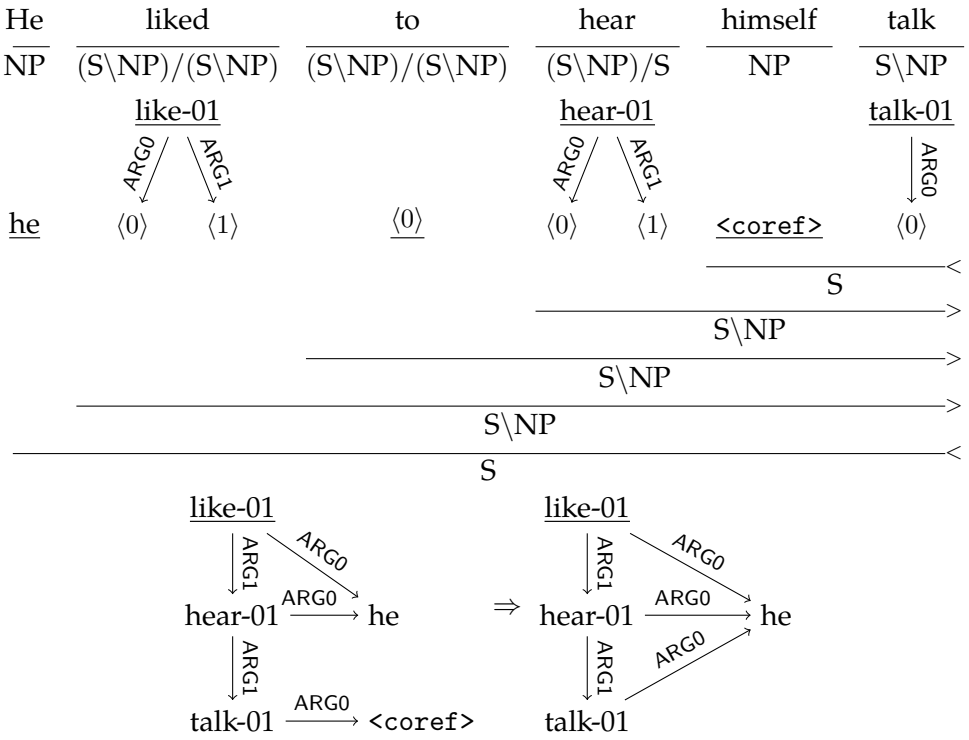


Figure 4.8.: A coreference node is used to represent the meaning of *themself*. At the end of the parsing process, the coreference node is merged with the appropriate referent.

4.3.2. Nested Conjunctions

Enumerations of more than two elements are represented in CCG derivations as right-branching trees, which poses a challenge since AMR represents them as flat lists of conjuncts. We therefore introduce a non-monotonic normalization rule, which is always applied following the **S** operator, and merges directly nested conjunctions.

Example 4.24 (Nested conjunction normalization)**.** Whenever a conjunction node (such as *and*) is an operand of another conjunction node of the same type after the **S** operator was invoked, both nodes are merged and their argument lists are combined. This operation rewrites part of the graph structure. It is not combinatory and therefore does not appear in the GA-CCG derivation. In Figure 4.9, the nested structure that is created before rewriting is shown for reference.

4.4. Limitations of GA-CCG

GA-CCG marries the unrelated formalisms of CCG and AMR. It also strives to do so not in a theoretically complete manner, but using simple, transparent rules and representations. It is therefore clear that there must be compromises in cases where the two formalisms diverge too far from each other, or where the statistical significance of an infrequent phenomenon does not warrant the introduction of additional complexity.

In this section, we discuss the most prominent limitations of GA-CCG from a linguistic point of view. These are phenomena which CCG handles well, but which the rules of GA-CCG do not, and cannot easily deal with. A more empirical overview of problems encountered when applying GA-CCG is provided in Section 5.4.

4.4.1. Relativisation and Type Raising

In relative clauses such as the one presented in Figure 4.6 (*the paper the scientist wrote*), type raising and composition are used to construct the constituent *the scientist wrote*, which correctly assigns *scientist* the ARG0-role of *write*. However, relativisation also expresses a focus shift towards the ARG1-role, which is expressed in the type changing rule from S/NP to NP\NP that is subsequently applied. It would be more correct for *paper* to be the root of the representation for *the paper the scientist wrote*. This becomes clear when the phrase is embedded in a sentence, such as the one

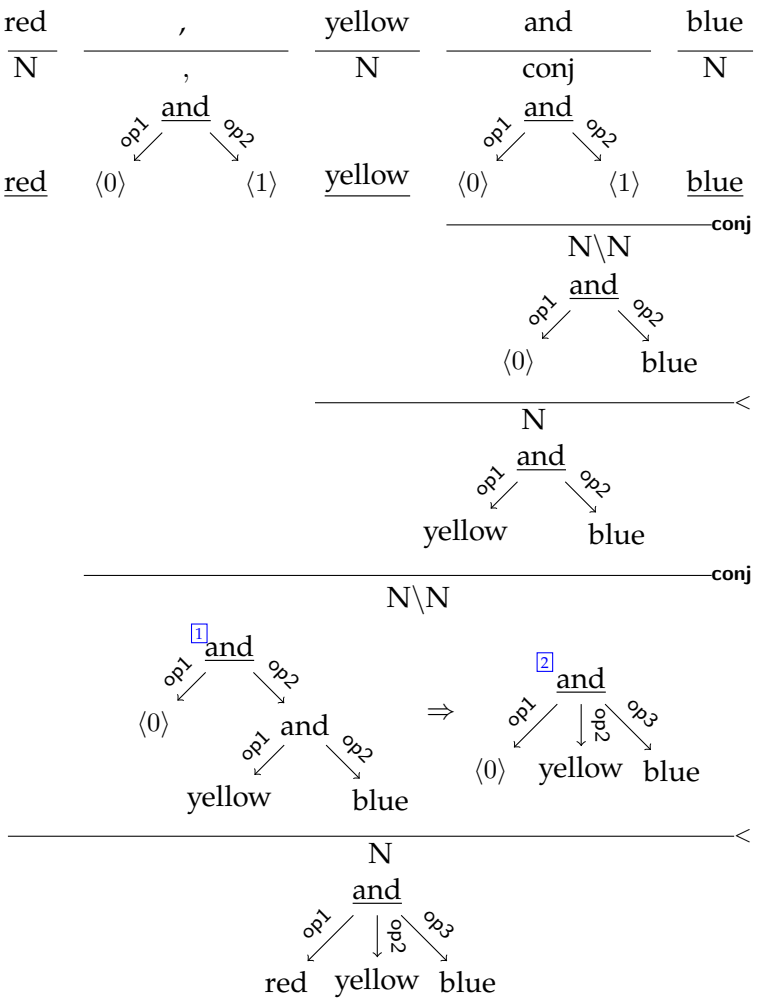


Figure 4.9.: Enumerations of more than two items initially cause conjunction nodes to be nested due to the binary-branching structure of CCG derivations 1. The normalization to the AMR-conforming flat structure is carried out immediately afterwards 2, without creating a new derivation node.

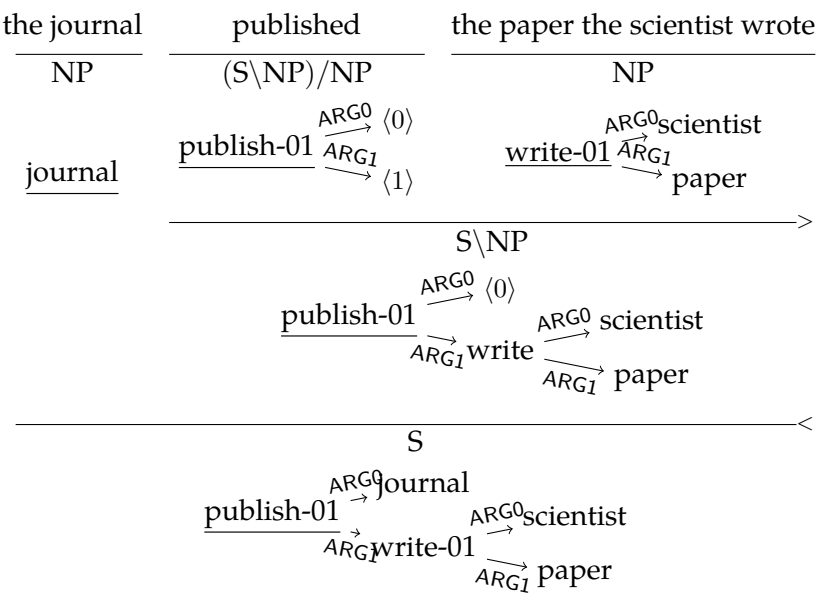


Figure 4.10.: When embedding the phrase from Figure 4.6 in a full sentence, it becomes clear that while the semantic roles are assigned correctly, the focus is incorrectly placed on the *write* node. As the paper is the thing being published, the ARG1 relation should be between *publish* and *paper*, not between *publish* and *write*.

shown in Figure 4.10, where an erroneous relation is constructed between *publish* and *write*.

While this behaviour could be encoded in the interpretation of the respective type changing rule, we consider it out of scope for this thesis.

4.4.2. Object Control

The simple and clean handling of control, where arguments are shared between verbs, has been an important argument put forward in favour of using graph algebras in semantic parsing (Koller 2015). In Figure 4.2, we have demonstrated how GA-CCG handles subject control. Unfortunately, object control verbs such as *ask* or *persuade* are not as cleanly modelled. While the subject control structure in Figure 4.2 emerges cleanly from the controlled verb's standard representation, a special argument-less representation is required to create an object control configuration, as Figure 4.11 shows.

4.4.3. Argument Cluster Coordination

CCG allows argument clusters of ditransitive verbs to be constructed as constituents and coordinated, as in *give* [*a teacher an apple*] *and* [*a policeman a flower*] (see Figure 4.12)[7]. Because GA-CCG requires semantic representations to be connected, there is no way to combine the noun phrases *a teacher* and *an apple* into a single semantic representation without a governing verb. Therefore, this construction cannot be represented in GA-CCG.

4.4.4. Non-Limitation: Substitution

The substitution combinator is conspicuously absent from the GA-CCG grammars defined in Section 4.2. The experiments presented in this thesis use the EasyCCG parser to obtain syntactic derivations. This parser outputs only a limited set of combinators that does not include substitution, implying that the substitution combinator is of limited statistical relevance at least in the CCGBank corpus on which the parser has been evaluated.

Nevertheless, the substitution combinator allows CCG to treat interesting constructions such as parasitic gaps, and is a central feature of the theory. Figure 4.13

[7]The example has been adapted from Steedman (2000, p. 46).

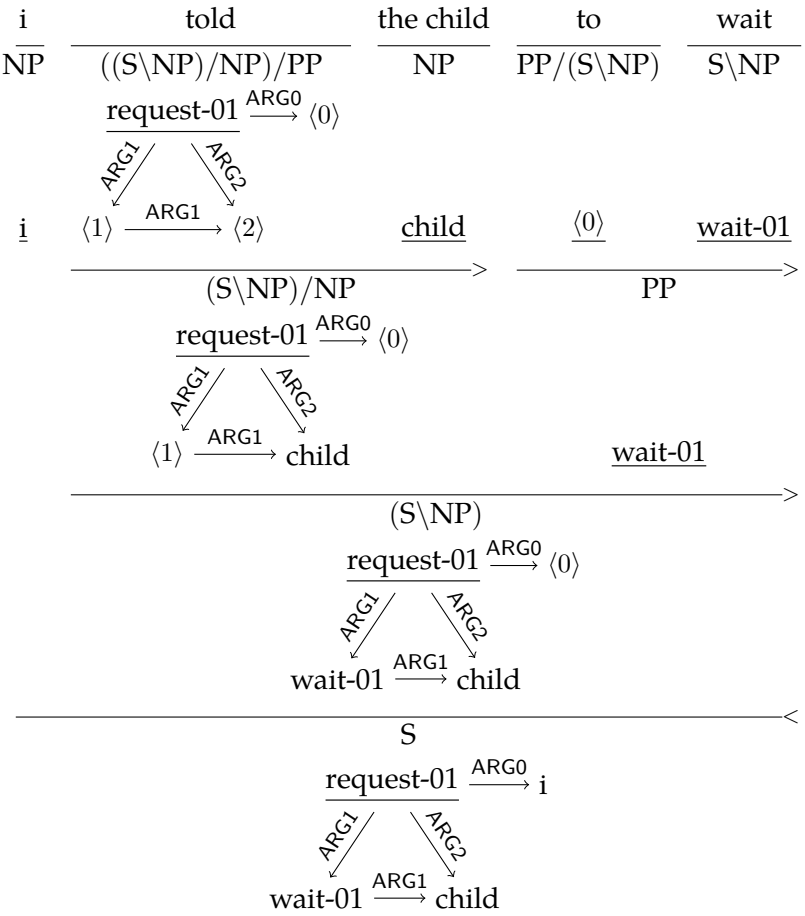


Figure 4.11.: In the sentences *I told the child to wait*, the object of *told* is the same as the subject of *wait*. Expressing this controlling behaviour requires a) introducing the ARG1-role of *wait* into the lexical entry for *told*, and b) adding an argument-less lexical entry for *wait*. This representation is not ideal because an interdependency between the lexical entries for both verbs is introduced.

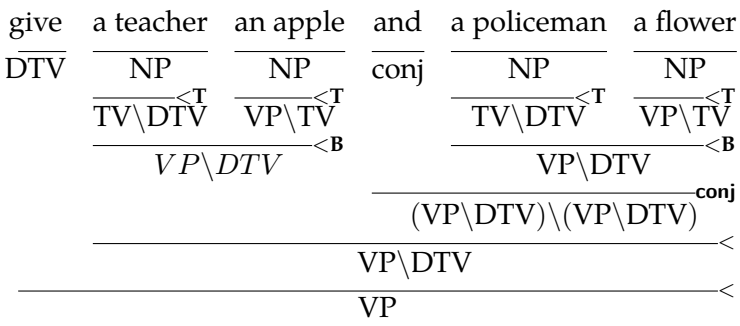


Figure 4.12.: With the backward composition combinator, CCG allows argument clusters such as *a teacher an apple* to act as constituents. However, the corresponding semantic representations cannot be constructed with GA-CCG: the resulting graph would not be connected as there is no verb to connect both arguments. For space reasons, the following abbreviations are used for syntactic categories: VP = S\NP, TV = (S\NP)/NP, DTV = ((S\NP)/NP)/NP.

shows how the substitution combinator can be interpreted using forward composition and backward crossed substitution rules.[8].

4.4.5. Discussion

The limitations discussed in this section are all caused by the inflexibility of GA-CCG operators regarding edges. An edge may (and must) occur in exactly one precursor graph, which, for instance, forces object-controlled verbs to omit their argument edges (because otherwise they would have to be merged with the controlling verb's edge). Blodgett and Schneider (2019) have proposed an alternative graph algebra for CCG-based AMR parsing, which crucially includes an additional *relation-wise application* operator. This operator, together with a special edge label for *underspecified relations*, permits simple and elegant interpretations for control, type raising, and by extension, argument clusters. Since there is so far no implementation of these operators, we do not further discuss them in this work; however, future work should consider including them.

While we demonstrated in Section 4.2 that GA-CCG permits the analysis of the bulk of natural language constructions, the examples in this section also show that

[8]The example has been adapted from Steedman (2000, p. 50).

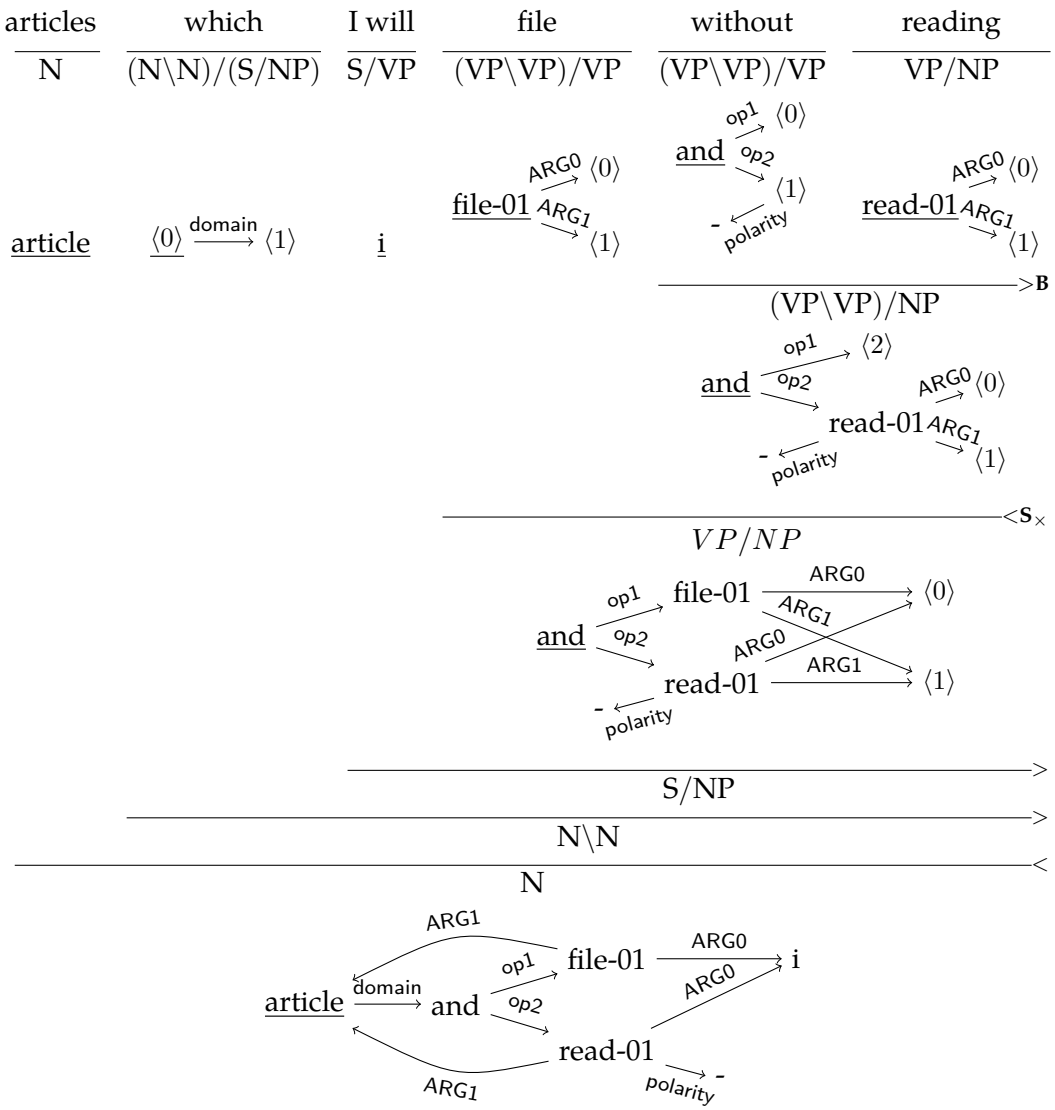


Figure 4.13.: A parasitic gap construction involving the substitution combinator can be modelled by interpreting the forward composition combinator as backward substitution, and interpreting the subsequent substitution as forward application. This interpretation identical to that of conjunctions (see Section 4.2.6). The abbreviated syntactic categories are the same as in Figure 4.12.

some notable linguistic phenomena are not covered. In practice, this is not necessarily an issue, as the main goal of this work is not to provide a linguistically complete grammar, but one with which larger text corpora can be processed effectively. In the following chapters, we will proceed to examine the actual coverage of the grammar on the AMR corpus.

Chapter 5.

Induction of Graph Algebraic CCG Lexica

The preceding chapter has described the theory of GA-CCG. From this chapter onwards, we turn to its implementation and investigate algorithms necessary for the practical application of GA-CCG to semantic parsing. This chapter deals with the problem of lexicon induction: how to obtain a GA-CCG lexicon from semantically annotated data. Chapter 6 introduces additional algorithms to clean up induced lexica and improve their ability to generalise to unseen texts. In this and the following chapter, the relevant concepts and algorithms are first described, followed by experiments to evaluate the performance of each component. Chapter 7 describes the implementation of a parser, with evaluation deferred to Chapter 8.

Building a CCG derivation starts with picking lexical items from the lexicon. The lexicon is therefore a required input for CCG parsing. While such a lexicon can be built by hand and there are projects in computational linguistics that have built sizeable lexica for some grammar formalisms, engineering a wide-coverage grammar is a daunting task which can consume many person-years.[1] In AMR parsing, the availablity of AMR-annotated corpora allows us to side-step this requirement and induce a lexicon automatically instead.

An AMR corpus consists of sentences paired with meaning representations, matching the desired input and output of a semantic parser. In this chapter, we assume that the corpus does not contain any additional information about the relationship between them.[2]

We approach lexicon induction by attempting to construct a GA-CCG derivation which yields both the sentence and the meaning representation. First, syntactic CCG derivations are generated by an external syntactic CCG parser. Then, the nodes of the derivation are annotated with meaning fragments which are obtained

[1] For instance, the effort for building the English Resource Grammar has been estimated at eleven person-years (Copestake and Flickinger 2000).

[2] Newer versions of the AMR corpora do include alignment annotations, which can be included in our algorithm (Knight, Bianca Badarau et al. 2020).

by breaking the full AMR into GA-CCG s*-graphs according to the rules of GA-CCG. If this process is successful, the leaves of the derivation are added to the lexicon as new lexical entries.

In this chapter, an algorithm for constructing GA-CCG derivations of corpus examples is introduced in Section 5.1. Section 5.2 covers some practical considerations such as parameters and additional constraints that are important for managing computational complexity. In Section 5.3, experiments on a large section of the AMR corpus are presented, which serve to validate the algorithm and find suitable parameters. Section 5.4 presents experiments on a small corpus section which examine the coverage achieved by GA-CCG grammars under ideal conditions.

5.1. Algorithms for Lexicon Induction

In this thesis, we consider the scenario that we are provided with an AMR annotated corpus and wish to induce a lexicon suitable for parsing similar sentences to those in the corpus. We also assume that the specific set of GA-CCG rules to be used is given as an input to the algorithm, as we wish to examine specific rule sets and their utility for parsing. The problem of GA-CCG lexicon induction can thus be stated as follows.

Definition 5.1 (GA-CCG Lexicon Induction)**.** Given a set of GA-CCG rules R and an AMR corpus $(s_1, g_1), \ldots, (s_n, g_n)$ consisting of sentences s_i paired with AMRs g_i, construct a set of GA-CCG lexical items L such that for every example (s_i, g_i), there exists a GA-CCG derivation d_i with the following properties:

- d_i uses only lexical items from L and rules from R
- $\textsc{tokens}(d_i) = s_i$ (the derivation derives the given sentence)
- $\textsc{sem}(d_i) = g_i$ (the derivation yields the given meaning representation)

It may be difficult or impossible to perfectly solve the lexicon induction problem as stated above because the rules R might not permit it. In particular, the meaning representations g_i are AMRs that are not in any formal way related to CCG syntax, and therefore might not be derivable using GA-CCG. Some such phenomena are discussed in Section 4.4.

Even in examples with derivable AMRs, the space of possible lexical items can be very large, as an AMR graph may be partitioned into a number of subgraph pairs that is exponential in the number of its nodes. Although in Section 5.2 we address

ways to reduce the number of candidate lexical items, it is possible for the candidate space to be too large to enumerate in practice.

Nonetheless, the problem as stated is precise enough to serve as a guideline for implementations. In Sections 5.3 and 5.4, it will be examined to what extent the algorithms presented here solve it in practice.

5.1.1. Syntax-Driven Lexicon Induction

We solve the lexicon induction problem by attempting to construct a syntactic-semantic GA-CCG derivation for every example in the training corpus. Furthermore, we take advantage of existing syntactic CCG parsers. Given a derivation produced by a syntactic parser, we are left with the task of annotating the derivation with GA-CCG graphs so that the derivation produces the gold-standard annotated AMR according to the rules of GA-CCG. Afterwards, the lexical items that have been induced can be read off the leaves of the derivation. This means that our lexicon induction problem is reduced to finding the right lexical meaning representations, whereas the syntactic part of the lexicon is supplied by the parser.

This annotation process is implemented using a syntax-directed splitting strategy. A recursive splitting algorithm walks down the derivation tree, starting at the root with the annotated meaning representation. At every combinatory node, it splits the meaning representation, producing meaning fragments in the form of GA-CCG s*-graphs which are assigned to the constituents—and finally, leaves—of the derivation.

Design Considerations

Employing an external syntactic parser is a design decision which comes with advantages and disadvantages. An alternative approach would be to enumerate possible syntactic and semantic categories at the same time, without relying on external tools (Kwiatkowski et al. 2010). Such an approach has to deal with searching over the exponential space of syntactic derivations and, without syntactic supervision, may end up producing linguistically implausible derivations.

On the other hand, if the syntactic structure is fixed, there is the risk of mismatches between the syntactic and semantic structure, which can hinder lexical induction. This can be the case because of errors produced by the syntactic parser, or because of systematic mismatches. Parser errors can partly be mitigated by using more than one candidate derivation per sentence, which increases the probability that at least one derivation is suitable for lexical induction.

In summary, assuming an externally provided syntactic derivation allows us to induce linguistically plausible lexical items with relative computational efficiency, at the expense of a certain loss of coverage.

Algorithm

The procedure of syntax-directed splitting is described in Algorithm 5.1. The SplitSyn takes a syntax derivation d as well as a meaning representation graph g as input. It can thus be called several (or zero) times for any given sentence depending on how many syntax derivations are available.

The algorithm starts with the derivation's root node and calls itself recursively for every child node. For every node in the derivation, it chooses an action according to the number of child derivation nodes:

- If there are two children, the node corresponds to a binary CCG combinator. In this case, the meaning representation needs to be split into components which can be assigned to the two child nodes in the recursive call. The algorithm thus searches for a rule that is applicable in the current context using the rule predicate p. For every matching rule, the meaning representation is then broken up by the SplitSem algorithm which is discussed in Section 5.1.2.
- If there is only one child, the node corresponds to a unary type raising or type changing rule. As explained in Section 4.4, the meaning representation is left unchanged and the algorithm continues at the child node.
- If there are no children, the algorithm has arrived at a leaf of the derivation, which corresponds to an individual token. The algorithm generates a lexical item from the meaning representation fragment that has been produced by the preceding recursive steps and adds it to the result set.

In addition to the lexical items read off the derivation's leaves, Algorithm 5.1 also creates phrasal lexical items at combinatory nodes if the result set of SplitSem is empty for all rules. In this case, the algorithm cannot descend to the token level and multi-token phrasal items are required in order to fully describe the example. This is especially important when estimating probabilities of lexical entries using EM (see Section 6.2), since the probability of an example would be zero if part of it could not be explained using the lexicon.

Algorithm 5.1 describes how all possible lexical items are enumerated from an entire sentence. In the following section, we examine how meaning representation fragments may be generated at each node of the syntax derivation.

5.1.2. Constrained AMR Splitting

Through recursive splitting, a constituent is divided into two smaller constituents. In CCG, each constituent corresponds to an adjacent span of tokens in the sentence, so that the larger constituent is divided into two adjacent smaller constituents. Accordingly, we wish to split the constituent's meaning representation into two smaller components, each corresponding to one of the child constituents. We call these components *precursor graphs*, as they can be used as an input to a semantic operator which then reconstructs the original graph.

The result of AMR splitting is a pair of two graphs, *left* and *right*. In GA-CCG, we can also think of the graphs playing the role of a *function* (the left graph for forward rules, the right graph for backward rules) and an *argument*.

In preparation for splitting the meaning representation, we assume that a complete *binary node labelling* is available which maps every node to one of the sides.

Definition 5.2 (Binary Node Labelling)**.** Let V be a set of vertices and $L = \{l_1, l_2\}$ a set of two labels. A *binary node labelling* of V is a mapping $m : V \to L$.

For illustration, we will assume in the following that $L = \{\textsc{Left}, \textsc{Right}\}$.

In our experimental settings, binary node labellings are obtained from token-to-node alignments as explained in Section 5.1.3.

Inverse Semantic Operators

Splitting an AMR graph g requires finding precursor graphs g_1, g_2 so that g can be constructed from g_1 and g_2 using some semantic operator. In other words, an inverse computation for semantic operators is to be defined.

Since semantic operators are not injective, there can be many such pairs of predecessors. In particular, nodes can be merged in the process of applying a semantic operator, and the information on which nodes were merged is erased from the resulting graph. The inverse of an operator therefore generates predecessor pairs for every node that can be *unmerged*.

Definition 5.3 (Inverse Semantic Operator)**.** Let o be a semantic operator. We define its *inverse* o^{-1} as follows:

$(g_1, g_2) \in o^{-1}(g) \Leftrightarrow o(g_1, g_2) = g$ for GA-CCG s*-graphs $g, g_1,$ and g_2

In practice, this definition results in too many predecessor pairs to iterate over. In the context of recursive splitting, every predecessor pair results in an extra recursive invocation, which may cause the number of lexical items to blow up. The algorithms

defined in the following therefore use a constrained form of inverse operators that take into account a node labelling.

Definition 5.4 (Constrained Inverse Semantic Operator)**.** Let o be a semantic operator, $g = (V, E, v_{\text{root}}, l, \mathit{slab})$ an GA-CCG s*-graph, and $m : V \to \{\textsc{Left}, \textsc{Right}\}$ a binary node labelling.

Let $g_1 = (V_1, E_1, v^1_{\text{root}}, l_1, \mathit{slab}_1)$ and $g_2 = (V_2, E_2, v^2_{\text{root}}, l_2, \mathit{slab}_2)$ be GA-CCG s*-graphs. The *constrained inverse semantic operator* o^{-1}_m is defined as follows:

$(g_1, g_2) \in o^{-1}_m(g)$ iff. the following conditions hold:

1. $o(g_1, g_2) = g$
2. $\forall v \in V\colon\ m(v) = \textsc{Left} \Leftrightarrow v \in V_1$
3. $\forall v \in V\colon\ m(v) = \textsc{Right} \Leftrightarrow v \in V_2$

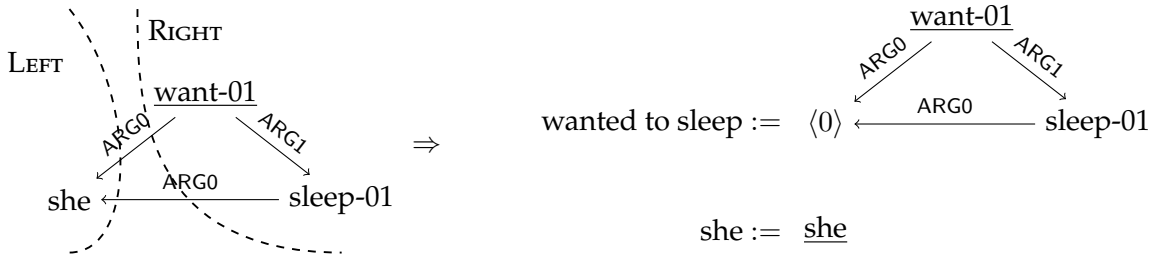
The implementation of the inverse semantic operators requires some care because it is not feasible to simply enumerate all possible precursor graphs. However, the rules of GA-CCG permit a relatively simple strategy: The only nodes whose environment needs to be altered are nodes that are labelled Left but are connected to nodes labelled Right. We call these nodes *frontier nodes*.

For these frontier nodes, actions are necessary to allow the separation of the graph into two disconnected components. They are *unmerged* by duplicating the node, assigning all `Right` edges to the copy. Then, one copy can be replaced by a placeholder or coreference according to the rules of GA-CCG (see Section 5.1.4 for details on coreference extraction).

Example 5.5. The examples in Figure 5.1 demonstrate how fully labelled graphs representing various grammatical constructions are split. In subject-predicate-object constructions such as *she wants to sleep*, which correspond to semantic application, the subject is replaced by a placeholder node to make up the function graph. The same happens, less obviously, in the phrase *the cat*, where the determiner *the* plays the function role but has no semantic content of its own[3]. It is assigned a graph consisting solely of a placeholder. Finally, splitting conjunctions such as *she ate and slept* involves the renaming of placeholders in the argument graph, but otherwise works equivalently.

[3]While in other semantic formalisms, determiners do commonly introduce semantic content such as quantifiers, AMR does not represent this aspect of semantics.

she wanted to sleep := S



the cat := NP

[she ate] and slept := (S\NP)\(S\NP)

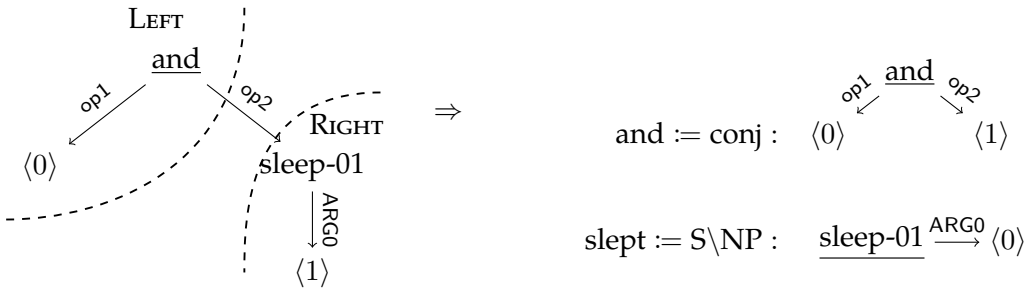


Figure 5.1.: Examples for constrained splitting.

Node Labelling Completion

Token-to-node alignments that are generated by alignment tools or found in gold-standard data do not necessarily cover all nodes. However, a complete labelling is required by the constrained inverse semantic operators as defined in Section 5.1.2.

Without consulting any additional sources of information, we cannot judge the correctness of one node labelling over another, provided that neither contradicts the annotated alignments. Our algorithm therefore takes a brute-force approach to node labelling completion.

Definition 5.6 (Completion of Node Labellings)**.** Let V be a set of vertices, $V' \subseteq V$, and $m : V' \to \{\textsc{Left}, \textsc{Right}\}$ a binary node labelling of V'.

A mapping $m' : V \to \{\textsc{Left}, \textsc{Right}\}$ is a *completion* of m to V iff. $m'(v) = m(v)$ for all $v \in V'$.

In the following, the set of all completions of m to V is written $\textsc{Completions}(m, V)$.

Constrained Graph Splitting

Given an initial, possibly incomplete node labelling, an AMR graph can be split by iterating over all completions of the labelling attempting to generate pairs of precursor graphs for each of them. This procedure is captured in Algorithm 5.2. Since the set of extensions is exponential in size, the algorithm potentionally generates an exponential number of precursor graph pairs. In practice, this means that the algorithm is only feasible if the number of unlabelled nodes does not exceed a certain threshold.

In the following sections, we discuss in detail how to obtain the inital node labelling from token-to-node alignments.

5.1.3. Alignment Constraints

The principle of compositionality in the strict form that is assumed by GA-CCG (and CCG in general) implies that every element of a meaning representation should be traceable to a lexical item and thus, to a token in the surface form. Even without an underlying theory, it is possible for many nodes in an AMR to intuitively point out the token that invoked the corresponding concept.

Based on this intuition, it is possible to create tools that create alignments between tokens and nodes of the AMR. While some tools also create alignments between tokens and edges, we disregard edge alignments because the placement of edges is dictated by GA-CCG rules.

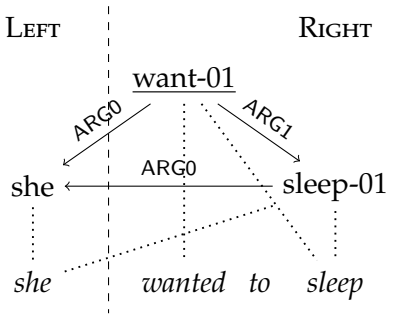


Figure 5.2.: Illustration for Example 5.8.

A node labelling is easily derived from an alignment: If a node is aligned to a token belonging to the left constituent, it is marked as `Left`; if its aligned tokens belong to the right constituent, the node is marked as `Right`. A node might also have alignments to tokens from both constituents, or no alignments at all; in this case it is left unmarked. This partial labelling is the *alignment-based node labelling* of the graph.

Definition 5.7 (Alignment-Based Node Labelling)**.** Let V be a set of vertices and $a : V \to \mathcal{P}(\mathbb{N})$ a node alignment which assigns each node a set of token indices.

Given an integer i, two *unambiguous subsets* of V can be defined. These vertices are aligned to indices that are all less than or equal to i, or all greater than i, respectively.

$$V_{\text{LEFT}}(i) = \{v \in V : |a(v) \geq 1| \wedge \forall j \in a(v) : j \leq i\}$$
$$V_{\text{RIGHT}}(i) = \{v \in V : |a(v) \geq 1| \wedge \forall j \in a(v) : j > i\}$$

For the unambiguous subsets of V, the binary node labelling $m_a^i : V_{\text{LEFT}}(i) \cup V_{\text{RIGHT}}(i) \to \{\text{LEFT}, \text{RIGHT}\}$ can be defined as follows:

$$m_a^i(v) = \begin{cases} \text{LEFT} & \text{if } v \in V_{\text{LEFT}} \\ \text{RIGHT} & \text{if } v \in V_{\text{RIGHT}} \end{cases}$$

We call m_a^i the *alignment-based node labelling* of V and a for index i.

Example 5.8. As an example, consider that the phrase *she wanted to sleep* is split after the word *she* (see Figure 5.2). In our example, the node *she* is properly aligned to the corresponding token, causing it to be assigned the label `Left`. The node *want* is aligned to both *wanted* and, erroneously, *sleep*, but since both tokens are in the right constituent, the node can be labelled *right*. The node *sleep* has alignments to both constituents, and therefore cannot be labelled.

5.1.4. Coreferences

In Section 4.3.1, a mechanism for handling of coreferences introduced. It was proposed that coreferences be merged with their referent nodes in a post-processing step after parsing. Mirroring this procedure for lexicon induction by unmerging arbitrary nodes in the meaning representation is infeasible as it is unknown a priori which and how many nodes should be turned into coreferences.

Instead, we allow coreferences to be created in every splitting step by unmerging a frontier node (a node assigned to the *function* side but with links to the *argument* side). This is a straightforward extension of the implementation of inverse semantic operators sketched in Section 5.1.2.

When more than one frontier node exists, there is an ambiguity as it is not clear which node should become a placeholder and which a coreference. Our algorithm always produces all feasible permutations of placeholders and coreferences. In contrast to placeholders, coreferences can also be unmerged to either side, leaving the lexical content either on the function or the argument side.

5.2. Recursive Splitting in Practice

Even when syntax and alignment restrictions are taken into account, the lexicon induction algorithm is still a brute force process which can cause the number of created lexical items to grow exponentially. Apart from the obvious computational problems implied by such growth, it also means that a large number of "noisy" lexical items may be introduced which are not useful for analysing novel sentences.

In this section, we introduce additional heuristics that either limit the number of lexical items that are induced and thus may reduce the amount of superfluous items in the lexicon, or that allow the lexicon induction algorithm to cover a larger part of the corpus.

5.2.1. Connectedness of Precursor Graphs

Arbitrary partitionings of graphs may result in disconnected subgraphs. While certain semantic phenomena may best be represented by disconnected graphs,[4] they are statistically rare. Intuitively, a constituent meaning should be representable by a connected graph. We therefore impose the restriction that all precursor graphs be connected and reject splits where this is not the case.

5.2.2. Limitation of Lexical Items per Syntax Derivation

When the token-node alignments do not sufficiently restrict splitting, this quickly leads to an exponential increase in the number of induced lexical items. This may lead to computation and memory issues: it may take an indeterminate amount of time and/or memory to complete the induction algorithm for any given derivation.

We therefore impose a limit on the number of lexical items that may be induced in a run of the induction algorithm (that is, per syntax derivation). Once this limit is hit, the run of the algorithm is aborted and all induced lexical items are discarded.

Apart from ensuring that the induction algorithm completes for every derivation, this also avoids the issue of tens of thousands of useless lexical items being added to the lexicon.

Unless otherwise noted, this limit is set to maxLexicalItems=10 000.

5.2.3. Limitation of Unaligned Nodes

The main cause of exponential growth in the number of lexical items is sparse token-node alignments. When many nodes in a meaning representation are unaligned, the induction algorithm will branch out over all permutations of their assignments. Sentences exceeding a certain number of unaligned nodes can therefore be skipped before induction is even started, so no computation has to be spent at all.

An alternative option for controlling the number of unaligned nodes would be to define a permitted ratio of unaligned nodes or tokens, based on the sentence's length. However, the motivation for this limitation is to manage computational complexity, which is driven by the number of alignment choices that the induction algorithm has to iterate over. For short sentences, it may be feasible to iterate over all alignment options even if a high proportion of nodes are unaligned, and we do not wish to exclude such sentences.

Unless otherwise noted, this limit is set to ten nodes (maxUnalignedNodes=10).

[4] See the discussion of argument cluster coordination in Section 4.4.3.

5.2.4. n-Best Parsing and Filtering by Token Coverage

Errors produced by a syntactic parser can cause the induction algorithm to abort. However, parsers such as EasyCCG are able to produce a ranked list of n best parses, offering an opportunity to improve coverage by selecting one of several possible derivations. To evaluate these parses, we introduce the measure of *token coverage*: the fraction of tokens of a given sentence for which lexical (non-phrasal) items could be induced.

A parse with a high token coverage is not necessarily more correct. Sometimes, unusual syntactic constructions can match the structure of a meaning representation more closely than the canonical alternative. Also, in the presence of ambiguities, there are several equally correct derivations to choose from, only one of which might match the interpretation represented in the semantic annotation.

To reduce the amount of superfluous lexical items, we run the induction algorithm for the top n derivations output by the parser, but keep only the items produced by the derivation which achieves the highest token coverage. In the event that there are several derivations achieving the same token coverage, only the first derivation is used (according to the ranking assigned by the syntax parser).

In most of the experiments in this chapter, the number of syntactic derivations is set to derivations=10, while experiments in the following chapters use the setting derivations=50.

5.2.5. Syntactic Arity Checking

CCG's syntactic categories are functional types. A word or constituent is interpreted as a function taking a certain number of arguments. In traditional CCG, where meaning representations are expressed in λ-calculus, a direct correspondence between the meaning representation's type and the syntactic category of the word or constituent is established. The arity, that is, the number of expected arguments, must match between both. Through the mechanics of λ-calculus, this correspondence is upheld automatically.

In GA-CCG, such a correspondence does not exist because the types of λ-calculus do not apply to s*-graphs. However, because every placeholder node represents an argument slot that may be filled, the number of placeholders can be considered the graph's arity. It can be argued that it would be implausible to assign the graph a syntactic category of lower arity: since every GA-CCG derivation step only fills one argument slot, placeholders would be left over at the root of the derivation.

Items where the semantic arity exceeds the syntactic arity can be excluded right

at the time of lexicon induction, saving computational effort and reducing lexicon size. In practice, assigning the proper arity to a syntactic category is not as simple as it seems on the surface. For example, the category PP (prepositional phrase) is atomic, but semantic representations of prepositional phrases tend to require a placeholder for the content modified by the phrase. We therefore employ the following rules to determine the arity of a syntactic category:

Definition 5.9 (Syntactic Arity)**.** Given a syntactic category X, the syntactic arity $\textsc{SyntacticArity}(X)$ is defined as followed:

$$\textsc{SyntacticArity}(X) = \begin{cases} \textsc{SyntacticArity}(Y) + 1 & \text{if } X = Y/Z \text{ for some } Z \\ \textsc{SyntacticArity}(Y) + 1 & \text{if } X = Y\backslash Z \text{ for some } Z \\ 2 & \text{if } X \in \{\text{conj}, ,, ;\} \\ 1 & \text{if } X \in \{\text{PP}\} \cup \textsc{punct} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

where

$$\textsc{punct} = \{\text{LRB}, \text{RRB}, \text{LQU}, \text{RQU}, .\} \tag{5.2}$$

By default, arity checking is employed in our experiments (arityCheck=true).

5.3. Large-Scale Lexicon Induction

In the experiments described in this section, we induce a lexicon from a large section of the AMR corpus as a first sanity check of the lexicon induction algorithm. The experiments provide a basis for choosing a set of parameters, most notably a reasonable choice of alignment tools. They also allow us to gain an overview of the behaviour of the algorithm regarding coverage and computation requirements.

5.3.1. Setup

The data set used in this experiment is the `proxy-train` section of the AMR 1.0 corpus (see Section 5.3.3 below).

We vary the configuration of the induction algorithm along various dimensions:

- Alignments from several AMR alignment tools are used individually and in various combinations.

- Various sets of grammar rules are employed.
- The number of CCG derivations from which the best derivation is chosen is varied.
- The maximum number of coreference nodes allowed per derivation step is varied.

Syntactic derivations are obtained from the EasyCCG parser.

To allow all runs to complete within the available memory on the experimental system (48 GB of JVM heap size), we set maxUnalignedNodes=10. Computation times are measured on a compute node with two Intel Xeon E5-2630v3 CPUs of eight cores each, allowing 32-fold parallelisation.

5.3.2. Key Metrics

We evaluate every condition according to the following metrics:

- Token coverage: Percentage of tokens for which lexical items are induced
- Lexicon size: Total number of unique lexical items induced
- Runtime: Time required to perform lexicon induction on all sentences

Token coverage suffers when the lexicon induction algorithm aborts, as phrasal items, which are created if the algorithm stops prematurely, are not counted. As such, it is the most direct measure of the success of lexicon induction.

Ideally, the lexicon should be just as large as necessary. That is, every distinct sense of a word that occurs in the corpus should receive a lexical entry, but the amount of "noisy" lexical items which do not accurately represent a word's semantics should be limited to a minimum. Since we do not have direct evidence for the minimum number of lexical items required to represent the corpus, in the evaluation, we simply assume that a larger lexicon size indicates more noise in the lexicon, and that therefore a smaller lexicon is preferable if it achieves the same level of token coverage.

The algorithm's runtime varies as annotations and grammar rules (which impose constraints upon the induction algorithm) permit a smaller or larger search space. Considering that lexicon induction is run repeatedly by the EM algorithm introduced in Chapter 6, runtime can be an important factor for the practicality of certain configuration settings.

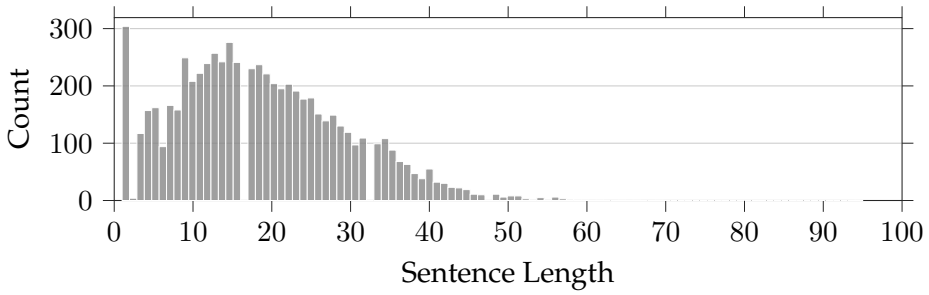


Figure 5.3.: The distribution of sentence lengths in the `proxy-train` data set.

5.3.3. Corpus

The `proxy-train` section of the AMR 1.0 corpus consists of 6.603 sentences of newswire text with a median length of 17 tokens. There are a large number of very short sentences (see Figure 5.3), which is due to the format of the included texts: Texts always start with an indication of a date, location, and keywords, which are included in the corpus as individual sentences[5].

Table 5.1 shows the occurrences of CCG combinators in the derivations produced by the EasyCCG parser. Application combinators are by far the most frequent, followed by punctuation and conjunction. In particular, type raising, type changing, and composition combinators are very rare and together make up less than 3 % of occurrences.

5.3.4. Comparing Alignment Strategies

Alignments between tokens and meaning representation nodes are important constraints for the lexicon induction algorithm and can influence the outcome to a large degree: if alignments are too sparse, the search space of lexical items grows exponentially, while incorrect alignment edges may preclude desirable lexical items from being found and even cause the algorithm to stop prematurely.

Various algorithms for aligning AMR elements have been introduced in the literature. Here, we use four tools which employ a range of techniques ranging from rules and heuristics to unsupervised learning.

- The *JAMR* system (Flanigan et al. 2014), one of the earliest AMR parsers,

[5]Although they are not very interesting, we include these sentences in the evaluation to allow comparisons with other systems.

Combinator	Frequency
fa	54.36 %
ba	18.32 %
lex	16.43 %
rp	4.80 %
conj	3.56 %
fc	1.31 %
bx	0.87 %
tr	0.24 %
tc-nmod	0.05 %
lp	0.04 %
tc-rel	0.04 %
gfc	0.00 %

Table 5.1.: Relative frequencies of combinators in `proxy-train` across the top 50 derivations produced by EasyCCG.

includes a rule-based aligner that creates alignment links based on lexical similarity and knowledge of common AMR patterns. This aligner links token sequences to AMR sub-graphs.

- The *TAMR* system (Liu et al. 2018) incorporates an AMR parser and an aligner. Similarly to the JAMR aligner, the aligner is based on rules. However, while the former deals with conflicting rules heuristically by prescribing a fixed precedence, the TAMR aligner runs an AMR parser on every possible alignment, and then outputs the alignment which leads to the highest-scoring parse, resulting in a *tuned* alignment.

- The *amr_ud* aligner (Szubert, Lopez and Schneider 2018) is a rule-based aligner which uses syntactic dependency trees and hierarchically aligns sub-trees of the dependency tree to sub-graphs of the meaning representation. Of these hierarchical alignments, we use only those at the token level.

- The *ISI aligner* (Pourdamghani et al. 2014) learns AMR-to-string alignments in an unsupervised fashion using an Expectation Maximisation (EM) algorithm based on the IBM machine translation models (Brown et al. 1993). This training method commonly used in statistical machine translation is extended by symmetrised EM training.

Aligner	Method	Sentence Part	AMR Element
jamr	Rules	Token Span	Node
tamr	Rules + Parser Tuning	Token Span	Node
amr_ud	Rules on Syntax Tree	Dependency Sub-Tree	Sub-Graph
isi	Expectation Maximisation	Token	Node or Edge

Table 5.2.: An overview of AMR alignment tools.

	vote1	vote2	vote3
none	n/a	–	–
jamr	0.5905	–	–
tamr	0.6105	–	–
amr_ud	0.3584	–	–
isi	0.5738	–	–
jamr+tamr+amr_ud	0.6534	0.6316	0.3474
jamr+tamr+isi	0.6409	0.6352	0.5598
jamr+amr_ud+isi	0.6446	0.5646	0.3480
tamr+amr_ud+isi	0.6598	0.5785	0.3584
jamr+tamr+amr_ud+isi	0.6623	0.6437	0.5572

Table 5.3.: Token coverages achieved using various combinations of alignment tools.

Table 5.2 provides an overview of the tools' features.

As these aligners employ a variety of algorithms, they can be expected to have distinct strengths and weaknesses and it could therefore be beneficial to work with the combined output of several tools. In the experiments, simple voting combinators **vote**k are used to select token-to-node alignments from several tools, where k indicates the number of tools that must agree on an alignment for it to be included in the output. **vote1** includes a token-node pair if it occurs in the output of at least one tool and is therefore equivalent to forming the union of all alignments, whereas **vote**n is equivalent to forming the intersection if n is the total number of alignment tools.

To understand what aligners are best suited for GA-CCG lexical induction and how they should be combined, we measure the token-level coverage achieved in the following conditions:

- Each aligner individually
- All four aligners combined using **vote1** through **vote3**
- Each combination of three aligners using **vote1** through **vote3**

The resulting token-level coverages for these conditions are compiled in Table 5.3. An additional condition using no alignments at all did not complete within a six-hour time window and is therefore not included in the results. The same is true for the condition **jamr+tamr+amr_ud+isi** with the **vote4** combinator.

The results show two clear trends. Firstly, combinations of aligners are better than individual aligners. As hypothesized, the strengths of the individual aligners complement each other. Secondly, the **vote1** combinator, equivalent to a simple union of all alignments, is superior. While it can be assumed that **vote2** and **vote3** reduce the amount of erroneous alignments, they inevitably also increase the sparsity of alignments and are unable to compensate for the resulting loss in coverage.

When interpreting the effect of the different **vote**k combinators, it is important to consider that the lexical induction algorithm is fundamentally equipped to deal with conflicting alignments. As is explained in Section 5.1, when a split is performed and a node has alignments to tokens on both sides of the split, both edges are deleted. This mechanism appears to be an effective way of dealing with noisy alignments in a local manner, in contrast to the global approach of filtering in advance.

Since the coverages for all combinations of three aligners are lower than the all-aligners condition, we can infer that each of the aligners brings an improvement, even if it is small. Interestingly, while the amr_ud aligner performs very weakly on its own due to the sparsity of its alignments on the token level, it is part of the strongest-performing combinations, suggesting that the alignments it contributes are of high quality. Overall, the all-aligner conditions shows the strongest performance and is thus chosen for all subsequent experiments.

5.3.5. Comparing Grammars

Apart from the full version **all** of GA-CCG, which has been motivated through examples in Section 4.2, alternative rule sets have also been defined: the simplified grammars **no-ignore**, **no-fa-modify**, **no-tr**, and **base**, as well as the unrestricted grammars **unrestricted-all** and **unrestricted-a**. The simplified grammars omit rules for certain linguistic phenomena, allowing us to study their statistical impact on token coverage. The unrestricted grammars do away with the principle of pairing

Condition	Token Cov.	Lexicon Size	Runtime (s)
all	0.6623	538,640	2,082
no-ignore	0.6396	520,232	1,981
no-modify	0.6405	498,578	1,693
no-tr	0.6604	532,824	2,057
base	0.6157	467,504	1,587
unrestricted-all	0.7078	1,261,980	18,301
unrestricted-a	0.6451	640,295	4,677

Table 5.4.: Token coverages achieved using various GA-CCG grammars.

CCG combinators with a specific semantic interpretation and instead allow applying semantic operators regardless of the syntactic context.

Table 5.4 shows the impact of the various grammars on the token-level coverage, lexicon size, and runtime. Reassuringly, the **all** grammar achieves a better token-level coverage than all of the simplified grammars. This indicates that all specialised rules do contribute to coverage. While the loss in coverage for each of the simplified grammars appears small, a comparison with the **base** grammar (from which all specialised rules are omitted) shows that the specialised rules cumulatively contribute almost five percentage points in coverage.

As expected, the coverage achieved by the **unrestricted-all** grammar surpasses those of specialised grammars, since the number of rules available at any given step is larger. The lexicon size suggests, however, that this comes at the cost of a large increase in ambiguity and noise in the lexicon, as well as an almost tenfold increase in runtime.

Interestingly, **unrestricted-a** achieves a similar coverage to **all**, making it a potential candidate for a simpler alternative to the elaborate **all** grammar. However, **all** is more efficient in terms of lexicon size and runtime. Overall, this is a tenuous confirmation of the intuition that a linguistically tailored grammar allows greater efficiency and may be better suited to downstream processing.

5.3.6. Measuring the Impact of n-Best Parsing

Since semantic construction in GA-CCG is tightly linked to syntactic structure, GA-CCG depends on the shared structure between both types of representation. Apart from the fundamental question of whether this parallelism is always given, which

n	Token Cov.	Lexicon Size	Runtime
1	0.4724	346,550	266
2	0.5383	407,040	458
5	0.6171	479,435	1,076
10	0.6623	538,640	2,088
20	0.6976	583,985	3,973
50	0.7401	664,527	10,488

Table 5.5.: Token coverages, lexical item counts, and runtime achieved by covering a varying number of n-best syntax derivations.

is explored in Section 5.4, it may also be violated if erroneous syntactic parses are supplied by the external parser.

As described in Section 5.2, for every sentence, we use the the syntax parser to generate the n top-scoring derivations and select a single derivation which achieves the highest token-level coverage. This is the derivation which matches the semantic structure best. The more derivations are considered, the higher the chance is of finding such a high-quality derivation; however, since the derivations are ranked by the score assigned to them by the parser's model, the chance of each individual derivation being high-quality decreases along with its rank.

At the same time, n cannot be increased limitlessly because of the computational cost incurred both due to running lexicon induction n times for each sentence, and the computational requirements of syntax parsing itself. In practice, we found that time and memory requirements of the EasyCCG parser made values of $n > 50$ difficult to handle.

Table 5.5 shows the lexicon induction results obtained with several values of n. As expected, higher settings of n are associated with higher token-level coverage. The moderate increase in lexicon size can be attributed to the improved coverage, while the increase in runtime directly reflects the increased number of derivations that need to be processed.

It is somewhat surprising that even at $n > 20$, additional parses still provide a large improvement to token-level coverage. Apparently, high-quality derivations seem to exist relatively far from the top of the ranking, suggesting that the ranking of derivations produced by the parser is not very reliable. Considering that EasyCCG's model considers only probabilities on the supertag level but does not directly score dependencies or constituents, we hypothesize that EasyCCG is weak

Max. Coreferences	Token Cov.	Lexicon Size	Runtime
0	0.4976	276,030	549
1	0.6623	538,640	2,071
2	0.6785	671,180	8,326

Table 5.6.: Token coverages, lexical item counts, and runtime achieved by allowing various numbers of coreferences per splitting step.

in making attachment decisions that are crucial for the induction algorithm to work. For instance, an incorrectly attached prepositional phrase may prevent the induction algorithm from generating lexical entries for a large chunk of the sentence, but EasyCCG's model does not always contain sufficient information about such attachment decisions.

While it would be interesting to compare EasyCCG's behaviour with that of other CCG parsers with richer models, we view the token-level coverage achieved at $n = 50$ as satisfactory. In Chapter 6, we describe how the lexical items produced in this step can be delexicalised so that the resulting generalised lexicon is able to cover a much larger section of the corpus.

5.3.7. Evaluating the Need for Coreference Nodes

Many AMRs contain edges between referents that are not directly syntactically related. For instance, this occurs if a pronoun references another noun in the same sentence and thus allows the corresponding entity to relate to several verbs at once. Such phenomena are accounted for by a coreference mechanism which allows additional nodes to be unmerged during a split (see Section 5.2 for details).

Intuitively, cases where more than one coreference needs to be unmerged in a single splitting step should be rare. At the same time, coreference extraction introduces a risk of adding noisy entries to the lexicon, as it may sometimes act as a "fallback" mechanism which allows induction to continue even in cases where there is a mismatch between the meaning representation and the syntax derivation or the alignments. In these situations, it is likely that the resulting lexical items will not generalise well.

We examine this tradeoff experimentally by running induction with 0, 1, or 2 coreferences allowed per splitting step. The results are presented in Table 5.6 and confirm the above intuition: While allowing a single coreference per step is

crucial for achieving a good token-level coverage, a second coreference provides little additional coverage and is expensive in terms of runtime. At the same time, the amount of additional lexical items points to a considerable amount of noise introduced to the lexicon.

5.4. Experiments on Grammar Coverage

While GA-CCG has been motivated on a theoretical level in Chapter 4, and the experiments in the previous section have given some indication on the coverage of GA-CCG on a wide-coverage corpus, it is important to understand in more detail how well the rules of GA-CCG match the meaning representations found in the AMR corpus, and what effect the additional errors have that are introduced by the aligners and the syntactic parser. Therefore, in this section, we conduct an additional experiment on a small subset of AMR data for which higher-quality annotations are available, and perform error analyses on the derivations generated by the induction algorithm.

5.4.1. Corpus

Syntactic derivations and token-to-node alignments are a source of noise when they are generated by external tools. Fortunately, there exists a sub-section of the AMR 1.0 corpus for which gold-standard annotations of both syntax and alignments are available. These sentences are contained within the `consensus-dev` and `consensus-test` sections of the corpus. In this section, we evaluate lexicon induction on the 100 sentences of the `consensus-dev` section.

5.4.2. Annotations

In this section, we evaluate the performance of the lexicon induction algorithm when either gold-standard or tool-derived annotations are used. The following annotation sources are used in the various experimental conditions.

Gold-Standard Data

The sentences in the `consensus-dev` corpus are part of the Wall Street Journal corpus of the Penn Treebank (Marcus, Santorini and Marcinkiewicz 1993) and, consequently, CCGBank (Hockenmaier and Steedman 2007). Syntactic CCG derivations can therefore be taken directly from CCGBank.

Gold standard token-to-meaning alignments have been made available by Pourdamghani et al. (2014)[6]. They have been used for evaluating the ISI aligner presented in the same paper and contain both token-to-node and token-to-edge alignments. Our algorithm disregards token-to-edge alignments and uses only the token-to-node alignments.

Annotation Tools

For tool-derived annotations, we use the settings validated by the large-scale induction experiments in Section 5.3. For syntax derivations, the ten highest-scoring derivations from the EasyCCG parser are used. Alignments are obtained by combining the outputs from the JAMR, ISI, amr_ud, and TAMR aligners using the *vote-1* strategy.

5.4.3. Additional Rules for Induction from CCGBank Syntax

This is the only experiment in this thesis which uses CCGBank annotations for lexicon induction. All other experiments use syntactic derivations produced by the statistical parser EasyCCG (Lewis and Steedman 2014). While EasyCCG is trained on CCGBank, it produces a subset of the combinators contained within CCGBank derivations. In fact, CCGBank contains a large number of non-standard combinators which are not contained within usual introductions to CCG such as Steedman (2000).

For these combinators to be processed by the lexicon induction algorithm, additional rules are required. They were assigned a semantic operator according to their function in the CCGBank derivations. A list of these rules is given in Table 5.7. As they are only required for this single experiment on a limited corpus, only the rules required for dealing with the `consensus-dev` corpus are included.

5.4.4. Error Analysis Methodology

We conduct error analyses on the output of the lexical induction algorithm to qualitatively assess shortcomings of the proposed grammar and algorithms. Error analysis is conducted on the basis of *induction stoppages*: combinatory nodes which have been assigned one or several meaning representations but whose children have not. Every such stoppage is assigned an error class as described below.

[6]The alignments are available for download at `https://isi.edu/~damghani/papers/gold_alignments.zip` (retrieved 19 Nov 2021).

Syntactic Rule	Semantic Operator
$X \quad Y \Rightarrow Y[\text{conj}] \quad (X \in \{,, ;, :, ., \text{conj}\})$	**S**, **A**, **Ki**
$X \quad X \Rightarrow X$	$\mathbf{A}^{\leftarrow}$, **S**
S\S $\quad , \Rightarrow$ S/S	$\mathbf{Ki}^{\leftarrow}$
NP $\quad , \Rightarrow$ S/S	$\mathbf{A}^{\leftarrow}$, $\mathbf{Ki}^{\leftarrow}$
conj $\quad$ S $\Rightarrow$ NP[conj])	**S**

Table 5.7.: Additional grammar rules used for processing CCGbank derivations. On the left are syntactic rules that have been observed in CCGbank derivations. On the right are possible semantic interpretations of these rules. In some cases, more than one semantic interpretation can be inferred.

According to this procedure, all sentences with a token coverage of less than 100 % are examined, and more than one error per sentence may be detected.

Error Classes

To simplify interpretation of the error analysis, error classes are organised in a hierarchy. The first important question we ask of any error is: Is the error the result of a weakness in the lexical induction system that could in theory be solved (for instance, by introducing additional rules to cover a specific case)? In this case, the error is due to a *grammar limitation*. Or does the phrase in question exhibit a structural divergence between syntax and semantics, which is at odds with the strong assumption of compositionality underlying our grammar? Then, the error is due to a *syntax-semantics divergence.*

We distinguish the following types of *grammar limitation*:

- Deep Modification: the function graph attaches to the argument graph at a level below the root, but the **M** operator is not applicable according to the grammar rules. For instance, this is the case if the attachment is two levels below the root, or if there are multiple nodes on level 1 (see Example 5.7).

- Disconnected Sub-Graph: the argument graph would become disconnected, which prohibits the split.

- Focus Shift: a type-changing operation shifts the focus to a different node, which is not modelled in the grammar, as explained in Example 4.10.
- Ignore Operator: the argument is semantically empty but the ignore operator is not applicable according to the grammar rules.
- Overlapping Edges: an edge is shared between the function and argument graphs, which cannot be analyzed using the inventory of semantic operators in GA-CCG (see Figure 5.8). However, relation-wise operators could be introduced to address such phenomena (Blodgett and Schneider 2019).

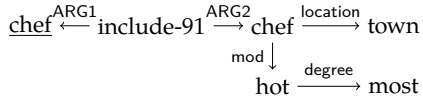
Furthermore, we identify the following types of *syntax-semantics divergence*:

- Dependencies: semantic dependencies are established to an entity that is not the focus of the phrase. For instance, modifiers sometimes refer to the noun of a phrase instead of the head verb (see Figure 5.6). In other cases, idiomatic expressions are represented in AMR in a non-compositional way (see Figure 5.5).
- Non-Monotonicity: AMR representations for several linguistic constructions are inherently non-monotonic as they involve the copying of nodes which are represented only once in the sentence. Partitives are an example thereof, as are some instances of quantification (see Figure 5.4).
- Annotation: errors in the annotated CCG derivations, meaning representations, or alignments can cause stoppages. We assign this error class only in cases where there clearly is a more correct structure that could be annotated. In particular, CCGBank contains systematic errors, for example in noun phrase bracketing (Honnibal, Curran and Bos 2010). See Figure 5.9 for an example.
- Ambiguity: Linguistic ambiguities such as attachment ambiguities may cause divergence as semantic and syntactic annotations are produced separately from each other and may therefore represent different readings of the sentence.

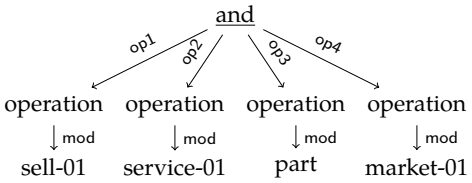
5.4.5. Results and Interpretation

We run lexicon induction with four combinations of annotations:

- **gold**: CCGBank syntax derivations with gold-standard alignments



(a) Partitives can cause nodes to be duplicated, such as the *chef* nodes in this AMR. The phrase represented in the example is *some of the hottest chefs in town*. From `wsj_0010.17`.



(b) Example for the duplication of nodes due to coordination. The phrase is *sales, service, parts and marketing operations*. Even though the token *operations* occurs only once, a separate *operation* node is introduced for each operand of the *and* conjunction. From `wsj_0009.2`.

Figure 5.4.: Examples for non-monotonic constructions.

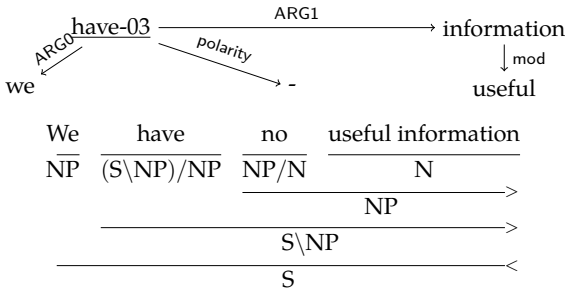


Figure 5.5.: Example for divergent dependencies. The polarity edge of *have-03* does not match the dependency between *no* and *information*. Simplified from `wsj_0003.9`

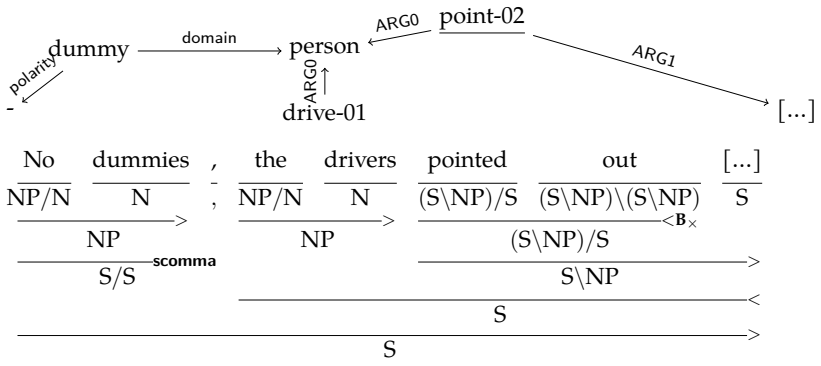


Figure 5.6.: Example for divergent dependencies. Although *point-02* is the focus of the main clause, the modifier *no dummies* attaches to the subject. Simplified from `wsj_0010.14`

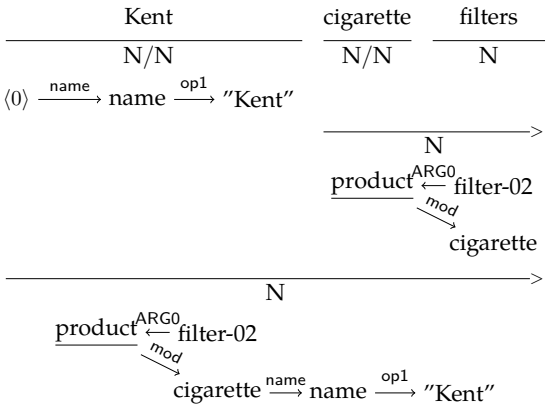
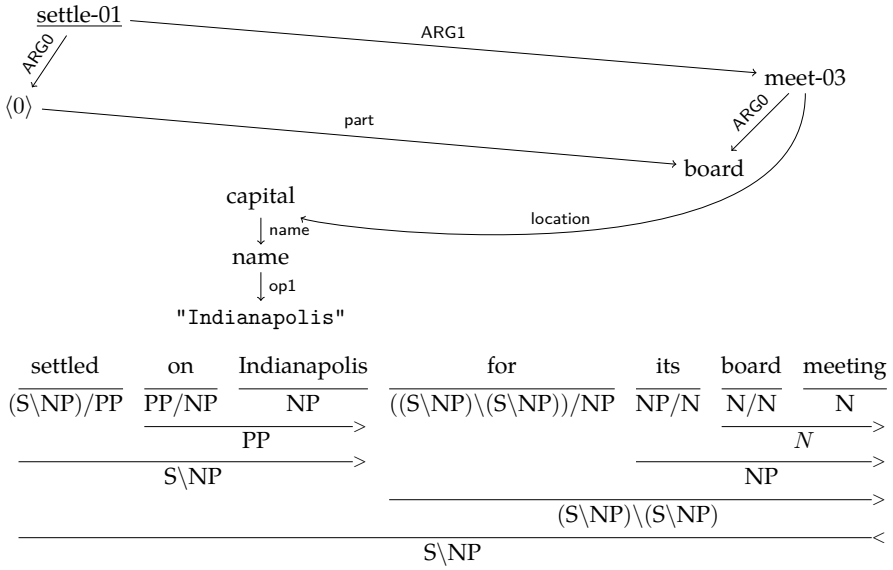
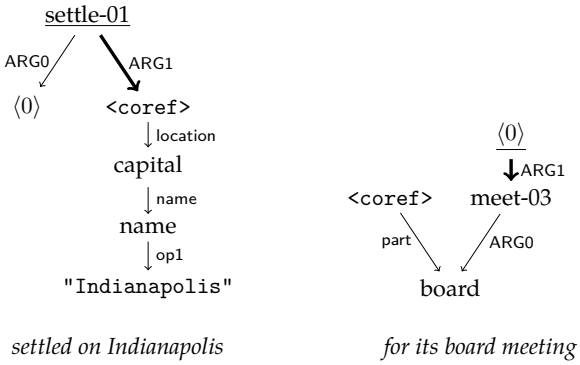


Figure 5.7.: Example for a non-unique depth-1 attachment. The *Kent* modifier cannot be attached to *cigarette* because of the sister node *filter-02*. Simplified from `wsj_0003.1`



(a) The semantic representation shown above cannot be decomposed into representations for constituents *settled on Indianapolis* and *for its board meeting* because the *location* edge is attached to the constituent on the right side.



(b) Proposed semantic representations for the sub-constituents of the phrase. The highlighted ARG1 edge appears in both representations, creating a link between the *location* edge and *settle-01*.

Figure 5.8.: Illustration of edge overlap. Simplified from `wsj_0010.3`.

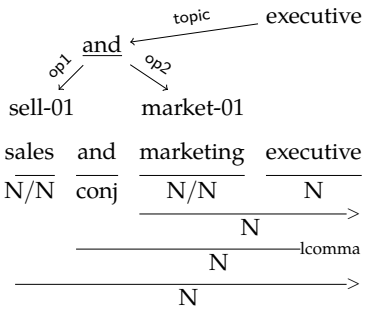


Figure 5.9.: Example for a syntactic annotation error. The semantic annotation correctly represents the conjunction of the modifiers *sales* and *marketing*. In the syntax, the *lcomma* combinator is annotated instead, effectively treating the conjunction as punctuation. As a result, *marketing executive* forms a syntactic constituent, but not a contiguous subgraph of the AMR. From `wsj_0009.4`.

	gold	**parser**	**aligner**
Syntax/Semantics Divergence	**30**	**39**	**33**
Dependencies	16	5	13
Non-Monotonicity	4	2	5
Annotation	8	31	13
semantic	1	1	1
syntactic	7	30	3
alignment	0	0	9
Ambiguity	2	1	2
Grammar Limitations	**19**	**8**	**15**
Deep Modification	11	4	8
Disconnected Sub-Graph	3	2	2
Focus Shift	2	1	2
Ignore Operator	2	1	2
Overlapping Edges	1	0	1
Overall	**49**	**47**	**48**

Table 5.8.: Error counts for grammar coverage experimental conditions.

Condition	Token Coverage	Perfect Sentences
gold	0.7360	55
parser	0.7298	58
aligner	0.6909	58
both	0.6825	60

Table 5.9.: Statistics for grammar coverage experimental conditions. The corpus size is 100 sentences.

- **parser**: EasyCCG syntax derivations with gold-standard alignments
- **aligner**: CCGBank syntax derivations with automatic alignments
- **both**: EasyCCG syntax derivations with automatic alignments

For the first three conditions, a manual error analysis was performed with the counts shown in Table 5.8. For all four conditions, the total token coverage and the number of "perfect" sentences with 100 % token coverage are measured and shown in Table 5.9. In the remainder of this section, the error analysis is discussed in detail and an interpretation of the results is given.

Gold Condition

To remove noise generated by external tools, the **gold** condition is conducted using gold-standard syntax and alignments. Under these conditions, an average token coverage of 73.6% is achieved. Only 55 of the 100 sentences have a perfect token coverage of 100% (see Table 5.9).

This result confirms that the grammar is very sensitive to the various types of syntax/semantics divergence, which make up 30 of 49 diagnosed errors (Table 5.8). In particular, the 16 errors in the *Dependencies* class show that semantic and syntactic dependencies diverge frequently in a way that prevents complete lexical induction.

Annotation errors are also an issue, although it is difficult in some cases to distinguish between faulty annotations (in the *annotation* class) and conscious annotation decisions (in the *dependencies* class). In many cases, divergences from the syntactic dependency structure are clearly motivated; this is illustrated in Figure 5.5, where the goal of *abstracting away from syntactic idiosyncracies* mandates that the negation be applied to the verb instead of the noun *information* (Banarescu et al. 2013). In other cases, the annotated AMR could be interpreted using an alternative syntactic

derivation, even if such an analysis might be considered unconventional. In such cases, the *dependencies* class was always annotated, since it was difficult to judge the idiomaticity of the various possible syntactic derivations and AMRs.

The *non-monotonicity* and *ambiguity* classes represent issues that are fundamentally challenging for the GA-CCG framework. Non-monotonicities such as those shown in Figure 5.4 cannot be built using compositional means and would therefore require the introduction of additional rules which are challenging to formulate. Ambiguity errors are an inevitable consequence of using independent sources of syntactic and semantic annotation and can only be eliminated by creating both representations in a single process to ensure that the same reading of a sentence is represented in both syntactic and semantic annotations. Again, there is possible confusion with the *dependencies* class, as the examples from that class are distributed across various levels of acceptability. Again, *dependencies* was always annotated in favour of *ambiguity* when there was any doubt about the acceptability of one of the possible representations.

Among errors related to limitations of the grammar, *deep modification* is the largest error class. These errors are mostly due to the fact that single words may invoke several concepts and can thus be represented by multiple nodes. While there is a grammar rule which allows modifiers to attach to nodes one level below the root, it only applies if there is only one such node. Since there are many contexts where this condition is violated, a more flexible strategy for the attachment of modifiers might be needed. Since such a modification to the grammar would lead to a large number of spurious lexical items and place a further burden on a semantic parser employing the grammar, we do not further pursue this option in this thesis.

The remaining error classes are small, showing that while parser behaviour might be improved by extending the grammar to deal with the phenomena in question, this is unlikely to have a big statistical effect on a parser's performance.

Parser Condition

The biggest shift in the error counts when comparing the **gold** condition to the **parser** condition is a shift of errors to the *annotation* class from all other classes. This is expected when comparing the output of a statistical parser to gold-standard data, and seem to indicate that other errors are masked by syntax errors produced by the parser.

In fact, the picture is slightly more complex. In fourteen sentences, an error from the **gold** condition was masked by the introduction of a syntax error produced by the parser. On the other hand, the overall error count between both conditions is

similar. This shows that the syntax parser introduces new errors, but also fixes some derivations that are problematic in the CCGbank annotations. This can be achieved by improving upon erroneous annotations or producing less conventional syntax derivations which better match the AMR's structure.

The results of the large-scale lexicon induction experiment in Section 5.3 show that the selection of the n-best parses has a large influence on the token coverage that is achieved. Selecting from ten parses causes improvements to nearly cancel out the additional errors, leading to similar total error counts and token-level coverage.

Aligner Condition

In contrast to CCGBank derivations, we did not detect any errors in the gold-standard alignments. There is therefore less opportunity in the **aligner** condition to improve upon the performance of the **gold** condition. However, the tool-generated alignments can be sparser than the gold-standard annotation, which increases the flexibility of the lexical induction algorithm and may allow additional, possibly noisy lexical items to be induced. Overspecified alignments can have the same effect, as they can cause the lexical induction algorithm to ignore certain alignment edges as explained in Section 5.2. We observed the induction of noisy lexical items due to the tool-generated alignments in nine sentences[7].

As with the **parser** condition, the total error counts are very similar to those of the **gold** condition, which indicates that sentences with improvements cancel out those with new errors. Still, these improvements have been found to increase the amount of noisy lexical items, and the diminished token coverage shows that alignment errors tend to cause problems early in the induction process, preventing lexical induction for large parts of the sentence.

5.4.6. Discussion

The analysis shows that the proposed grammar fits the corpus rather roughly: While almost three quarters of tokens are covered under ideal conditions, almost half of all sentences contain at least one construction that cannot be analysed. The performance gap between gold-standard and tool-generated annotations is small, which is encouraging as gold-standard annotations are not available in our large-scale experimental setting. The achieved token coverage is also comparable to the results of the large-scale experiment described in Section 5.3.

[7] As we only count errors that lead to an induction stoppage, these are distinct from the nine sentences in which alignment errors are diagnosed.

Perfect coverage is not a requirement for successful lexicon induction. As the ultimate aim is to utilise the induced lexical items for parsing novel sentences, it should suffice to achieve a good coverage of high-frequency tokens and categories across the induction corpus. That is, even if a lexical item for a given token cannot be derived from a specific sentence in the corpus, there is likely to be another sentence in the corpus containing that token. In Chapter 6, we also discuss how lexical items can be delexicalised to generalise across words of the same syntactic category, which further reduces the dependence of lexicon quality on the coverage of individual tokens.

Tool-generated annotations have been observed to lead to the creation of noisy lexical items with little potential for generalisation. While the amount of noise in the induced lexicon has not been quantified in this experiment, token coverage figures need to be interpreted with caution. The quality of lexical items, as well as noise reduction, are addressed in more depth in Chapter 6.

The issues discussed in this chapter also largely apply to parsing: certain constructions found in the AMR corpus cannot be derived using GA-CCG. Again, perfect treatment of such constructions is not necessarily needed to produce a strongly performing parser. Since Smatch scores[8] are calculated on the edge level, individual mis-attachments do not impact the Smatch score strongly. In Chapter 7, the impact of the grammar on parser performance is examined in detail.

[8] See Section 2.1.5.

Algorithm 5.1 Syntax-Driven Splitting Algorithm

Inputs:

- A derivation d
- An AMR graph g
- A GA-CCG rule set (R_1, R_2)

Output: A set of lexical items

function SplitSyn(d, g, R)
  if $|\textsc{children}(d)| = 0$ **then** ▷ At a leaf of the derivation
    return $\{(\textsc{tok}(d), \textsc{syn}(d), \textsc{sem}(d))\}$ ▷ Generate a lexical item
  else if $|\textsc{children}(d)| = 1$ **then** ▷ At a unary branching node
    $d' \leftarrow \textsc{children}(d)[0]$ ▷ Move one level down
    return SplitSyn(d', g, R) ▷ Do not split the graph
  else ▷ At a binary branching node
    $O \leftarrow \{\}$ ▷ Initialise output set
    $(d'_1, d'_2) \leftarrow \textsc{children}(d)$
    for all $(C, p, o) \in R_2$ **do** ▷ For every rule
      if $C = \textsc{cmb}(d) \wedge p(\textsc{syn}(d'_1), \textsc{syn}(d'_2))$ **then** ▷ If the rule is applicable
        for all $(g'_1, g'_2) \in$ SplitSem(g, o) **do** ▷ Split the graph
          $O \leftarrow O \cup$ SplitSyn(d'_1, g'_1, R) ▷ Move to the child nodes
          $O \leftarrow O \cup$ SplitSyn(d'_2, g'_2, R)
        end for
      end if
    end for
    if $O \neq \{\}$ **then**
      return O ▷ Return all lexical items
    else
      return $\{(\textsc{tok}(d), \textsc{syn}(d), \textsc{sem}(d))\}$ ▷ Generate phrasal item
    end if
  end if
end function

Algorithm 5.2 Constrained graph splitting algorithm

Inputs:

- GA-CCG s*-graph $g = (V, E, v_{\text{root}}, l, \textit{slab})$
- Semantic operator o

Output: A set of pairs of GA-CCG s*-graphs $O = \{(g_1^1, g_2^1), (g_1^2, g_2^2), \dots\}$
Description: Generates predecessors for g, assuming that g is the result of applying o to the predecessors. Thus, for each generated pair g_1^i, g_2^i, it holds that $g = o(g_1^i, g_2^i)$.

function SplitSem(g, o)
    $O \leftarrow \{\}$ ▷ Initialise output set
    $m \leftarrow$ InitialLabelling(g) ▷ Assign colours from alignments
    for all $m' \in$ Completions(m, V) **do** ▷ Colour remaining vertices
        $O \leftarrow O \cup o^{-1}(g, m')$ ▷ Split the graph
    end for
    return O
end function

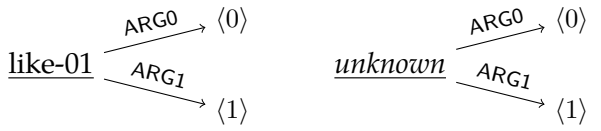
Chapter 6.

Post-Processing the Lexicon: Delexicalisation, Filtering, and Supertagging

In the preceding chapter, algorithms to induce a syntactic-semantic lexicon were discussed, with the goal of using the induced lexicon to parse novel sentences. It was shown that this can be achieved with an acceptable coverage and efficiency. However, the lexicon induced in Chapter 5 suffers from a crucial weakness: Since lexical entries are tied to surface tokens, words that do not occur in the training data cannot be interpreted.

For instance, the lexicon does not generalise from the induced meaning of the transitive verb *like* to the unseen word *love*, even though both words share the same syntactic category and would intuitively be expected to be exchangeable on the semantic level.

Assume that the induced lexicon contains a lexical entry for the word *like* as shown in Figure 6.1a. When a parser is using this lexicon and is presented with the unseen transitive verb *love*, we would at least expect it to generate an appropriate semantic structure, independently of its concrete lexical content. Even if the concrete concept that corresponds to *love* has not been learned, the instantiation of a generic



(a) Meaning representation for *like*.

(b) Generic template for transitive verbs.

Figure 6.1.: A lexicalised and a delexicalised lexical entry.

transitive verb structure, as shown in Figure 6.1b, would help the parser form an analysis of the sentence. It could even guess that *love-01* might be an appropriate concept to associate with the unknown node.

We call the process by which lexical content is removed from a lexical item *delexicalisation* and describe it in detail in Section 6.1. Besides allowing effective generalisation, delexicalisation also maps each class of structurally identical lexical items to a single template, which reduces the amount of redundancy in the lexicon and potentially improves parser training by allowing the model to score lexical items on a structural level, abstracting away from concrete node labels.

On the other hand, delexicalisation creates a large amount of noise because incorrect delexicalisations are performed where non-lexical content is erroneously removed. This adds to the superfluous lexical items that are already present in the induced lexicon, for instance due to sparse alignments. It would be helpful to have an indication of which lexical items are plausible and likely to generalise, and which are likely to be noise.

Expectation Maximisation (EM) is a class of algorithms which estimate a probability distribution in an unsupervised manner, and in Section 6.2 we derive an EM algorithm from the well-known Inside-Outside algorithm (see Section 3.4) which scores lexical items and allows us to filter out those with a low probability.

The result of EM filtering is a greatly reduced lexicon which nonetheless retains the same token coverage of the training corpus as the original lexicon. As its entries are delexicalised, its generalisation is improved. However, a downstream parser now faces a more difficult problem when selecting lexical items, as every delexicalised item is available for every token. While the EM algorithm has associated lexical items with probabilities, these do not take the concrete sentence context into account and are therefore of limited utility for parsing.

Using EM filtering the number of delexicalised templates can be reduced enough to permit supertagging – that is, directly predicting the templates to be instantiated for every token using a BiLSTM neural network (see Section 6.3). This provides the downstream parser with a ranked list of options for every token, greatly reducing its search space. The pipeline of delexicalisation, EM filtering, and supertagging therefore transforms the induced lexicon into input suitable for the CKY-based parser presented in the following chapter.

6.1. Delexicalisation

Lexical entries that are tied to a specific surface form generalise poorly. This causes problems when they are applied to wide-coverage corpora such as the newswire texts contained in the AMR corpus. At the same time, many lexical items share an identical structure that differs only in node labels, as shown in Figure 6.1.

One approach to this issue in CCG parsing is to separate lexical entries into *lexemes* and *lexical templates* (Kwiatkowski et al. 2011). Lexical templates provide a "delexicalised" abstraction of the semantic content of a word. This abstraction is achieved by replacing logical constants with special placeholders, so that the structure of the meaning representation is retained without the lexical content. A lexeme can be used to instantiate a template for a specific word by re-introducing the lexical content: the lexeme contains a word and a list of logical constants[1], which are inserted into the delexicalised placeholders of the template. A template and a lexeme can thus be recombined into a full lexical entry.

We adapt this approach to AMR parsing. Instead of logical constants, our lexemes contain concepts, constants or relations. In templates, these labels are replaced with special placeholders. A *delexicalised lexicon* contains both lexemes and templates. To simplify the delexicalisation of the lexicon, we furthermore limit lexemes to contain one token and one label only, and likewise, templates to contain only one lexicalisation placeholder. To make delexicalisation optional, the original graph is always added as a template which can be combined with the lexeme (t, ϵ).

Definition 6.1 (Delexicalisation)**.** Let $x = (w, c_{\text{syn}}, c_{\text{sem}})$ be a CCG lexical entry where $c_{\text{sem}} = (V, E, v_{\text{root}}, l, \mathit{slab})$ is a GA-CCG s*-graph.

The *delexicalised set* of x is defined as follows:

$$\begin{aligned}\textsc{Delex}(w, c_{\text{syn}}, c_{\text{sem}}) = &\{((w, \epsilon), (c_{\text{syn}}, c_{\text{sem}}))\} \\ &\cup \{(\textsc{lex}(e), \textsc{tmpl}(e)) : e \in V \cup E\}\end{aligned}$$

where

$$\begin{aligned}\textsc{lex}(e) &= (w, \textsc{label}(e)) \\ \textsc{tmpl}(e) &= (c_{\text{syn}}, c'_{\text{sem}}) \quad \text{where } c'_{\text{sem}} = (V, E, v_{\text{root}}, l_{e \to \texttt{<lex>}}, \mathit{slab})\end{aligned}$$

[1] Like a lexeme in morphology, such a lexeme can represent various different forms of a word, even if they correspond to different templates – hence the name.

$\mathcal{L}$	$\mathcal{T}$
(join,)	join-01 $\xrightarrow{\text{ARG0}}$ $\langle 0 \rangle$; join-01 $\xrightarrow{\text{ARG1}}$ $\langle 1 \rangle$
(join, join-01)	<lex> $\xrightarrow{\text{ARG0}}$ $\langle 0 \rangle$; <lex> $\xrightarrow{\text{ARG1}}$ $\langle 1 \rangle$
(join, ARG0)	join-01 $\xrightarrow{\text{<lex>}}$ $\langle 0 \rangle$; join-01 $\xrightarrow{\text{ARG1}}$ $\langle 1 \rangle$
(join, ARG1)	join-01 $\xrightarrow{\text{ARG0}}$ $\langle 0 \rangle$; join-01 $\xrightarrow{\text{<lex>}}$ $\langle 1 \rangle$

Table 6.1.: Delexicalised meaning representations derived from the word *join*.

Example 6.2 (Delexicalisation)**.** Consider the following lexical entry[2]:

$$\text{join} := ((\text{S}\backslash\text{NP})/\text{PP})/\text{NP} : \underline{\text{join-01}} \xrightarrow{\text{ARG0}} \langle 0 \rangle;\ \underline{\text{join-01}} \xrightarrow{\text{ARG1}} \langle 1 \rangle$$

Four lexeme-template pairs can be derived from this entry. They are shown in Table 6.1.

6.1.1. Lexeme Patterns

Many lexemes take one of several predictable forms. To aid generalisation, these can in turn be abstracted into a number of patterns. Given a token w, we generate lexemes on the fly using the following rules:

- $(w, \mathit{lemma}(w))$: many concepts are lemmas of the token they represent

[2]Induced from sentence `nw.wsj_0001.1`.

Parameter	Value
grammar	all
alignments	jamr+tamr+amr_ud+isi-vote1
maxCorefs	1
derivations	50
maxUnalignedNodes	13
maxItemCount	10000

Table 6.2.: Parameter settings for the validation run of the delexicalisation algorithm.

- $(w, \texttt{"}w\texttt{"})$: quoted constants
- (w, f) where f is an OntoNotes frame associated with the lemma of w.

Since the lexeme dictionary contains distinct items, a lexeme is not added if it matches one of these patterns; this is important to consider when interpreting the lexeme count resulting from the validation experiment in the following section.

Another common pattern is (t, ϵ). This pattern is excluded because it causes problems during EM filtering (see Section 6.2), where it occurs so frequently that it drowns out all other lexemes and forces their probabilities to zero.

6.1.2. Validation Experiment

Given a GA-CCG lexicon, the above definition can easily be implemented as a brute-force algorithm. For every lexical entry, template-lexeme pairs are generated by relabelling every node or edge in turn.

To confirm the viability of the proposed algorithm, we conduct a large-scale induction experiment using the entire `proxy-train` section of the AMR 1.0 corpus. The settings are derived from the best-performing large-scale induction experiment in Section 5.3 and shown in Table 6.2.

Care must be taken when interpreting lexical item counts because they include phrasal items which span more than one token in cases where the algorithm has aborted. This is necessary to allow the EM algorithm to allocate probabilities. These lexical items are however not used for supertagging or parsing.

The resulting lexicon contains 690.527 lexical entries. The brute-force delexicalisation algorithm creates several templates per lexical entry on average, resulting in 4,844.324 distinct templates, and 511.436 distinct lexemes, of which 36,299 are empty

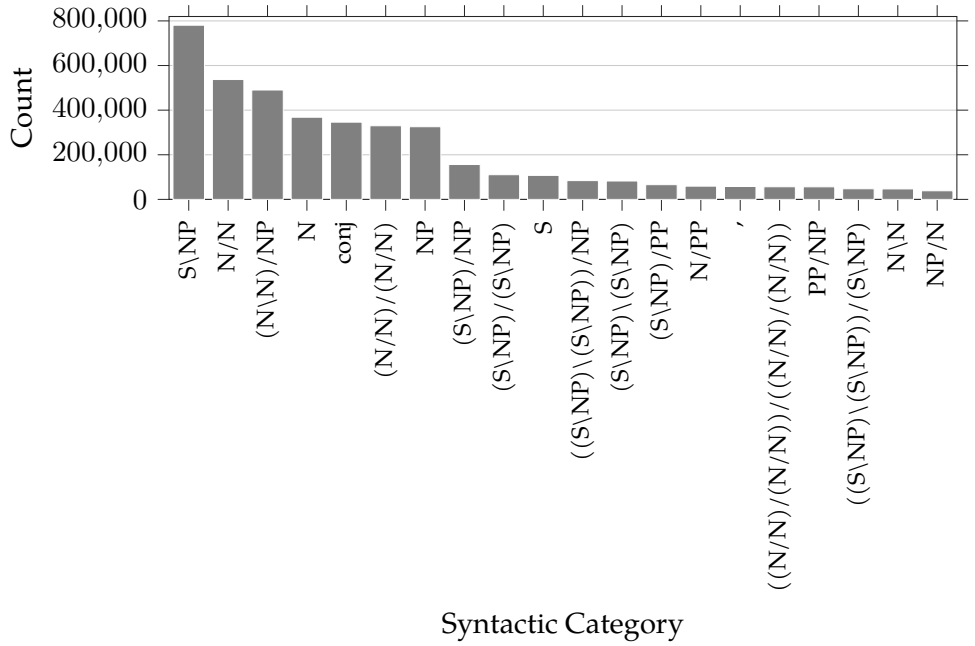


Figure 6.2.: Number of templates per syntactic category after delexicalisation. The top 20 out of 181 syntactic categories are shown.

lexemes of the form (t, ϵ). These very large counts are a result of the brute-force delexicalisation, which creates many implausible template-lexeme pairs.

On average, there are 14 lexemes per token (median 12), with a maximum of 917 lexemes for the token "and". Templates are distributed across 181 syntactic categories with an average of 26,764 templates per syntactic category (median 178). Template counts for the most common syntactic categories are visualized in Figure 6.2. These distributions illustrate the difficult task a parser faces when selecting lexical entries.

Due to the large number of templates and lexemes, the delexicalised lexicon is not directly suitable for parsing. In the next section, we tackle the task of filtering out unnecessary or implausible items.

6.2. Expectation Maximisation Filtering

Every step of the lexicon induction pipeline so far has added ambiguity. For every node of a syntax derivation, the lexicon induction algorithm may have produced one, several, or no semantic representations. Each of these may have been delexicalised into a number of template-lexeme pairs.

Some of the templates and lexemes are plausible because they are applicable to a number of concrete syntactic contexts. For example, the template for *like* shown in Figure 6.1b can be applied to other transitive verbs in combination with the proper lexeme. Other templates and lexemes are erroneously created due to the brute-force nature of delexicalisation, gappy alignments, or other causes. Lexemes such as $(\text{join}, \text{ARG0})$ from Example 6.2 fall into this category. Such templates and lexemes are unlikely to be useful in the analysis of novel sentences.

Intuitively, templates and lexemes that occur in the induction results for more than one sentence are more likely to generalise than those induced only once. One might therefore count the number of examples supporting a given template or lexeme to determine its utility. However, induction of superfluous items happens systematically, especially during delexicalisation; therefore, this simple heuristic is not sufficient. Instead, a score for generalisation should be computed for every template and lexeme. Then, for every sentence, only the highest-scoring templates and lexemes should be kept. If a template or lexeme is not among this set for any sentence, it can be pruned from the lexicon.

These scores can be computed using an expectation maximisation (EM) algorithm. Since EM maximises the overall probability of the dataset, it tends to concentrate probability mass on a small set of templates and lexemes which can be used many times over across the data set, and thus automatically identifies the most generalisable items. In the remainder of this section, we describe an algorithm derived from the inside-outside algorithm described in Section 3.4.

6.2.1. An EM Algorithm for Scoring Templates and Lexemes

In order to optimise a given grammar with EM, a generative process needs to be defined which describes probabilistically how a data set may be generated from the EM model step by step. For PCFGs, a top-down generative process can be defined by starting with the start symbol and then choosing production rules according to their probabilities. For CCG, this process is less obvious since the formalism is usually described as a bottom-up process, starting with the selection of lexical items. A top-down process for GA-CCG could be described as follows:

1. Choose a atomic syntactic category X for the entire sentence, such as S, NP, or N.
2. Either
   a) Choose a syntactic combinator and syntactic categories X_1, X_2 such that the combinator can be applied to produce the original category X. Then repeat step 2 for both X_1 and X_2.
   b) Choose a template t matching the syntactic category X with probability $q_{\mathcal{T}}(t)$, and a lexeme l with probability $q_{\mathcal{L}}$.

This process invokes the probability distributions $q_{\mathcal{T}}$ over templates and $q_{\mathcal{L}}$ over lexemes, which are the quantities we are interested in estimating. All other choices are not parameterised and thus assumed to be made according to a uniform distribution. While this model is simple, it has been shown that the lexical choices in a CCG derivation capture a large part of its information (Lewis and Steedman 2014).

The model described above can also be viewed as a generalisation of Algorithm 5.1. In fact, the recursive splitting algorithm explores the slice of the generative process which is conditioned on a concrete example consisting of a sentence, a meaning representation and a syntactic CCG derivation.

Split charts produced by the recursive splitting algorithm are therefore used in our modified inside-outside algorithm for GA-CCG (Algorithm 6.1). We assume that, apart from the lexical entries, Algorithm 5.1 also outputs a structure of split tuples of the form $e \to e_1, e_2$, where e, e_1, and e_2 are lexical or phrasal items (possibly spanning several tokens). Any time a binary rule is applied (lines 13 and 14), a split tuple is generated linking the parent lexical item e to its children e_1 and e_2.

These splits play the same role as rule productions in the PCFG inside-outside algorithm, Algorithm 3.6. In the PCFG algorithm, inside and outside probabilities relate to nonterminals occurring in a given position, and are transferred to other nonterminals by the grammar's production rules. In the GA-CCG algorithm, inside and outside probabilities are computed for lexical items, and are transferred along relationships in the split structure.

To provide access to the split structure, we define the following functions:

- Entries(*splits*) is the set of all split tuples referenced in any split.
- Entries(*splits*, i, j) with $1 \leq i \leq j < n$ is the set of all split tuples spanning tokens i through j.
- Roots(*splits*) is the set of split tuples spanning the entire sentence.

The term for inside scores (Algorithm 6.1 line 9) can be explained as follows: Either the lexical entry e is drawn directly from the lexicon with probability *score*(e). This score is zero if the e is not in the lexicon. Alternatively, e can be generated from children e_1 and e_2 if $e \rightarrow e_1, e_2$ is in the splits table. In this case, the probability is the product of both children's inside scores since it requires generating all of their respective lexical entries.

The term for the outside score of an entry e (line 14) is a sum over all splits in which e occurs as a left or right child, since each of these splits may have been responsible for producing e. For any specific split, the outside probability is the product of the inside probability of its sibling (which after all is generated at the same time as e) with the outside probability of the parent entry e'.

In the calculation of the counts (line 19f), lexemes and templates are weighted according to their outside probabilities. The inside probability is not included in this term because the generative process, as defined above, stops if a lexical entry is generated. Multi-token phrasal items may be generated, but in this case, no derivation exists below these items.

The EM algorithm for GA-CCG (Algorithm 6.2) is almost identical to that for PCFG (Algorithm 3.5). The main difference is that probability distributions for both lexemes and templates are computed.

6.2.2. Filtering Templates and Lexemes

The EM algorithm for GA-CCG produces probability distributions over templates and lexemes. These can be used to filter items from the lexicon which are not needed to explain the corpus. To avoid losing coverage during filtering, we keep all lexical items belonging to the most likely derivation for every given item, even if their individual probability is low. The procedure can be summarised as follows:

1. For every example, find the sequence of lexeme-template pairs which (a) can be derived from a root entry in the split structure of SplitSyn, and (b) has the highest probability among such sequences. Mark all of its templates and lexemes.

2. Remove all unmarked templates and lexemes from the lexicon.

The search for an optimal sequence of lexeme-template pairs can be implemented using a simple dynamic programming algorithm over the split structure. While rare lexical items might be required to describe some examples, this algorithm favours the use of frequent lexical items with good generalisation wherever possible.

Algorithm 6.1 Calculation of template and lexeme counts for a single example.

Inputs:

- Sets of templates $\mathcal{T}$ and lexemes $\mathcal{L}$
- An example x of length m
- Probability distributions $q_{\mathcal{T}} : \mathcal{T} \to [0, 1], q_{\mathcal{L}} : \mathcal{L} \to [0, 1]$

Output: A mapping $\mathit{count} : \mathcal{T} \cup \mathcal{L} \to \mathbb{R}^+$

function COUNTS($x, q_{\mathcal{T}}, q_{\mathcal{L}}$)
  $\mathit{splits} \leftarrow$ SPLITSYN(x)
  for all $e \in$ ENTRIES(splits) **do** ▷ Initialisation
    $\mathit{outside}(e) \leftarrow \begin{cases} 1 \text{ if } (e_i) \in \text{ROOTS}(\mathit{splits}) \\ 0 \text{ otherwise} \end{cases}$
    $\mathit{score}(e) \leftarrow \sum_{(l,t) \in \text{DELEX}(e)} q_{\mathcal{T}}(t) q_{\mathcal{L}}(l)$
  end for
  for $j \leftarrow 1 \ldots m, i \leftarrow j \ldots 1$ **do** ▷ Inside computation
    for $e \in$ ENTRIES(splits, i, j) **do**
      $\mathit{inside}(e) \leftarrow \mathit{score}(e) + \sum_{(e \to e_1, e_2) \in \mathit{splits}} \mathit{inside}(e_1)\, \mathit{inside}(e_2)$
    end for
  end for
  for $j \leftarrow m \ldots 1, i \leftarrow 1 \ldots j$ **do** ▷ Outside computation
    for all $e \in$ ENTRIES(splits, i, j) **do**
      $$\mathit{outside}(e) \leftarrow \sum_{(e' \to e, e_2) \in \mathit{splits}} \mathit{outside}(e')\, \mathit{inside}(e_2) + \sum_{(e' \to e_1, e) \in \mathit{splits}} \mathit{outside}(e')\, \mathit{inside}(e_1)$$
    end for
  end for
  $Z \leftarrow \sum_{e \in \text{ROOTS}(\mathit{splits})} \mathit{inside}(e)$
  for all $e \in$ ENTRIES(splits), $(l, t) \in$ DELEX(e) **do** ▷ Compute counts
    $\mathit{count}(l) \leftarrow \mathit{count}(l) + \frac{\mathit{score}(e)\, \mathit{outside}(e)}{Z}$
    $\mathit{count}(t) \leftarrow \mathit{count}(t) + \frac{\mathit{score}(e)\, \mathit{outside}(e)}{Z}$
  end for
end function

Algorithm 6.2 The EM algorithm for GA-CCG filtering.

Inputs:

- A set of templates $\mathcal{T}$
- A set of lexemes $\mathcal{L}$
- Corpus $X = (x_1 \dots x_n)$
- Iteration count T
- Initial probability distributions $q^0_{\mathcal{T}} : \mathcal{T} \to [0, 1], q^0_{\mathcal{L}} : \mathcal{L} \to [0, 1]$

Output: Probability distributions $q^T_{\mathcal{T}} : \mathcal{T} \to [0, 1], q^T_{\mathcal{L}} : \mathcal{L} \to [0, 1]$

for $k \leftarrow 1 \dots T$ **do**
  for all $l \in \mathcal{L}$ **do** ▷ Initialise counts with zero
    $\mathit{count}(l) \leftarrow 0$
  end for
  for all $t \in \mathcal{T}$ **do**
    $\mathit{count}(t) \leftarrow 0$
  end for
  for $i \leftarrow 1 \dots n$ **do** ▷ Expectation step: aggregate counts
    $c \leftarrow \textsc{Counts}(x_i, q^{k-1})$
    for all $l \in \mathcal{L}$ **do**
      $\mathit{count}(l) \leftarrow \mathit{count}(l) + c(l)$
    end for
    for all $t \in \mathcal{T}$ **do**
      $\mathit{count}(t) \leftarrow \mathit{count}(t) + c(t)$
    end for
  end for
  for all $l \in \mathcal{L}$ **do** ▷ Maximisation step: normalise counts
    $q^k_{\mathcal{L}}(l) \leftarrow \frac{\mathit{count}(l)}{\sum_{l' \in \mathcal{L}} \mathit{count}(l')}$
  end for
  for all $t \in \mathcal{T}$ **do**
    $q^k_{\mathcal{T}}(t) \leftarrow \frac{\mathit{count}(t)}{\sum_{t' \in \mathcal{T}} \mathit{count}(t')}$
  end for
end for
return $q^T_{\mathcal{L}}, q^T_{\mathcal{T}}$

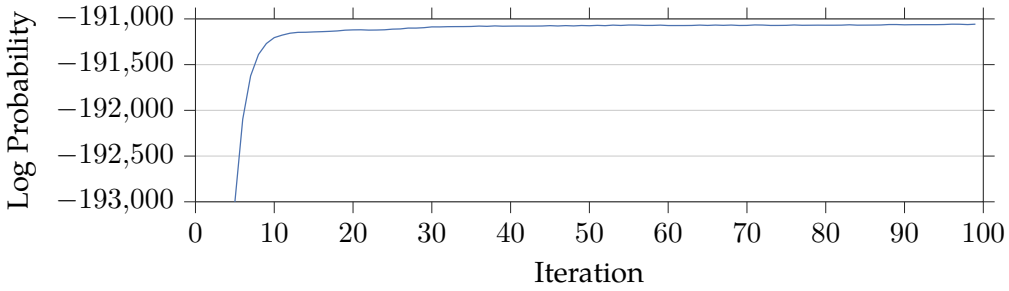


Figure 6.3.: Convergence of the EM algorithm over 100 iterations. The y axis shows the logarithm of the probability assigned to the entire training corpus.

6.2.3. Results

The primary goal of EM filtering is to reduce the burden of the large lexicon produced by delexicalisation. We run 100 EM iterations to filter the lexicon produced in Section 6.1.2. Figure 6.3 shows that the total probability of the training corpus has converged by this time. The filtering procedure is effective in reducing the number of lexical items: the number of templates is reduced from 4,844,324 to 15,498 and the number of single-token lexemes from 76,533 to 6,389.

An advantage of EM filtering is that it not only reduces the numbers of templates and lexemes that need to be stored, but also assigns them probabilities. As EM attempts to maximise the total probability of the corpus, probabilities can be interpreted to represent each item's explanatory contribution to the corpus: the more frequently an item contributes to a highly scored derivation, the higher its probability is pushed by the algorithm.

By further examining the items with the highest probabilities, which are most likely to be applicable to novel sentences, we can therefore gain a sense of the content of the filtered lexicon.

Templates

We manually assign each of the 286 templates with a probability of over 0.0001 to one of the following classes:

- **function**: Templates that contain no delexicalisations or lexical content; they usually represent function words such as prepositions or punctuation.

Class	Count	Examples	Interpretation
Total	286		
function	63	NP/N : ⟨0⟩	*the*
		(S/S)/NP : ⟨0⟩ —time→ ⟨1⟩	*in*
generic	144	N : <lex>	*noun*
		NP : person —name→ name —op1→ <lex>	*person*
lex	79	N : person ←ARG1— expert-41	*expert*
date (of lex)	60	N\N : date-entity —day→ <lex>; —month→ 4; —year→ 2008	*date*

Table 6.3.: Classes of the 286 templates with the highest EM probabilities.

- **generic**: Templates that contain delexicalisations but no further lexical content. They can thus represent an entire class of content words by being instantiated together with a matching lexeme.

- **lex**: Templates that may or may not contain delexicalisations but that also contain lexical content. Often, these templates are created because the lexical content is distributed across several nodes, which is not permitted by the delexicalisation algorithm. A special case of this behaviour is represented by the sub-class **date**: In the corpus, dates often occur in specially formatted headers in the format YYYY-MM-DD, while the AMR represents year, month and day separately.

Counts and examples for each class are given in Table 6.3. The results show that after EM filtering, the large majority of templates are either delexicalised or are function words without lexical content. The majority of templates with lexical content are specially formatted dates.

Class	Count	Example
Total	6,389	
empty	5,772 (90 %)	
concept	235 (4 %)	(analysts, analyze-01); (nuclear, nucleus)
role	62 (1 %)	(but, ARG2)
constant	320 (5 %)	(indian, `"India"`)

Table 6.4.: Classes of all 6,389 filtered lexemes.

Lexemes

Lexemes can abstract over concepts, roles, or constants. They can also be empty, allowing a token to be instantiated with fully lexicalised templates. In fact, empty lexemes make up 90 % of the filtered lexeme dictionary. The counts for all lexeme classes are presented in Table 6.4.

The low counts for the non-empty categories are explained by the fact that the lexeme patterns cover most common cases of lexeme derivation (see Section 6.1.1). Table 6.4 thus only represents lexemes which do not fit into any of these patterns.

6.3. Supertagging

While EM filtering narrows down the number of templates required to explain the training corpus, the number of remaining templates (over 15,000) is still too large to be searched during parsing. The large number of hypotheses at the token level would lead to even larger numbers of hypotheses at later levels, causing parsing times to explode.

A supertagger can help reduce the search space that is exposed to the parser. It achieves this by predicting a distribution of scores over all templates for every token. This allows the parser to choose from the highest-scored templates instead of having to consider all templates.

In this section, we describe a supertagger based on the BiLSTM model introduced in Section 3.5. After running EM filtering on the training corpus, each token in the training corpus is associated with a single highest-scored template derived from the respective token. This allows us to train the supertagger on these template sequences. During testing, the model thus obtained can be used to predict templates for the parser to choose from.

Besides narrowing down the parser's search space, a second function of our supertagger is the completion of the training data. Since the lexicon induction algorithm is prone to early stopping, lexical items are not induced for every token, and these tokens are therefore not associated with a template. By using a masked loss function, which exempts these tokens from the backpropagation update, we prevent the tagger from predicting the "unknown" tag for these tokens. Using a jackknifing training scheme, we can then fill the gaps in the training data, allowing the parser to be trained with complete supertag predictions.

6.3.1. Architecture

Our supertagger follows the BiLSTM architecture introduced in Section 3.5. Figure 6.4 gives an overview of the tagger's architecture. It consists of the following components:

- Embedding matrices for tokens and syntactic categories.
- One or several BiLSTM layers.
- A fully connected layer which maps the BiLSTM output states to the dimension of template IDs.
- A softmax layer which normalises the predictions.

As inputs, indices of tokens and syntactic categories can be provided along with GloVe (Pennington, Socher and Manning 2014) or ELMo (Peters et al. 2018) embeddings. The tagger's output is a distribution over template IDs for every input token.

The embeddings are chosen for representing the two important classes of word representations: GloVe is a typical word-vector representation which is stored as a table, whereas ELMo is a large pre-trained language model which processes sentences as a whole and outputs contextualised word representations.

6.3.2. Training Data Extraction

The lexicon induction algorithm produces zero, one, or several lexical items for every constituent of a sentence which is represented in the CCG derivation used by the induction algorithm. Delexicalisation derives template and lexeme pairs from the lexical items, and EM filtering identifies the best templates and lexemes to explain the sentence.

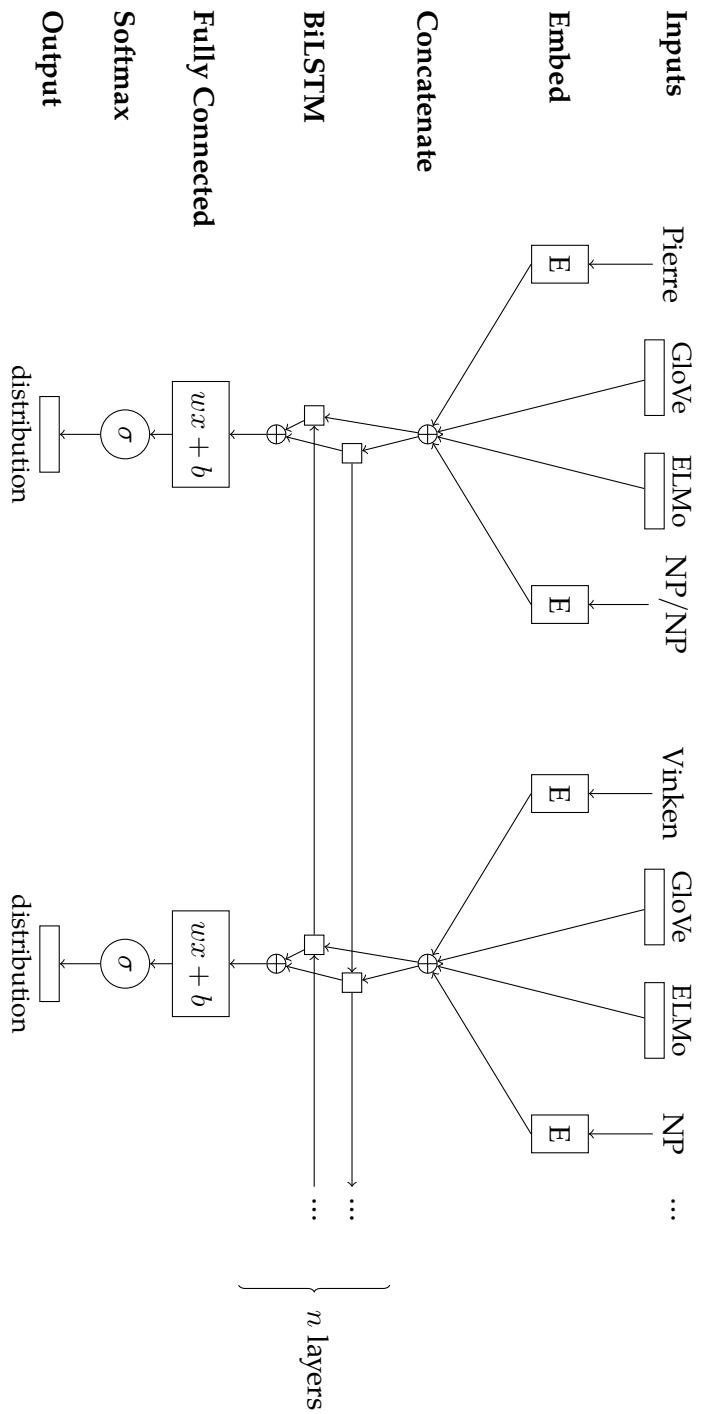


Figure 6.4.: Architecture overview of the BiLSTM supertagger.

After filtering, each token of the sentence is therefore covered by exactly one template; however, this template could span several tokens (if the induction algorithm aborted before reaching the token level and created a phrasal item). Since supertagger predictions are made on the token level, such templates cannot be predicted.

Intuitively, this is just as well since such abortions arise from configurations which GA-CCG is not equipped to deal with: either fundamental limitations of the grammar or problems with the annotations. In either case, it would be preferable to find alternative interpretations for the tokens in question to allow them to be processed, even if imperfectly.

Another issue which arises during training data extraction is sparsity. Templates which occur only once in the training data are unlikely to be predicted accurately by the supertagger. Such templates are therefore excluded from the training data.

When extracting training data for the supertagger, we therefore proceed as follows:

1. Assign every template a numerical ID, reserving an ID for a special `UNK` template.
2. For every token in the corpus, assign a template ID as follows:
   a) If the token is covered by a single-token template:
      - If the token occurs more than once in the training data, assign the template's ID.
      - Otherwise, assign `UNK`.
   b) If the token is covered by a multi-token template, assign `UNK`.

In addition to template IDs, the tagger uses syntactic categories as input. These are obtained by using the highest-scored predictions produced by EasyCCG's supertagger (Lewis and Steedman 2014).

The supertagger can then be trained to predict the annotated template IDs. However, it is not desirable to train the tagger to predict `UNK` templates, since these annotations do not provide any useful information to the parser. Instead, during prediction, the supertagger should always output plausible template IDs. This can be achieved by training with a masked loss function.

6.3.3. Masking the Loss Function

When the supertagger is trained using backpropagation, the output of the tagger is compared to the gold standard annotation using a loss function. The latter

is represented by a one-hot vector which has a value of 1 at the position of the annotated template and 0 everywhere else, whereas the prediction is a distribution over all templates with values summing to one. In such a situation, a categorical cross-entropy loss is appropriate[3]:

$$L(\hat{y}, y) = -\sum_{i=1}^{n} y_i \log \hat{y}_i \tag{6.1}$$

where $y = y_1 \dots y_n$ is the one-hot vector representing the annotated template ID, and $\hat{y} = \hat{y}_1 \dots \hat{y}_n$ the distribution predicted by the model.

The loss is then backpropagated through the neural network layers and used to update the weights in order to shift the output distribution closer towards the label.

To prevent the model from learning to predict the `UNK` template, the loss function can be masked. Let $y_{\texttt{UNK}}$ be the one-hot vector representing the `UNK` label. The masked loss is then defined as follows:

$$L_{\text{masked}}(\hat{y}, y) = \begin{cases} 0 & \text{if } y = y_{\texttt{UNK}} \\ L(\hat{y}, y) & \text{otherwise} \end{cases} \tag{6.2}$$

For tokens labelled `UNK`, the masked loss function outputs zero, so that the token does not contribute to weight updates. This causes it to be effectively exempt from backpropagation. Since the `UNK` label is never positively reinforced, it disappears from the model's output space.

6.3.4. Decoding

For every input token, the supertagger produces a full distribution over all templates in the lexicon: each template is assigned a score between 0 and 1, with all scores summing to 1. That is, the output distribution fulfills the formal properties of a probability distribution, even if the model that produced it is not probabilistic.

Decoding these output distributions is a matter of selecting a subset of top-scoring template IDs to pass on to the downstream parser. The simplest decoding strategy would output the single highest-scoring template for every token, yielding a unique sequence of template IDs mirroring the tagger's training data. However, given that the tagger has no explicit capability for representing the syntactic and semantic mechanisms which are the domain of the parser's search, it makes sense to present several possible templates to the parser so that it can select from representations of

[3] See Goldberg (2017), page 27.

a token that may have different syntactic or semantic properties. This is especially important when the tagger fails to identify a clear "winning" template but assigns similar scores to several candidates.

Our supertagger therefore implements the following filtering parameters which can be combined to construct a flexible decoding strategy:

- **max-predictions:** Specifies the maximum number of template IDs that are included in the output for any token.
- **min-confidence:** Specifies a minimum score threshold for a predicted template to be included in the output.
- **clip-confidence:** Specifies that the output should be clipped after a given cumulative score has been reached. That is, if the combined confidence of the top k predictions exceeds this value, no further predictions are output.

In combination, these parameters allow us to efficiently handle both situations where the supertagger makes a clear prediction (in which case no low-probability predictions are output), and cases where the output distribution is less defined (in which case the output is still pruned so as to avoid overloading the parser).

6.3.5. Predicting Tags for Training Data

In Section 6.3.2, we described how tag sequences can be extracted from the filtered output of the lexicon induction algorithm. However, these sequences contain gaps in places where no token-level templates were induced. In addition, the trained supertagger is not guaranteed to exactly reproduce its input distribution on novel sentences; in fact, it is plausible to assume that it performs some degree of smoothing, preferring frequent templates to rare ones.

Since the parser must work with supertagger predictions when processing novel sentences, it makes sense to also use predictions when training the parser. However, using the predictions produced by a tagger for its own training data should be avoided, as the tagger fits its training data much more tightly than it does novel sentences.

Jackknifing is an algorithm for obtaining predictions on a training data set while avoiding the issue of overfitting to the training data. By splitting the training corpus into n parts, the supertagger can in turn be trained on $n - 1$ parts to predict tag distributions for the nth part.

The procedure of jackknifing can be described as follows:

1. Split the training data into n parts $train_1, \ldots, train_n$ of equal size.
2. For all $i = 1, \ldots, n$:
   a) Train a tagger model $tagger_i$ on the concatenated training data parts $train_1, \ldots, train_{i-1}, train_{i+1}, \ldots, train_n$.
   b) Use $tagger_i$ to predict tags for the sentences in $train_i$, yielding the output $tags_i$.
3. Concatenate $tags_1, \ldots, tags_n$.

While we use the jackknifed tagger models to predict labels for the training data, we use a model trained on the entire training data set to obtain predictions for novel sentences in the dev and test data sets.

6.3.6. Tuning Experiments

We conduct a series of experiments to validate the supertagger architecture and determine the best settings for the hyperparameters of the model. We identify the following dimensions which are likely to impact the model's performance:

- The representation of input words via GloVe embeddings, ELMo vectors, and/or trained embeddings.
- The inclusion of syntactic categories as a separate feature.
- The layout of the BiLSTM, varying by the size of the LSTM state vector as well as by the number of LSTM layers.

Tuning experiments are performed by varying these parameters individually, starting from a baseline configuration. Table 6.5 gives an overview of the settings tested in the experiments.

We keep other aspects of the model constant, using sensible defaults:

- For optimization, the Adam algorithm is used (Kingma and Ba 2017). Adam is a momentum-based algorithm commonly used as a default algorithm for training neural networks.
- Initialisation is left at the defaults set by the Keras library[4]: Uniform for trained embeddings and Glorot Uniform with a zero bias vector for the output layer

[4] See `https://keras.io/` (Retrieved 19 Nov 2021).

Parameter	Settings
Embeddings	trained20, trained100, GloVe, ELMo, GloVe+trained20, **ELMo+trained20**, ELMo+GloVe+trained20
Features	none, **syncat**
LSTM size	50, **100**, 200
BiLSTM layers	**1**, 3, 7

Table 6.5.: The hyperparameter settings for the supertagger which are covered by our experiments. Settings in bold are used as baseline settings unless otherwise noted.

(Glorot and Bengio 2010). For the LSTM layers, the kernel matrices W^{x*} are initialised using the Glorot Uniform method, while the recurrent kernel matrices W^{h*} are initialised using a random orthogonal matrix.

- A dropout of 0.1 is applied on the embedded inputs. Each LSTM layer uses a dropout of 0.1 for the recurrent state.
- Syntactic categories are embedded using 20-dimensional trained embeddings.
- During training, 10 % of the training set are kept for validation. Training continues until the top5 metric on the validation set has not increased for five iterations, and the weights from the iteration with the highest validation top5 score are stored.

We evaluate by computing the *top1* and *top5* metrics, representing the percentage of tokens where the annotated template was the highest scored prediction, or among the five highest scored, respectively. Tokens labeled `UNK` are ignored by the metric. The evaluation is performed using leave-one-out cross validation by averaging the scores obtained for each jackknifing split.

Embeddings

To examine the influence of embeddings on the supertagger, we experiment with various combinations of trained embeddings, GloVe, and ELMo embeddings. The results are shown in Table 6.6.

The baseline configuration with the elmo+trained setting achieves the highest top1 score, while adding GloVe embeddings slightly increases the top5 score – but

Condition	top1	top5
trained20	0.6856	0.8663
trained100	0.6858	0.8653
glove	0.6367	0.8962
glove+trained20	0.6348	0.8917
elmo	0.7041	0.8967
elmo+trained20	**0.7150**	0.9009
elmo+glove+trained20	0.6457	**0.9027**

Table 6.6.: The performance of the supertagger with various embedding methods.

Condition	top1	top5
none	0.7101	0.8992
syncat	**0.7166**	**0.9005**

Table 6.7.: The performance of the supertagger with and without syntactic category features.

with a large negative impact on the top1 score[5].

This effect is consistent across all GloVe conditions: While GloVe embeddings perform well in the top5 evaluation, the top1 scores of the respective conditions are low. This may be a consequence of the overgeneralisation inherent in word vectors. In the GloVe approach, all words with the same surface form share a representation, including homonyms as well as grammatically distinct forms of the same root. This ambiguity could explain how GloVe vectors allow the model to place the correct prediction near the top, but not necessary in first place.

The contextualised embeddings of ELMo perform much better. Trained embeddings also perform surprisingly well on their own. However, they only provide a small improvement when used together with ELMo vectors.

Condition	top1	top5
size=50.layers=1	0.7108	0.8930
size=50.layers=3	0.6740	0.8416
size=50.layers=7	0.6212	0.7888
size=100.layers=1	0.7156	0.9023
size=100.layers=3	0.6843	0.8536
size=100.layers=7	0.6210	0.7916
size=200.layers=1	**0.7171**	**0.9026**
size=200.layers=3	0.6850	0.8606
size=200.layers=7	0.6461	0.8128

Table 6.8.: The performance of the supertagger with varied BiLSTM size and layer count.

Features

Since syntactic categories are a component of templates, syntactic category features could be a good predictor for templates. Also, the supertagger of EasyCCG is trained on the sizeable CCGbank corpus, so its model contains linguistic information that goes beyond the supertagger's training corpus. On the other hand, adding a syntactic category feature puts the burden on the model of judging the correctness of the supplied category. Also, ELMo embeddings already include syntactic information that could be sufficient to predict syntactic categories.

The experimental results in Table 6.7 confirm that the benefit of adding syntactic category features is small, but detectable. Since the added computational cost is also small, there is no harm in using them.

LSTM Architecture

The LSTM architecture is varied along two dimensions: the size of the LSTM state vector, and the number of BiLSTM layers. The results in Table 6.8 show clear trends along both axes.

A larger LSTM size is associated with higher model performance. However, the increase achieved beyond a dimension of 100 is very small.

[5]The baseline configuration from Table 6.5 appears in the various experimental series with slightly varying scores. These small variations are due to random initialisation in repeated runs of the experiments.

On the other hand, additional BiLSTM layers diminish performance. Presumably, the training data set is too small to allow deeper models to converge without overfitting.

Summary

The experimental results validate the baseline parameters outlined in Table 6.5. When interpreting the top1 and top5 accuracy scores, it should be noted that we do not expect the tagger to predict all tags as annotated, as the annotations themselves are not fully trusted. For example, although `UNK` gaps are excluded from the evaluation, the templates induced for neighbouring tokens could also be affected by an induction error and contain erroneous material. In an ideal situation, the tagger could smooth over such issues by predicting more general templates than those in the annotations. Nevertheless, the top5 accuracy of 90 % achieved by the model shows that the supertagger is able to produce high quality predictions on the sizeable template dictionary.

Chapter 7.

Parsing with Graph Algebraic Combinatory Categorial Grammars

In the preceding chapters, we have described the components required for producing a compact GA-CCG lexicon. In this chapter, we show how to put this lexicon to use by applying it to AMR parsing.

The core of our GA-CCG parser is a bottom-up chart parsing algorithm, a version of the CKY algorithm described in Section 3.1.6. The parser is driven by a linear model trained using the structured perceptron algorithm introduced in Section 3.3. This parsing algorithm is described in Section 7.1.

When training the parser, we face an additional obstacle: While we require the parser to produce GA-CCG derivations, there is no trusted source of such derivations as they are not contained in the AMR annotations. This means that we cannot directly train the parser on gold-standard derivations. Training the parser is therefore an instance of *hidden variable learning*, and we thus propose an oracle intended to produce good derivations to use as a training objective. The oracle is described in Section 7.3, and Section 7.2 covers how to train the parser using the oracle and a structured perceptron algorithm. In section 7.4, we examine how the trained model may be used to parse novel sentences.

In contrast to the preceding chapters, this chapter does not include an evaluation. Instead, Chapter 8 is dedicated to the end-to-end evaluation of the parsing pipeline described in this thesis.

7.1. Parsing Algorithm

The algorithm used to parse GA-CCG is a simple extension of the beam-search CKY algorithm (Algorithm 3.2) presented in Section 3.3.1. We extend it with an explicit representation of unary conversions, as shown in Algorithm 7.1.

Apart from this change, the approach of the algorithm is the same: The algorithm iterates over spans of the sentence in an order that ensures small spans get processed before the larger spans encompassing them. On the token level, lexical entries are entered into the chart. On larger spans, possible combinations of chart entries are explored. After a given chart cell has been filled, its entries are scored and pruned according to the beam size. Afterwards, unary conversions of the chart entries are added and another round of beam pruning is performed.

Algorithm 7.1 is fairly abstract in terms of the grammar it encodes, only assuming a binary branching structure of derivations. Its connection to GA-CCG is established by the three generator functions GenLex, GenCmb, and GenUnary, which implement the lexicon and combinators of GA-CCG.

The GenLex function (Algorithm 7.2) is used to enumerate lexical entries for a given token. It accesses the template and lexeme dictionaries created through delexicalisation (see Section 6.1). First, the lexeme dictionary is scanned for lexemes that match the current token. The algorithm then attempts to use the lexeme to lexicalise templates. Lexicalisation is carried out using the Delex^{-1} function. For all templates for which lexicalisation is successful, the resulting lexical entry is added to the result set.

Definition 7.1 (Lexicalisation)**.** Lexicalisation is the process of constructing a GA-CCG derivation node containing a token, a syntactic category, and a semantic category, from a template-lexeme pair. It is the inverse of the Delex function described in Definition 6.1. Let τ be a template and λ be a lexeme. The lexicalisation of τ and λ is defined as follows:

$$\textsc{Delex}^{-1}(\tau, \lambda) = \{(w, c_{\text{syn}}, c_{\text{sem}}) \mid (\tau, \lambda) \in \textsc{Delex}(t, c_{\text{syn}}, c_{\text{sem}})\} \tag{7.1}$$

For any template-lexeme pair, there are either one or zero lexicalisations. In particular, there are zero lexicalisations if the `<lex>`-slot in τ does not match the label in λ: for example, if the lexeme contains an edge label, but the template contains a `<lex>`-slot for an edge label. If the template and lexeme do match, the lexicalised meaning representation can be obtained by replacing the `<lex>`-slot in the template with the lexeme's label.

The GenCmb function (Algorithm 7.3) computes the constituents that result from applying a binary grammar rule. It is called on a pair of adjacent derivation nodes x_1, x_2. The algorithm checks every binary grammar rule for applicability by matching the syntactic categories of x_1 and x_2 against the rule's patterns. If the patterns match, the combinator and semantic operator associated with the rule are used to compute the syntactic and semantic categories of the resulting constituent.

As explained in Section 4.2.3, unary rules do not affect the semantic content of a constituent in our implementation. The job of the GenUnary function (Algorithm 7.4) is therefore to find all applicable unary rules by matching the syntactic category of a constituent against the rule's pattern, and then compute a resulting syntactic category using the rule's combinator. The semantic category is simply copied to the output constituent.

7.1.1. Coreference Resolution

GA-CCG includes a mechanism for representing coreferences, which allows the same entity to be referred to by several nodes (see Section 4.3.1). This mechanism is required because of the context-free nature of CKY parsing: the parser cannot look outside the span it is currently processing. For example, if an entity is referred to by a pronoun such as "it", the parser has to represent it by a semantically underspecified node in order to build a connected meaning representation graph.

In GA-CCG, underspecified nodes are labelled `<coref>`. Coreference nodes are expected to be merged with another node which represents the fully specified entity that is being referred to. We implement this requirement by adding *coreference resolution* as a post-processing step after parsing. For every AMR graph in the result set of the parsing algorithm, we apply merge operations to all of its coreference nodes. For every coreference, we greedily choose the merge operation that maximises the score assigned to the resulting graph by the parsing model (see Algorithm 7.5).

The algorithm for coreference resolution makes use of the Merge operation. Given a graph G and vertices v_1, v_2, Merge merges v_2 into v_1, so that all edges adjacent to v_2 are now adjacent to v_1. In detail, this operation can be defined as follows:

$$\textsc{Merge}(G, v_1, v_2) = (V', E', v'_{\text{root}}, l', \mathit{slab}')$$

where

$$
\begin{aligned}
G &= (V, E, v_{\text{root}}, l, \mathit{slab}) \\
V' &= V \backslash \{v_2\} \\
E' &= V \backslash \{(v_2, v) | v \in V\} \\
&\quad \backslash \{(v, v_2) | v \in V\} \\
&\quad \cup \{(v_1, v) | (v_2, v) \in E\} \\
&\quad \cup \{(v, v_1) | (v, v_2) \in E\}
\end{aligned}
$$

$$
v'_{\text{root}} = \begin{cases} v_1 & \text{if } v_{\text{root}} = v_2 \\ v_{\text{root}} & \text{otherwise} \end{cases}
$$

$$
l'(e) = \begin{cases} l(v_1) & \text{if } e = v_2 \\ l(e) & \text{otherwise} \end{cases}
$$

$$
\mathit{slab}'(v) = \begin{cases} \mathit{slab}(v_1) \cup \mathit{slab}(v_2) & \text{if } v = v_2 \\ \mathit{slab}(v) & \text{otherwise} \end{cases}
$$

The algorithm for coreference resolution is deliberately kept simple as we do not expect it to have a large impact on Smatch scores. Importantly, it ensures that well-formed AMR graphs are created.

Algorithm 7.1 A beam-search CKY algorithm for GA-CCG parsing.

Inputs:

- A token sequence $w_1, \ldots, w_n$
- A beam size k
- A generator of lexical items GenLex
- A generator of combined chart items GenCmb
- A generator of unary conversions GenUnary
- A scoring function for chart entries Score

Output: A parse chart C containing the derived items

function Parse($w_1, \ldots, w_n$)
  $C[i, j] \leftarrow \{\}$ for $1 \leq i \leq j \leq n$
  for $j \leftarrow 1, \ldots, n$ **do**
    for $i \leftarrow j, \ldots, 1$ **do** ▷ Iterate over start index
      if $i = j$ **then**
        for $d \in$ GenLex(w_i) **do**
          $C[j, j] \leftarrow C[j, j] \cup \{d\}$ ▷ Insert lexical items
        end for
      else
        for $k \leftarrow i + 1, \ldots, j$ **do** ▷ Iterate over split index
          for $d_1 \in C[i, k-1]; d_2 \in C[k, j]$ **do**
            for $d \in$ GenCmb(d_1, d_2) **do**
              $C[i, j] \leftarrow C[i, j] \cup \{d\}$ ▷ Apply combinatory rules
            end for
          end for
        end for
      end if
      $C[i, j] \leftarrow \max k_{\text{Score}(d)}\{d \in C[i, j]\}$ ▷ Enforce beam limitation
      for $d \in C[i, j]$ **do**
        for $d' \in$ GenUnary(d) **do**
          $C[i, j] \leftarrow C[i, j] \cup \{d'\}$ ▷ Add unary conversions
        end for
      end for
      $C[i, j] \leftarrow \max k_{\text{Score}(d)}\{d \in C[i, j]\}$ ▷ Enforce beam limitation
    end for
  end for
end function

Algorithm 7.2 Lexical generation algorithm for GA-CCG.

Inputs:
- A token w
- A set of templates $\mathcal{T}$
- A set of lexemes $\mathcal{L}$

Output: A set of GA-CCG constituents R

function GENLEX($w, \mathcal{T}, \mathcal{L}$)
  $R \leftarrow \{\}$ ▷ Initialise result set
  for $\lambda = (w', \mathit{label}) \in \mathcal{L}$ **do**
    if $w = w'$ **then** ▷ Find lexemes matching the token
      for $\tau \in \mathcal{T}$ **do** ▷ Try applying the available templates
        $R \leftarrow R \cup \text{DELEX}^{-1}(\tau, \lambda)$ ▷ Combine template and lexeme
      end for
    end if
  end for
  return R
end function

Algorithm 7.3 Combinatory hypothesis generation algorithm for GA-CCG.

Inputs:
- GA-CCG derivation nodes d_1, d_2
- A set of binary GA-CCG rules R_2

Output: A set of GA-CCG derivation nodes R

function GENCMB(d_1, d_2)
  $R \leftarrow \{\}$ ▷ Initialise result set
  for $r = (C, p, o) \in R_2$ **do** ▷ Iterate over grammar rules
    if ISAPPLICABLE(r, SYN(d_1), SYN(d_2)) **then** ▷ Check applicability of rule
      $C \leftarrow \text{CMB}(r)$
      $c'_{\text{syn}} \leftarrow C(\text{SYN}(d_1), \text{SYN}(d_2))$ ▷ Compute syntactic category
      $o \leftarrow \text{SEM}(r)$
      $c'_{\text{sem}} \leftarrow o(\text{SEM}(d_1), \text{SEM}(d_2))$ ▷ Compute semantic category
      $R \leftarrow R \cup \{(C, c'_{\text{syn}}, c'_{\text{sem}}, d_1 d_2)\}$
    end if
  end for
  return R
end function

Algorithm 7.4 Unary hypothesis generation algorithm for GA-CCG.

Inputs:
- A GA-CCG derivation node d
- A set of unary CCG combinators R_1

Output: A set of GA-CCG derivation nodes R

function GENUNARY(d)
  $R \leftarrow \{\}$ ▷ Initialise result set
  for $C \in R_1$ **do** ▷ Iterate over unary rules
    if ISAPPLICABLE(C, SYN(d)) **then** ▷ Check applicability of rule
      $c'_{\text{syn}} \leftarrow C(\text{SYN}(d))$ ▷ Use combinator to obtain syntactic category
      $R \leftarrow R \cup \{(C, c'_{\text{syn}}, \text{SEM}(d), d)\}$
    end if
  end for
  return R
end function

Algorithm 7.5 Algorithm for greedy coreference resolution.

Inputs:

- A GA-CCG s*-graph $G = (V, E, v_{root}, l, slab)$
- Two vertices $v_1, v_2 \in V$

Output: A GA-CCG s*-graph G' which does not contain any coreference nodes

function ResolveCoreferences(G)
  $G' \leftarrow G$
  for $v \in V$ with $l(v) =$ `<coref>` **do**
    $C \leftarrow \{\}$ ▷ Initialise candidate set
    for $v' \in V'$ with $l(v) \neq$ `<coref>` **do**
      $G'' \leftarrow$ Merge(G', v', v)
      $C \leftarrow C \cup G''$
    end for
    $G' \leftarrow \text{argmax}_{G'' \in C}$ Score(G'')
  end for
  return G'
end function

7.2. Training the Parser

Algorithm 7.1 describes how to build GA-CCG derivations given a scoring function Score. The scoring function decides which hypotheses are kept in the beam and ranks the parser's potential final outputs. It therefore plays a crucial role in obtaining high-quality derivations and meaning representations.

In this thesis, we describe two scoring functions: The principal scoring function is a linear model trained using a perceptron algorithm (see Equation 3.2). It is described in this section. When training this model, we also make use of an oracle scoring function which computes a heuristic score based on knowledge of the annotated AMR. The oracle scoring function is described in Section 7.3.

Our training loop operates on the parse charts produced by Algorithm 7.1. It computes updates by comparing the k outputs of the parser, where k is the configured beam size. The parser outputs are ranked by the scoring function, but they can also be evaluated against the gold standard annotations. The training objective is to push the best-evaluated parses to the top of the parser's top-k list.

7.2.1. Training Loop

We first introduce the algorithm for training the GA-CCG parser model in a general form in Algorithm 7.6. It is an instance of the classic perceptron algorithm (Algorithm 3.3):

1. First, a set of predictions $\mathcal{Y}$ is computed by parsing the example sentence and performing coreference resolution (lines 5–7).

2. Next, an update vector is computed (line 8). The Update function is implemented by the CostSensitiveUpdate described in this section.

3. Finally, a learning rate vector is applied to the update vector (line 9). The LearningRate function returns a vector of per-parameter learning rates which are be computed using the Adadelta algorithm (see Section 3.3.2).

The algorithm iterates over the training data set for a given number of iterations. In each iteration, the data set is shuffled in order to reduce the dependency of the model on the order of examples.

Algorithm 7.6 The training loop.

Inputs:

- A data set $(\tilde{x}_1, \tilde{y}_1), \ldots, (\tilde{x}_N, \tilde{y}_N)$
- An iteration count T
- An initial weight vector $\mathbf{w}_{\text{init}}$

Output: A weight vector $\mathbf{w}_t$

$\mathbf{w}_N^0 \leftarrow \mathbf{w}_{\text{init}}$
for $t \leftarrow 1 \ldots T$ **do**
  $\mathbf{w}_0^t \leftarrow \mathbf{w}_N^{t-1}$
  for $i \leftarrow \textsc{Shuffle}(1 \ldots N)$ **do**
    $C \leftarrow \textsc{Parse}(\tilde{x}_i)$ ▷ Run parser
    $\mathcal{Y} \leftarrow C[1, |\tilde{x}_i|]$ ▷ Extract output parses
    $\mathcal{Y}_r \leftarrow \{\textsc{ResolveCoreferences}(y) \mid y \in \mathcal{Y}\}$ ▷ Resolve coreferences
    $u \leftarrow \textsc{Update}(\mathcal{Y}_r, \tilde{y}_i)$ ▷ Compute update vector
    $r \leftarrow \textsc{LearningRate}(u)$ ▷ Compute learning rate
    $w_i^t \leftarrow w_{i-1}^t + r \odot u$
  end for
end for
return $\mathbf{w}_N^T$

Algorithm 7.7 The cost-sensitive perceptron update algorithm.

Inputs:

- A set of output derivations $\mathcal{Y}$
- A gold-standard output $\tilde{y}$
- A feature extractor $\Phi(y)$
- A cost function $\text{cost}(y)$
- A margin scaling factor λ

Output: An update vector u

function CostSensitiveUpdate($\mathcal{Y}, \tilde{y}$)
  $\mathit{cost}_{\text{min}} \leftarrow \min_{y \in \mathcal{Y}} \text{cost}(y, \tilde{y})$
  $\Delta(y) \leftarrow \text{cost}(y) - \mathit{cost}_{\text{min}}$ for all $y \in \mathcal{Y}(\tilde{x}_i)$
  $G \leftarrow \{g \mid g \in \mathcal{Y}, \Delta(g) = 0\}$
  $B \leftarrow \mathcal{Y} \backslash G$
  $C \leftarrow \{c \mid c \in G; \exists z : z \in B, \mathbf{w}_{i-1}^t(\Phi(c) - \Phi(z)) < \lambda\Delta(z)\}$
  $E \leftarrow \{e \mid e \in B; \exists y : y \in G, \mathbf{w}_{i-1}^t(\Phi(y) - \Phi(e)) < \lambda\Delta(e)\}$
  $\tau(e) \leftarrow \sum_{c \in C} \frac{v_c(e)}{|C| \sum_{e' \in E} v_c(e)}$ for all $e \in E$
    where $v_c(e) = \begin{cases} 1 & \text{if } \mathbf{w}_{i-1}^t(\Phi(c) - \Phi(e)) < \lambda\Delta(e) \\ 0 & \text{otherwise} \end{cases}$
  return $\sum_{c \in C} \frac{\Phi(c)}{|C|} - \sum_{e \in E} \tau(e)\Phi(e)$
end function

7.2.2. Cost-sensitive Perceptron

A cost-sensitive perceptron algorithm has been introduced in Section 3.3.4. The update function described in Algorithm 7.7 applies this technique to GA-CCG parsing with very little modification. The update function can be used as an implementation for UPDATE in Algorithm 7.6.

The input of the update function is a set of complete parses, each of which is evaluated. Given an output derivation y which yields the meaning representation g_y, and the annotated meaning representation $g_{\tilde{y}}$, we define the cost function as follows:

$$\text{cost}(g_y, g_{\tilde{y}}) = 1 - \textsc{Smatch}_{\text{F1}}(g_y, g_{\tilde{y}}) \tag{7.2}$$

Since Smatch scores range between 0 and 1, likewise does the cost function. Like Singh-Miller and Collins (2007), we set the margin scaling factor λ to 1.

7.2.3. Scoring Function

The score assigned to a hypothesis is calculated from the final weight vector of the perceptron.

$$\textsc{Score}(x, y) = \mathbf{w}_N^{T\top} \Phi(x, y) \tag{7.3}$$

7.3. Oracle Parsing

Training a GA-CCG parser is an instance of hidden variable learning, as the parser needs to reason about derivations which are not observed in the training data. Apart from the the input of the parser, the sentence x, and the AMR graph y which forms the label, the parser also needs to concern itself with the GA-CCG derivation z which transforms x into y.

When making an update, the structured perceptron algorithm requires a pair of states: an *incorrect* state whose feature vector is subtracted, and a *correct* state whose feature vector is added (see Section 3.3.3). The correct state should stem from a derivation which produces the correct meaning representation. However, such a derivation is not trivial to construct. Due to the size of the search space, an untrained parser is unlikely to find a derivation which produces the annotated meaning representation.

To solve this bootstrapping problem, we define an oracle which guides the parser towards the correct label. It scores hypotheses according to correctness and coverage of the annotated meaning representation. The parser is driven using the oracle as a scoring function. The best full derivation produced by the parser can then be used as a positive example for the parser to update towards.

7.3.1. Computing the Oracle Function

Our oracle is derived from the precision and recall measures that are also used for the Smatch evaluation score (see Section 2.1.5). Indeed, Smatch appears to be a good starting point for defining an oracle, as it is the score we wish to optimise. However, Smatch has two properties which make it unsuitable for direct use as an oracle:

- Smatch is intended to compare two complete meaning representation, while the oracle needs to be able to evaluate partial hypotheses in the form of GA-CCG s*-graphs.
- The computation of a Smatch score involves globally optimising an alignment between the nodes of both graphs to achieve the best possible score. This makes Smatch expensive to compute.

To account for these issues, we simplify the computation of the precision and recall measures. Our measures disregard the global structure of the graph and evaluate its elements individually. *Element-wise precision* checks nodes, half-edges, and edges of the hypothesis graph for presence in the annotated meaning representation. *Alignment coverage* encourages hypotheses that instantiate all relevant concepts using the annotated token-node alignments. Our oracle function is the harmonic mean of both measures.

Definition 7.2 (Element-Wise Precision)**.** Let G^{hyp} and G^{gold} be GA-CCG s*-graphs. The *element-wise precision* of G^{hyp} is defined as follows:

$$P(G^{\text{hyp}}, G^{\text{gold}}) = 1 - \frac{|\,\text{elements}(G^{\text{hyp}}) \setminus \text{elements}(G^{\text{gold}})|}{|\,\text{elements}(G^{\text{hyp}})|} \tag{7.4}$$

where $\text{elements}(G)$ with $G = (V, E, v_{\text{root}}, l, \mathit{slab})$ is the following multiset:

$$\begin{aligned}\text{elements}(G) = &\{(l(v)) \,|\, v \in V'\} \\ &\cup \{(l(v_1), l(e)) \,|\, e = (v_1, v_2) \in E; v_1 \in V')\} \\ &\cup \{(l(e), l(v_2)) \,|\, e = (v_1, v_2) \in E; v_2 \in V'\} \\ &\cup \{(l(v_1), l(e), l(v_2)) \,|\, e = (v_1, v_2) \in E; v_1, v_2 \in V'\}\end{aligned} \tag{7.5}$$

$$V' = \{v \,|\, v \in V; \langle i \rangle \notin \mathit{slab}(v); l(v) \neq \texttt{<coref>}\} \tag{7.6}$$

When calculating the element-wise precision, we count the number of incorrect elements in the hypothesis. These are nodes, half-edges, or edges which are not found in the gold-standard meaning representation. Differently from the Smatch algorithm, each element is identified only by its labels, not by its links to other elements. Since the elements are stored as a multiset, element-wise precision takes element counts into account: for example, if two nodes of a given label are in the hypothesis, but the label occurs only once in the gold-standard representation, the second instance is counted as an error.

Definition 7.3 (Alignment Coverage)**.** Let $G^{\text{hyp}} = (V^{\text{hyp}}, E^{\text{hyp}}, v_{\text{root}}^{\text{hyp}}, l^{\text{hyp}}, \mathit{slab}^{\text{hyp}})$ and $G^{\text{gold}} = (V^{\text{gold}}, E^{\text{gold}}, v_{\text{root}}^{\text{gold}}, l^{\text{gold}}, \mathit{slab}^{\text{gold}})$ be GA-CCG s*-graphs. Furthermore, let $A \subset \mathbb{N} \times V^{\text{gold}}$ be the set of token-node alignment edges annotated for the sentence of G^{gold}, and let i and k be the first and last indices of the tokens represented by G^{hyp}.

The *alignment coverage* of G^{hyp} is defined as follows:

$$R(G^{\text{hyp}}, G^{\text{gold}}) = \frac{|V_c|}{|V_a|} \tag{7.7}$$

where

$$V_a = \{v \,|\, v \in V^{\text{gold}}; \exists (j, v) \in A : i \leq j \leq k\} \tag{7.8}$$

$$V_c = \{v \,|\, v \in V_a; \exists v' \in V^{\text{hyp}} : l^{\text{gold}}(v') = l^{\text{hyp}}(v)\} \tag{7.9}$$

Alignment coverage is the ratio of two sets of nodes: V_a is the set of nodes that are annotated with token-node alignments, limited to the tokens represented by the current hypothesis. The concepts represented by these nodes are therefore expected to be present in the hypothesis. For every aligned node in the gold meaning representation, we check whether the hypothesis contains a node with a matching

label, and collect the corresponding nodes in V_c. Alignment coverage can thus be considered an approximation of the recall of node labels in the hypothesis graph.

Both edge precision and alignment coverage suffer from pathological edge cases. With edge precision, (almost) empty meaning representations would be rewarded since they contain few wrong edges. On the other hand, alignment coverage does not take additional nodes into account and would therefore reward hypotheses containing more nodes than necessary. Analogous to the F-score metric used in Smatch (Section 2.1.5), we construct our oracle as the harmonic mean of both measures.

Definition 7.4 (Oracle Scoring Function)**.** Let G^{hyp} and G^{gold} be GA-CCG s*-graphs. We define an *oracle scoring function* as follows:

$$\textsc{Score}_O(G^{\text{hyp}}, G^{\text{gold}}) = 2\frac{P(G^{\text{hyp}}, G^{\text{gold}})R(G^{\text{hyp}}, G^{\text{gold}})}{P(G^{\text{hyp}}, G^{\text{gold}}) + R(G^{\text{hyp}}, G^{\text{gold}})} \tag{7.10}$$

This scoring function can be used as an implementation of the scoring function Score in the parsing algorithm (Algorithm 7.1). Using the oracle scoring function, the parser searches for derivations whose yield is as similar as possible to the annotated meaning representation, without requiring an exact match.

7.3.2. Bootstrapping the Training Loop

As noted above, oracle parsing can help solve the bootstrapping problem of an untrained parser. With its initial parameters, a parser is unlikely to find any good parses, and is therefore unable to perform high-quality updates. The extended training loop in Algorithm 7.8 uses the oracle to provide high-quality parses. In the first iteration, the parser and coreference resolution are run separately using the oracle scoring function. The oracle parsing results are made available to update computation along with the learned parser's results.

7.4. Inference

The result of training is a weight vector w_N^T which represents the information extracted from the data set by the training algorithm. This weight vector is equally suitable for parsing novel sentences. No special steps are required: it is sufficient to perform the same Parse and ResolveCoreferences steps that are used in the training loop. The predicted derivation is the result with the highest score according to the weight vector. This procedure is summarised in Algorithm 7.9.

Algorithm 7.8 An extended training loop with oracle bootstrapping in the first iteration.

Inputs:
- A data set $(\tilde{x}_1, \tilde{y}_1), \ldots, (\tilde{x}_N, \tilde{y}_N)$
- An iteration count T
- An initial weight vector $\mathbf{w}_{\text{init}}$

Output: A weight vector $\mathbf{w}_t$

$\mathbf{w}_N^0 \leftarrow \mathbf{w}_{\text{init}}$
for $t \leftarrow 1 \ldots T$ **do**
  $\mathbf{w}_0^t \leftarrow \mathbf{w}_N^{t-1}$
  for $i \leftarrow \textsc{Shuffle}(1 \ldots N)$ **do**
    $C \leftarrow \textsc{Parse}(\tilde{x}_i)$ ▷ Run learned parser
    $\mathcal{Y} \leftarrow C[1, |\tilde{x}_i|]$ ▷ Extract output parses
    $\mathcal{Y}_r \leftarrow \{\textsc{ResolveCoreferences}(y) \mid y \in \mathcal{Y}\}$ ▷ Resolve coreferences
    if $i = 1$ **then**
      $C' \leftarrow \textsc{Parse}_{\textsc{Oracle}}(\tilde{x}_i)$ ▷ Run oracle parser
      $\mathcal{Y}' \leftarrow C'[1, |\tilde{x}_i|]$ ▷ Extract output parses
      $\mathcal{Y}'_r \leftarrow \{\textsc{ResolveCoreferences}_{\textsc{Oracle}}(y) \mid y \in \mathcal{Y}'\}$ ▷ Resolve coreferences
      $\mathcal{Y}_r \leftarrow \mathcal{Y}_r \cup \mathcal{Y}'_r$ ▷ Combine parser outputs
    end if
    $u \leftarrow \textsc{Update}(\mathcal{Y}_r, \tilde{y}_i)$ ▷ Compute update vector
    $r \leftarrow \textsc{LearningRate}(u)$ ▷ Compute learning rate
    $w_i^t \leftarrow w_{i-1}^t + r \odot u$
  end for
end for
return $\mathbf{w}_N^T$

Algorithm 7.9 The inference procedure for parsing novel sentences.

Input: An example $w_1, \ldots, w_n$
Output: A set of derivations $\mathcal{Y}_r$

$C \leftarrow \textsc{Parse}(\tilde{x}_i)$ ▷ Run parser
$\mathcal{Y} \leftarrow C[1, |\tilde{x}_i|]$ ▷ Extract output parses
$\mathcal{Y}_r \leftarrow \{\textsc{ResolveCoreferences}(y) \mid y \in \mathcal{Y}\}$ ▷ Resolve coreferences
return $\mathcal{Y}_r$

In the following chapter, we describe a series of experiments which examine the behaviour of the training algorithm and the resulting model. Depending on the level of detail of the evaluation, we will refer to the various outputs of the inference procedure: the highest-scoring meaning representation, the derivation that produces it, or even the complete parse chart.

Chapter 8.

Evaluation of Graph Algebraic CCG Grammars for Semantic Parsing

In the preceding chapters, we have described a complete pipeline for training a GA-CCG parser, including the augmentation of an AMR corpus with CCG derivations and token-node alignments, the induction of a GA-CCG lexicon from the corpus, the creation of delexicalised templates along with a lexeme dictionary, the training of a supertagger to provide template predictions, to the training of a parsing model which uses this lexicon to analyse novel sentences.

Throughout the description of this pipeline, validation experiments were performed with the goal of examining the plausibility of each component's output and finding good settings for the components' hyperparameters. However, the suitability of the system for the task of AMR parsing is determined by the interplay of all components, and an end-to-end evaluation is therefore needed to give meaningful results. This evaluation is the topic of this chapter.

We begin by observing the parser's performance in relation to a set of hyperparameters, including feature extraction, beam size, bootstrapping, and grammar rules (Section 8.1).

These experiments lead to the configuration of our final system, which is examined in detail in Section 8.2. We provide both a quantitative evaluation with a comparison to other AMR parsers, and an error analysis which gives insights into behaviours specific to the parser. We find that our parser performs similarly to other CCG-based approaches, but does not reach the higher benchmarks set by state-of-the-art systems.

8.1. Parser Tuning

As with any complex system, the GA-CCG parser permits many degrees of configuration. We single out a small set of parameters for further evaluation:

Stage	Parameter	Value
Lex. Induction	Grammar	`all`
	Alignments	`jamr+tamr+amr_ud+isi-vote1`
	maxItemCount	10,000
	maxUnalignedNodes	10
	maxCorefs	1
	ccgDerivations	50
	EM Iterations	100
Supertagging	Embeddings	ELMo
	LSTM layout	1 layer, 100 dimensions
	Syntactic Category Feature	enabled
	Max. predictions per token	20

Table 8.1.: Settings for lexicon induction and supertagging in the parser tuning experiments.

1. The *bootstrapping* learning schedule, as this setup is not commonly found in semantic parsing systems.
2. The *beam size* setting, as this setting directly influences the parser's ability to search for correct parses, while also strongly affecting the required processing time.
3. The *feature set*, since features are an important input to a linear learning system and feature extraction also makes up a significant amount of processing time.

Another important factor that determines the running time of the parser is sentence length. Figure 5.3 shows the sentence length distribution of the training corpus. To limit computational requirements and allow experimentation with other settings, we limit the length of training sentences to 25 tokens in the tuning experiments. In the final evaluation in Section 8.2, this limit is increased to 30 tokens.

8.1.1. Setup

To allow results to be comparable, we use a common baseline setting for the tuning experiments.

All runs use the same lexicon, which is induced with the settings determined in Sections 5.3 and 6.3.6, as summarized in Table 8.1.

Reported Metrics

In the experiments in this section, we report Smatch F1 scores on the `dev` set of the AMR 1.0 proxy corpus. The `test` set is used only for the final evaluation in Section 8.2. The reported validation score is computed after training the parser for ten iterations, or after the last iteration that has been completed within a time budget of seven days.

Training the parser is a compute-intensive process, and many of the parameters trade off parser accuracy against the amount of computation required. In some experiments, we therefore also report the time spent training the algorithm in hours, measured on the same system as the lexicon induction experiment in Section 5.3: A two-CPU, 16-core Intel Xeon E5-2630v3 compute node with 64 GB of RAM.

8.1.2. Feature Set

The parser computes a feature vector for every intermediate result in order to evaluate its score and rank it among other intermediate results (see Section 7.2). So far, we have not discussed how this feature vector is derived.

We divide the features that are included in our feature vector into several classes. In Section 8.1.6, the individual contribution of each feature class is evaluated. In all other experiments, all features are enabled.

We define the following feature classes:

Identity Features: This feature class counts every invocation of templates, lexemes, and lexical generators. It encompasses the following features:

- `lexeme` $i = 1$ for every time lexeme i is instantiated
- `template` $i = 1$ for every time template i is instantiated

Path Features: This feature class contains features for all paths of lengths up to 2 in the meaning representation. The intention is to allow the parsing algorithm to pick up on common, plausible relationships between concepts.

Path features are based on node labels (concepts or constants) and the roles of the edges connecting the nodes. Both types of labels can be transformed in various ways to form a concrete feature. We use the following notation to describe the various path features:

Let $n_1, n_2, \ldots$ be nodes and $e_1, e_2, \ldots$ edges, where each $e_i = (n_i, n_{i+1})$. The nodes can be either constant or variable nodes. Our features use the following derived properties:

- l_i is the *label* of n_i: if n_i is a constant node, it is the value of the constant; if it is a variable node, it is the concept instantiated by the variable. Features containing l_i are not triggered if n_i is a placeholder or coreference node.
- c_i is the *class* of n_i: Node labels can be categorised according to their rough function, which allows features to generalise across concrete labels. We define the following classes:
  - `[quoted]` for quoted constants, e.g. `"Iran"`
  - `[num]` for numeric constants, e.g. `1`
  - `[verbal]` for nodes that are linked to an OntoNotes frame, e.g. `sleep-01`
  - `[named-entity]` for named entity nodes such as `country`, `government-organization`, etc.
  - `[name]` for `name` nodes (which are part of named-entity constructions)
  - `[nominal]` for other nodes not linked to an OntoNotes frame, e.g. `boy`

  Features containing c_i are not triggered if n_i is a placeholder or coreference node.
- r_i^d is the *denumeralised role* of e_i. Digits are removed from all role labels except ARGi labels. For example, op1 is transformed to op, but ARG0 is left intact. This is because ARGi labels actually signify semantic differences whereas in op$i$ edges, the index is insignificant and depends only on word order.
- Node labels can also be ignored and replaced with `*`. This allows path features to also take edges into account that connect to placeholders or coreference nodes.

For every edge e connecting node n_1 to node n_2, the following features are triggered:

- `path(`l_1`,` r_1^d`,` l_2`)` $= 1$
- `path(`l_1`, *,` l_2`)` $= 1$
- `path(`c_1`,` r_1^d`,` l_2`)` $= 1$
- `path(`l_1`,` r_1^d`,` c_2`)` $= 1$
- `path(*,` r_1^d`,` l_2`)` $= 1$

- `path(`l_1`, `r_1^d`, *)` $= 1$

For every path e_1, e_2 connecting nodes n_1, n_2, n_3, the following features are triggered:

- `path(`l_1`, `r_1^d`, `l_2`, `r_2^d`, `l_3`)` $= 1$
- `path(*, `r_1^d`, *, `r_2^d`, *)` $= 1$
- `path(`l_1`, `r_1^d`, *, `r_2^d`, *)` $= 1$

Duplication Features: These features are triggered if elements of a meaning representation are duplicated, as this is usually not desired.

`dup-edge` features are triggered if two edges (either both incoming or both outgoing) sharing the same role are adjacent to a node.

- `dup-edge(`l_1`, `r_1`)` $= i$ is triggered for every non-placeholder, non-coreference, variable node that has more than one outgoing edge labelled r_1, where i is the number of such edges
- `dup-edge(*, `r_1`)` $= i$ is triggered exactly as above, but does not include the node label

`dup-neighbour` features are triggered if two nodes with the same label (either instantiated concept, or constant value) are adjacent to a node.

- `dup-neighbour(`l_1`, `l_2`)` $= i$ is triggered for every node n_1 which has more than one neighbour labelled l_2, where i is the number of the duplicate neighbours
- `dup-neighbour(*, *)` $= i$ is triggered exactly as above, but does not include the node labels

Token Features: We employ various features that relate the semantic content of a lexical entry to the tokens that it was selected for. This group includes several types of features: lexeme-lemma, template-lemma, pattern-token, and node-lemma cooccurences.

These features are triggered on the level of lexical selection, that is, for meaning representations that are output by GenLex. If an intermediate result is created by the application of a combinatory rule, it simply accumulates the features from its preceding lexical selections.

- `cooc(lemma=`λ`, lexeme=`i`)` $= 1$ where λ is the lemma of a word for which lexeme i has been instantiated
- `cooc(lemma=`λ`, template=`i`)` $= 1$ where λ is the lemma of a word for which template i has been instantiated
- `lex-wm-cooc(`λ`,` l`)` $= 1$ where λ is the lemma of a word and l the label of a node in the meaning representation instantiated for the word.

Syntax Features: This feature class relates the semantic root of a lexical item to its syntactic category. Like the token features, it is triggered for lexical selections.

- `syncat-root(`s`,` l`)` $= 1$ where s is the syntactic category of a lexical item and l the label of its root node.

Supertagger Confidence: This feature incorporates the confidence assigned to each template by the supertagger. While the available templates are already pruned to the supertagger's top-n list, the parser may still benefit from taking into account this confidence. Since feature values are added up across the several lexical selections that contribute to an intermediate result, we take the logarithm of the confidences to make them additive. The feature is defined as follows:

- `supertagger-confidence` $= \log c$ where c is the confidence assigned by a supertagger to a template.

CCG Supertagger Confidence: Another source of information about lexical selection lies in syntactic supervision. For this, we supply the confidence assigned to each lexical item's syntactic category to the parser as follows:

- `ccg-supertagger-confidence` $= \log c$ where c is the probability assigned to the synactic category of a lexical item by the EasyCCG supertagger.

Skeleton: Templates are already delexicalised, but only one label is removed from them at a time. However, the pure structure of a lexical entry could also contain some generalisable information. Skeleton features are based on such structures, in which all node labels are removed from a template. Each such structure is then mapped to an ID, so that the same structure can be recognised across sentences.

- `skeleton(`i`)` $= n$ where i is a skeleton ID and n is the number of templates instantiated at lexical nodes that are based on this skeleton.

Parameter	Value
Bootstrapping	1 iteration with oracle
Iterations	10 iterations without oracle
Grammar	full
Features	all
Beam Size	20
Max. Sentence Length	25

Table 8.2.: Baseline parser settings in the parser tuning experiments.

Run	Prec	Rec	F1
1	0.6912	0.5540	0.6150
2	0.7022	0.5637	0.6254
3	0.6993	0.5562	0.6196

Table 8.3.: Smatch F1 scores on the Proxy dev set across three runs of the baseline configuration.

8.1.3. Baseline Settings

As a foundation for our experiments regarding parser configuration, we define a set of baseline settings, outlined in Table 8.2. Throughout this section, our procedure will be to vary each of the settings in turn and compare the scores achieved by the parser.

In each experiment, we report the Smatch F1 score achieved on the `dev` section of the AMR 1.0 Proxy subcorpus after the final training iteration.

Before turning to tuning experiments, we conduct a first baseline evaluation to ensure that the parser converges reliably. We train the parser three times using the baseline settings and evaluate after each iteration. This helps estimate the random variation caused by the training process.[1] Table 8.3 shows a gap of less than 0.01 F1 between the best and worst achieved result.

Figure 8.1 shows the validation F1 scores after each iteration of training across the three runs. It can be seen that the runs behave similarly, with differences in the range of up to 0.015 Smatch F1 points. All three systems reach a plateau within

[1] As we initialise feature weights with zero, random variation in our system is caused by data shuffling.

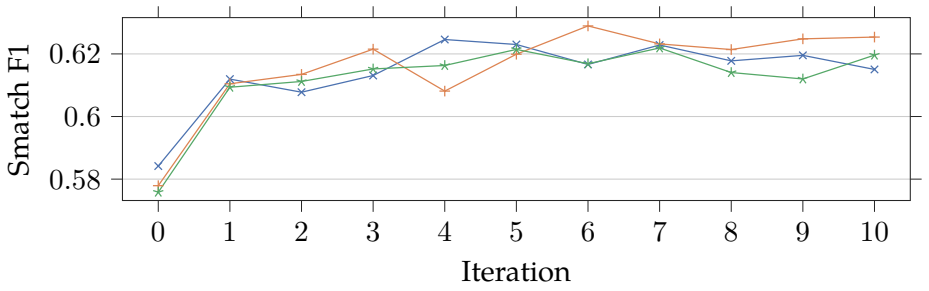


Figure 8.1.: Per-iteration validation scores of three training runs with the baseline configuration (0 is the bootstrapping iteration).

four to six iterations, which validates our evaluation strategy of using the Smatch F1 score obtained after the tenth regular training iteration.

8.1.4. Bootstrapping

Many semantic parsing systems rely on forced decoding approaches to solve the problem that correct derivations are not contained in semantic parsing training data (see Artzi, K. Lee and L. S. Zettlemoyer (2015), among others). In such an approach, an oracle is defined which makes use of training-data annotation to produce "good" derivations (that is, derivations which lead to the correct meaning representation). This means that a strong oracle is required which produces good parses for most sentences, as any sentence for which the oracle fails would effectively be excluded from training.

In the case of GA-CCG, such an oracle is difficult to define due to the fact that GA-CCG covers only a subset of the constructions found in the AMR corpus. In Section 5.3.6, it was found that only about 74 % of tokens in the AMR 1.0 Proxy section's training set are covered by the induced grammar, implying that many meaning representations in the corpus cannot be reproduced exactly. This in turn means that the search for forced derivations cannot rely on hard exclusion criteria but must be somewhat error-tolerant, exacerbating the associated search problem.

In Section 7.3, we introduced such an oracle, which uses a heuristic scoring function to drive the CKY parsing algorithm, in place of the learned scoring function of the linear model. This oracle is weak in the sense that it cannot be expected to output a derivation of the correct meaning representation for every example. However, it does attempt to derive a meaning representation that is similar to the correct one. Importantly, we can expect it to perform better than an untrained

Bootstrapping	Prec	Rec	F1
all	0.6853	0.4319	0.5298
bootstrap	**0.6956**	**0.5575**	**0.6189**
none	0.6953	0.5545	0.617

Table 8.4.: Smatch F1 scores on the Proxy dev set achieved using various bootstrapping schedules.

parsing model would.

Since we employ a cost-sensitive learning algorithm which takes the quality of the generated meaning representations into account in the computation of a parser update, it is possible to simply add the best oracle-generated derivation to the output set of the parser. If this derivation is better than all parses produced by the learned model, it is included in the update computation. In contrast, if the learned parser is able to produce an output which surpasses the oracle output, that output is selected instead. This behaviour follows naturally from the definition of the cost-sensitive learning algorithm in Section 7.2.2.

To determine to what extent the inclusion of an oracle is helpful to the training process, we experiment with training schedules which include oracle parses in the set of hypotheses in no, one, or all training iterations. The experimental conditions are defined as follows:

- **all**: 11 iterations, all with oracle enabled
- **bootstrap**: 1 iteration with oracle, 10 iterations without
- **none**: 11 iterations without oracle

The **bootstrap** configuration is thus identical to the schedule used in the other experiments in this section.

The results in Table 8.4 show that the effect of a single bootstrapping iteration (the **bootstrap** condition) on final performance is marginal compared to training without an oracle (**none**). At the same time, using an oracle in all iterations (**all**) is clearly not an effective way to train the parser. Although the oracle achieves an F1 score of 0.77 on the training data, the parser is not able to learn from it effectively. Apparently, the distribution of derivations preferred by the oracle cannot be fully represented using a trained parser model.

In contrast to the system of Artzi, K. Lee and L. S. Zettlemoyer (2015), which uses forced decoding, and semantic parsers such as the system of Berant and Liang (2015) which uses an oracle to train an agenda-based parser, our parser therefore does not require a separate generator of good derivations.

8.1.5. Beam Size

Due to the high asymptotic complexity of CKY parsing, we perform inexact decoding with beam search. The beam size is thus an important parameter which controls both the ability to search for the highest-scored full derivation, and the amount of computational resources needed. The model's scoring function does not fully decompose to the level of individual lexical entries, and therefore the optimal final result might depend on some intermediate result with a relatively low score. With a small beam size, this result would not be included in the CKY algorithm's search space, causing a search error. At the same time, a large beam size requires a larger amount of intermediate results to be enumerated, increasing the amount of computation required for graph composition and feature extraction.

In addition, beam search plays an especially important role in the context of cost-sensitive learning. Our learning algorithm depends on the parser finding high-quality results even if they are not favoured by the current model, in order to push those results to the top and thus improve the model parameters. The larger the beam size, the larger the chance is for the parser to find a high-quality result with a low model score.

Our goal is therefore to find a "sweet spot" setting for the beam size, which is large enough to prevent a significant amount of search errors, but which does not exceed a reasonable computational budget. As in the other experiments in this chapter, we define this budget as seven days of computation on a 24-core compute node.

Results

We train the parser using beam sizes in the range from 5 to 30. The results are shown in Table 8.5. Training is performed for one bootstrapping iteration plus ten regular iterations. For each beam size setting, we perform a validation run after every iteration and record the highest achieved Smatch F1 score.

The results show that an increased beam size leads to improved results up to a setting of 25, while the beam size of 30 brings no further improvement, although it also does not cause a noticeable deterioration.

Beam Size	Prec	Rec	F1	Training Time (h:m)
5	**0.7186**	0.3170	0.4399	8:13
10	0.6948	0.5025	0.5832	25:47
15	0.6920	0.5457	0.6102	47:42
20	0.6962	0.5565	0.6186	78:11
25	0.6975	0.5636	0.6235	114:37
30	0.6972	**0.5670**	**0.6254**	113:16

Table 8.5.: Smatch F1 scores and training times on the Proxy dev set achieved using various beam sizes.

8.1.6. Features

In Section 8.1.2, we described a diverse set of features that are assigned to groups based on the information they incorporate. To measure the contribution of each feature group, we run an experiment starting with a minimal feature set including only the identities of lexical items and lexical item generators. This feature set is extended in each condition by activating another feature group, so that every condition builds on the previous one.

The Smatch scores achieved in the experiment are shown in Table 8.6. For most feature groups, the results show a clear contribution. The exceptions are *duplicate*, *ccgtagger* and *skeleton*.

In the case of *skeleton* features, the effect is easily explained because template IDs already provide a feature that abstracts away from lexical content – although in templates, only a single node is delexicalised whereas in skeletons, all lexical content is removed. Still, the information provided by skeleton features seems to be subsumed by other feature classes.

That the CCG supertagger does not provide a performance gain could be because the template supertagger already takes CCG syntactic categories into account when predicting templates, and since each template is associated with a syntactic category, it indirectly assigns probabilities to them. There is thus little room left for considering the predictions of the purely syntactic supertagger.

The small effect of *duplicate* features is more difficult to explain, but could be because other features are already effective in controlling cases of duplicate edges or neighbours, or because such cases are rare enough to have a small effect on the Smatch score.

Feature Set	Prec	Rec	F1
identities	0.5814	0.3051	0.4002
+path	0.6161	0.4798	0.5395
+duplicate	0.6165	0.4858	0.5434
+tokens	0.6411	0.5105	0.5684
+syn	0.6686	0.5335	0.5934
+supertagger	0.6982	0.5539	0.6177
+ccgtagger	0.6970	0.5541	0.6174
+skeleton	**0.6985**	**0.5605**	**0.6219**

Table 8.6.: Smatch F1 scores on the Proxy validation set achieved using various feature sets. In each row, the feature set of the previous row is extended by a group of features from Section 8.1.2. The reported scores are computed after 5 iterations of training plus one bootstrapping iteration.

Stage	Parameter	Value
Lex. Induction	ccgDerivations	10
	EM Iterations	20

Table 8.7.: Modified settings for lexicon induction and supertagging in the experiment on rule sets.

8.1.7. Rule Sets

The GA-CCG grammar rules are the distinguishing feature of our parsing system and therefore particularly interesting to examine. In Section 5.3.5, the effect of the grammar rules on coverage has been examined, justifying the use of the **all** grammar.

However, induction coverage does not necessarily translate to better parser performance, since the parser operates under more severe computational constraints due to the complexity of CKY parsing and the beam size limitation.

To show the effect of various grammars on the parser, we train parsers for each of the rule sets defined in Tables 4.1 and 4.2.

Due to the need to run lexicon induction, filtering, and supertagging steps for each configuration, this experiment is configured to reduce its runtime compared to other experiments in this section. Its results are therefore not directly comparable

Grammar	Prec	Rec	F1
all	0.7029	0.5531	0.6191
base	0.6997	0.5552	0.6191
no-ignore	0.7034	0.5570	0.6217
no-modify	**0.7023**	**0.5620**	**0.6244**
no-tr	0.7031	0.5488	0.6165
unrestricted-a	0.6920	0.5521	0.6142
unrestricted-all	0.6778	0.5280	0.5936

Table 8.8.: Best validation score achieved using different GA-CCG grammars.

to the other experiments. The settings modified for this experiment are described in Table 8.7; all other settings are left as shown in Table 8.1.

Table 8.7 shows the results of the experiment. Interestingly, the **all** grammar fails to outperform the **base** and **unrestricted-a** grammars, both of which are much simpler. At first sight, this casts doubt on our strategy of adding rules for specific linguistic phenomena. If a minimal grammar such as **unrestricted-a** is able to do the job, it is unnecessary to bother with complex grammar rules.

However, the results for the three grammars **no-modify** and **no-ignore** suggest that the picture is more complex, as each of them achieves a higher score than **base**. This suggests the interpretation that the added rules do have a positive effect individually, but not in combination. A plausible explanation for this effect is the increase in search space caused by these grammars: in situations where extra rules apply, the parser now has more rules to choose from, which increases the likelihood of search errors. While some additional rules have a positive effect on their own, this advantage disappears when they are combined.

8.1.8. Amount of Training Data

For all machine learning systems, the amount of available training data is an important determining factor for the resulting model's performance. To understand how our parser behaves in this regard, we train it on reduced training data sets. In each run, we randomly select a certain percentage of sentences from the training data set, and use only these for training.

We train parsers for ten percent increments from 10 % to 100 % of the training data set, but also include 1 % as a very-low-data condition. The results are given in Table 8.9 and plotted in Figure 8.2.

Training Data %	Prec	Rec	F1
1	0.4261	0.2487	0.3141
10	0.5921	0.4396	0.5046
20	0.6338	0.4905	0.5530
30	0.6593	0.5025	0.5703
40	0.6483	0.5142	0.5735
50	0.6744	0.5303	0.5937
60	0.6824	0.5333	0.5987
70	0.6946	0.5468	0.6119
80	0.6961	0.5468	0.6125
90	0.694	**0.5611**	**0.6205**
100	**0.7024**	0.5553	0.6202

Table 8.9.: Smatch scores on the Proxy dev set using various amounts of training data.

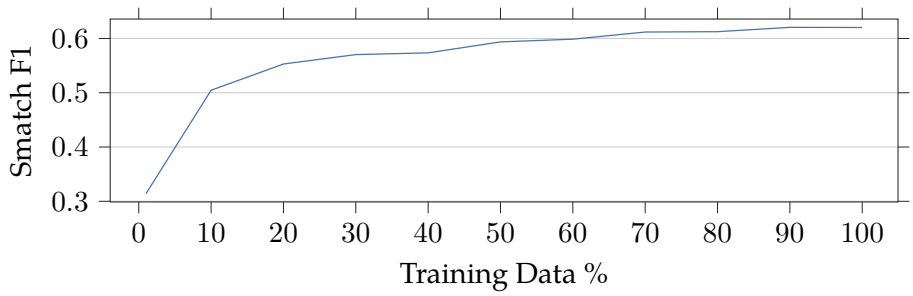


Figure 8.2.: Smatch F1 scores achieved using a given percentage of the Proxy training data.

As expected, the achieved Smatch score rises with the amount of training data. The plot in Figure 8.2 shows that the parser approaches its final score relatively early: the second 50 % of training data only contribute less than 0.03 F1 score.

Apparently, the learning algorithm is able to make sense out of small data sets, reaching 0.55 F1 score with just 20 % of the training data (equivalent to around 1,300 sentences). One likely explanation for this effect is the linguistic knowledge supplied to the parser particularly by the syntactic CCG parser, which is not affected by the training data restriction in this experiment. Likewise, the GA-CCG grammar rules themselves also encode linguistic information. However, the training algorithm does not access the syntactic parser and needs to encode this information in the parser model's parameters, which it appears to succeed at.

On the other hand, model performance appears to plateau at 100 % of training data, suggesting that adding more training data is unlikely to result in a large improvement in performance.

8.2. Final System Evaluation

In the experiments in the previous section, we have examined the sensitivity of the parser regarding each individual hyperparameter. Now we create the final parser configuration by setting each hyperparameter so that parser performance is maximised. In particular, we raise the beam size to 25 and the maximum training sentence length to 30. Since the **no-modify** rule set achieved the strongest performance, we use it for the final evaluation. We also reduce the number of EM iterations to 20 since EM makes up a substantial part of the computation needed to train the parser, and no performance drop was noticeable in conditions with this setting. Since the experiments in the previous section showed good convergence well before the full training iterations, we train the parser for a full seven days and use the output of the last completed iteration. In this time, it completes one bootstrapping iteration and six regular iterations. Table 8.10 summarises the configuration.

We evaluate this system quantitatively and qualitatively: Firstly, for the only time within the scope of this thesis, we run an evaluation on the `test` set of the AMR 1.0 Proxy corpus. This allows us to compare the parser's results to other parsers evaluated on the same dataset. Secondly, we conduct an error analysis on the output of the parser, including an attempt at error diagnosis, which allows us to characterise the parser's peculiarities and identify areas that need improvement.

In this section, too, our experiments are limited to the Proxy subset of the AMR 1.0 corpus. Our rationale for this limitation is twofold: Firstly, the CCG-based AMR

Stage	Parameter	Value
Lex. Induction	Grammar	`all`
	Alignments	`jamr+tamr+amr_ud+isi-vote1`
	maxItemCount	10 000
	maxUnalignedNodes	10
	maxCorefs	1
	ccgDerivations	50
	EM Iterations	20
Supertagging	Embeddings	ELMo
	LSTM layout	1 layer, 100 dimensions
	Syntactic Category Feature	enabled
	Max. predictions per token	20
Parser	Bootstrapping	1 iteration with oracle
	Iterations	6 iterations without oracle
	Grammar	no-modify
	Features	all
	Beam Size	25
	Max. Sentence Length	30

Table 8.10.: Settings for lexicon induction and supertagging in the parser tuning experiments.

System	Precision	Recall	F1
gramr	0.670	0.625	0.647
Misra and Artzi (2016)	0.681	0.642	0.661
Artzi, K. Lee and L. S. Zettlemoyer (2015)	0.668	0.657	0.663
Liu et al. (2018)	–	–	**0.733**
Ballesteros and Al-Onaizan (2017)	–	–	0.69
Goodman, Vlachos and Naradowsky (2016)	0.68	0.73	0.70
Zhou et al. (2016)	0.73	0.68	0.71
Wang, Xue and Pradhan (2015b)	0.720	0.670	0.700
Flanigan et al. (2014)	0.52	0.66	0.58

Table 8.11.: A comparison of our system (gramr) to other systems evaluated on AMR 1.0 Proxy in terms of Smatch precision, recall, and F1 score. The first group are CCG-based systems. The second group comprises systems that follow different approaches, including the best currently reported score on the dataset.

parsers which are most interesting to compare against are evaluated on this corpus. Secondly, evaluation on the newer and larger AMR 2.0 and AMR 3.0 datasets would require substantial additional engineering effort to reduce memory consumption and computational requirements, while our results already strongly suggest that our system is not competitive with newer AMR parsers that are evaluated on these datasets. We therefore focus on the experimental setting which we deem the most insightful for our system.

8.2.1. Quantitative Evaluation

After one bootstrapping iteration and six iterations of training, the parser achieves a Smatch F1 score of 0.6295 on the `dev` set, which aligns with the scores reported for experiments in the previous section. It fails to exceed the score of 0.6339 achieved with the **no-modify** grammar in Section 8.1.7, but is within the range of expected variation.

To obtain a final score on the `test` set, we parse all sentences except a single sentence of length 73: due to the nonlinear runtime complexity of the CKY algorithm, parsing this sentence takes up a disproportionate amount of computation time. The excluded sentence is nonetheless included in the final score with zero correct edges.

The total Smatch F1 score achieved by our parser on the test set is 0.647. Table 8.11 shows the comparison of this result to a number of other semantic parsing systems with published results on the AMR 1.0 Proxy dataset. Our system is called gramr, short for "Grammar-based AMR Parser". The score is close to, but does not surpass the scores achieved by other CCG-based parsers by Artzi, K. Lee and L. S. Zettlemoyer (2015) and Misra and Artzi (2016), as shown in Table 8.11. As the table also shows, more recent systems have reached F1 scores of up to 0.733. The most recent AMR parsing systems do not report data on the AMR 1.0 corpus and therefore do not occur in Table 8.11.

Our results therefore reinforce the impression that there is a performance gap between CCG-based approaches and other approaches to AMR parsing, such as the Stack-LSTM-driven transition-based parser of Ballesteros and Al-Onaizan (2017) and Liu et al. (2018), or the maximum spanning tree-based parser of Zhou et al. (2016).

8.2.2. Error Analysis

To gain a more detailed picture of the operation of the parsing pipeline, we conduct an in-depth, two-step error analysis on a set of 20 randomly sampled sentences from the `dev` set. For each of these sentences, we manually compare the AMR produced by the parser to the annotated gold-standard AMR, and assign an error class for each difference between the two representations.

In a second step, we look for errors on the syntactic-semantic level: we examine the GA-CCG derivation created by the parser, and look for the underlying cause of each graph-level error. The resulting overview should give us a clearer view of areas within the parser that need improvement. Both types of errors are summarised in Table 8.12. For both graph-level and syntactic-semantic errors, we count the number of sentences that are affected by the error. For graph-level errors, we also count the total number of occurrences across all graphs. Syntactic-semantic errors are counted per token.

Error Classes

We define the following graph-level error classes:

- **focus:** the focus (root) of the graph is assigned to the wrong node.
- **label:** a node is labelled incorrectly. We define this error class broadly and apply this error class to both mislabelled constant nodes and to variable nodes

which instantiate the wrong concept.

- **missingc:** a node or (connected) subgraph from the gold-standard graph is not represented in the parser output.
- **extrac:** a node or subgraph is in the parser output but has no representation in the gold standard graph.
- **decomp:** an entity should have been represented in a decompositional manner, but was represented as a single node instead.
- **coref:** an entity should have been co-referenced, but a separate node representing the same entity was created instead.
- **link:** an expected edge was present, but between the wrong nodes (that is, the node was linked to a wrong node).
- **link+:** an edge was present in the parsed AMR that was not present in the gold-standard AMR, and the error cannot be interpreted as **link**.
- **link-:** an edge from the gold-standard AMR was missing in the parsed AMR, and the error cannot be interpreted as **link**.
- **direction:** an expected edge was present, but with reversed direction.
- **role:** an expected edge was present, but assigned the wrong role.

Each of these errors is rooted in a parser decision. In the root-cause analysis, we attempt to identify a decision that would have to be changed to eliminate the error. Importantly, we do not evaluate each CCG derivation as a whole, but only identify those errors that can be attributed to each graph-level error. We identify the following syntactic-semantic error causes:

- **lex:** the selected lexical entry is inadequate.
- **scope:** a conjunction was scoped inappropriately.
- **attach:** a prepositional phrase was attached to the wrong head.
- **compound:** a compound was bracketed inappropriately.
- **control:** a control structure was scoped inappropriately.

Error Class	Count	Sentences
focus	1	1
label	28	17
missingc	21	10
extrac	4	4
decomp	6	5
coref	5	4
link	28	12
link+	1	1
link-	8	6
direction	3	3
role	31	16
lex	88	19
scope	2	2
attach	7	4
compound	1	1
control	1	1
grammar	1	1
anno	1	1

Table 8.12.: Counts for each of the error classes across the 20 sentences included in the error analysis. The first group are graph-level errors, the second group comprises syntactic-semantic errors.

- **grammar:** the correct graph could not be constructed due to an intrinsic limitation of GA-CCG.
- **anno:** the gold-standard annotation is incorrect, leading to the observed graph-level error.

Results

Table 8.12 gives an overview of the errors that were identified in the sample. Counting the errors is not entirely straightforward since errors depend on assumptions about the correspondence between parts of the annotated and the parsed graphs. They can also involve several graph elements, and may also be interdependent. The following fundamental considerations were applied when counting the errors:

- Graph-level errors are counted so as to maximise the overlap between the gold-standard and the parsed graph, similarly to how Smatch scores are computed.
- Errors pertaining to more than one node or edge, such as missingc and extrac, are counted once for every connected subgraph that is affected.
- Syntax-level errors are counted per affected token.
- In the case of scoping or attachment errors, the incorrectly scoped token (such as a conjunction or preposition) is counted.

Apart from the total error occurrences, Table 8.12 also contains the number of sentences with at least one occurrence of each error class.

The error analysis reveals several interesting patterns.

- Mislabellings of both nodes (*label*) and edges (*role*) are frequent, occurring in at least 80 % of the examined sentences. AMR knows multiple kinds of representation for any given concepts (with or without sense index, as a quoted constant, etc.), and the supertagger appears to have difficulties distinguishing between them. Distinguishing between roles is similarly difficult: core (ARGn) roles depend on the governing event and therefore require lexical knowledge. At the same time, prepositions are polysemous and may invoke different non-core roles such as mod, location, instrument, etc. While the parser contains features to model preferences for these choices, the supertagger preselects the lexical items that are available to the parser.
- Incorrect connections (*link*), where nodes are connected that should not be, are somewhat less frequent but still occur in the majority of sentences. This error class is interesting because it indicates that the parser fails to infer the "who did what to whom" which forms the backbone of the semantic structure.
- The parser is biased towards omitting content, both nodes or subgraphs (*missingc*) and individual edges (*link-*). Only rarely does it invent elements (*extrac*, *link+*). One explanation for this bias is that the parser can easily employ templates that contribute no content (such as"identity" graphs with a single placeholder) to fill a syntactic gap in the derivation.
- The selection of lexical items is by far the most important source of errors, occurring in all but one sentences. This means that even most structural errors in the graph representations could be avoided by improving lexical selection.

The error analysis paints a detailed picture of the parser's error profile, showing clear potentials for future engineering improvements.

8.2.3. Conclusion

In the preceding chapters, we have described a pipeline of tools to tackle the problem of Abstract Meaning Representation parsing. This chapter is dedicated to evaluating the resulting semantic parsing system, which we call *gramr*.

In Section 8.2.1, the parser's performance is evaluated quantitatively and compared to other semantic parsing systems, finding that our system is almost on par with other CCG-based systems in terms of Smatch F1-performance. However, this class of systems is not representative of the state of the art for AMR parsing overall, leaving a wide gap of at more than 0.07 Smatch F1 points to the best reported system on the same dataset.

A possible interpretation for this result is that the structural constraints of GA-CCG limit AMR parsing. GA-CCG enforces a structure on AMR derivations that is sometimes at odds with the constructions of AMR, and is unable to represent some constructions, as has been examined in Chapter 4.

Other CCG-based parsers may however be less affected by this effect, since they use a different, λ-calculus-based construction mechanism (Artzi, K. Lee and L. S. Zettlemoyer 2015; Misra and Artzi 2016), even though the authors do not examine its coverage.

In contrast, transition-based or graph-based parsers can more easily be engineered to achieve a high theoretical coverage. They also face fewer formal constraints on their output. While the rigidity of a formal framework such as CCG may serve the parser by leading it to higher-quality outputs, this positive effect seems to be outweighed by the drawbacks.

The error analysis in Section 8.2.2 reveals that the vast majority of errors can be traced to the selection of lexical entries. Since gramr uses a pipeline of lexicon induction, supertagging, and parsing, the parser has no chance of producing adequate parses if the supertagger does not predict appropriate lexical entries. In turn, if lexicon induction does not produce the appropriate lexical entry for a token, the supertagger stands no chance of predicting it. The error analysis therefore points to error propagation as a main weakness of the pipelined architecture of gramr.

Again, this argument is less applicable to the other CCG-based systems, since the lexicon induction algorithm used by Artzi, K. Lee and L. S. Zettlemoyer (2015) and Misra and Artzi (2016) is invoked in lockstep with the training of the parser. The lexicon can thus be amended at any time during training.

An important motivation for the design of the pipelined architecture in gramr was to reduce computational requirements. The supertagging step is necessary because of the high complexity of the CKY algorithm, which is not able to cope with the selection of templates from the full lexicon. A less complex parsing algorithm might be able to consider more templates for each token and reduce the problem of error propagation. However, Misra and Artzi (ibid.) fail to achieve an improved Smatch score with a linear-time shift-reduce parsing algorithm.

In summary, CCG-based parsers are subject to certain constraints that other types of semantic parsers are less affected by. Although CCG provides a strong basis for parsing mor logic-based meaning representations, from the research so far it appears that in the case of AMR parsing, the rigidity of CCG outweighs the benefits gained from using a formalism geared towards linguistically plausible semantic construction.

Chapter 9.

Conclusion

The preceding chapters describe the design and implementation of a semantic parsing system, from its theoretical underpinnings to its empirical evaluation. The approach taken in this thesis is opinionated and somewhat in opposition to recent developments in the field of natural language processing, in that we have taken a modular pipelined approach instead of end-to-end large-scale training, and by emphasizing symbolic over continuous representations.

The motivation behind these design choices was to explore the utility of CCG as a foundation for parsing an unrelated semantic formalism, and to work towards a semantic parsing pipeline that could leverage the increased transparency provided by a modular, grammar-based approach. In this context, this thesis can be considered a proof-of-concept of our approach. It sheds light on the complexity and challenges remaining to be tackled.

In this chapter, we first recapitulate the main findings of this thesis. Then, we disscuss these findings in the broader context of semantic parsing research and applications. Finally, we provide an outlook of possible followup activities, ranging from engineering challenges to new areas of application.

9.1. Summary

In this thesis, we describe and evaluate an approach to the semantic parsing of Abstract Meaning Representations (AMRs). The scope of this work includes the design of a grammar to model the construction of AMRs, the implementation of algorithms to induce lexica for this grammar and to train a statistical parser which applies them, and finally a round of evaluations examining the behaviour and performance of the resulting parser.

Chapter 4 describes Graph Algebraic Combinatory Categorial Grammar (GA-CCG), a new formalism based on Combinatory Categorial Grammar (CCG), which

is tailored to the construction of AMRs. The chapter starts with the definition of GA-CCG s*-graphs, which are the semantic building blocks of the grammar, and graph algebraic rules which allow the construction of complete meaning representations from them. Then, GA-CCG is defined by equipping the combinators, which define the syntactic-semantic operations of CCG, with graph-algebraic interpretations. The idea of using graph-algebraic operations for AMR construction is not new (Koller 2015), but its application to CCG is.

Since the aim is for GA-CCG to employ a limited set of rules which are easy to understand and implement, it is focused on supporting linguistic constructions that are frequent in AMR corpora. Chapter 4 therefore motivates every GA-CCG rule by demonstrating how it supports common constructions, but also includes examples of unsupported constructions.

Chapter 5 is concerned with deriving a GA-CCG lexicon from an AMR-annotated corpus. To this end, it discusses algorithms to recursively break down the sentential AMRs contained in the corpus into fragments that represent the meanings of individual words. As there is no gold-standard data for this association, the algorithm is essentially a constrained brute-force search: it uses a CCG derivation generated by an off-the-shelf syntax parser as a backbone and annotates it with graph fragments that conform to alignments generated by specialised AMR alignment tools. Several other constraints and heuristics help manage the algorithm's run time and the size of the generated lexicon.

The chapter also includes an empirical validation of the algorithms. In particular, we examine the coverage of the induced lexica. Due to the heuristic nature of the induction algorithm, as well as limitations posed by the postulated GA-CCG rules, the induction algorithm cannot generate a lexicon that is capable of reproducing every observed meaning representation. However, we show that it covers a large percentage of the overall tokens contained in the AMR corpus.

In Chapter 6, tools for processing the induced lexicon are discussed. Since the output of the lexicon induction algorithm both contains implausible lexical items and is tied to the concrete words observed in the corpus, its usefulness for semantic parsing can be improved by combining lexical generalisation with filtering. This creates word-independent lexical entries and removes entries that are not required to describe the corpus. However, lexical generalisation exacerbates the problem of lexical item selection. To relieve the parsing algorithm of this problem, we employ a neural network-based supertagger to predict the most likely lexical entries for each word.

As a key result, the lexical generalisation and filtering procedure is able to reduce the lexicon from hundreds of thousands to mere thousands of entries without loss

of coverage, demonstrating that automatically induced lexica do not have to be orders of magnitude larger than manually created ones. This allows us to inspect the lexicon and confirm that the entries contained in it are plausible.

Training results for the supertagger show that despite the relatively large output space of several thousand lexical entries, and the fact that it has to deal with gaps in its training data created by the lexical induction algorithm, it is able to predict the fitting lexical entry with high accuracy.

Chapter 7 describes how a statistical parser for GA-CCG can be constructed and trained. While the core of the parser is a fairly simple linear model, training it requires additional tricks: we cover an adapted cost-sensitive perceptron algorithm, which helps train the parser even if it cannot generate a correct output, as well as an oracle heuristic to bootstrap the training process.

In Chapter 8, the parser is evaluated in a series of experiments, with some surprising results. The evaluation shows that not all of the GA-CCG rules defined in Chapter 4 actually improve end-to-end parser performance because even though they lead to improved grammar coverage, they also increase the number of possibilities the parser has to consider. Also, even though an oracle-based bootstrapping schedule has been considered during parser development, it does not improve end-to-end performance.

In the end-to-end evaluation, the parser is shown to perform comparably with other CCG-based AMR parsers, and thus considerably worse than more current parsers based on other techniques. An error analysis of the parser's output, which includes the syntactic-semantic derivations inferred by the parser, determines that the parser is biased towards precision in favour of recall, tending to omit elements of meaning representations. It also reveals that the selection of lexical items is a crucial bottleneck, contributing to the vast majority of observed errors.

9.2. Discussion

The semantic parsing system described in this thesis serves as a proof-of-concept implementation for GA-CCG-based semantic parsing. Importantly, we show that our system achieves a similar end-to-end performance as the other CCG-based AMR parsers by Artzi, K. Lee and L. S. Zettlemoyer (2015) and Misra and Artzi (2016), with a gap of less than two Smatch F1 percentage points.

However, although our parser involves a significant amount of engineering, it fails to improve upon these prior results. Is this indicative of engineering deficits in our system, or are there fundamental limits for applying CCG to AMR parsing?

Our experiments shed some light on this question and reveal that both aspects play a certain role.

1. The entire pipeline suffers significantly from error propagation. At each step, uncertainties in the form of noisy incomplete results are introduced, and subsequent steps have to deal with this. For instance:

   - Gaps or errors in word-to-node alignments produced by external aligners can lead to erroneous or missing lexical entries.
   - Similarly, incongruities between the modelling of certain constructions in CCG and AMR can lead to failures during lexicon induction, and thus to missing lexical entries.
   - This in turn leads to incomplete training data being available to the supertagger, limiting the quality of its predictions.
   - Suboptimal supertagger predictions limit the parser by making favourable derivations unavailable.

   In principle, these problems could be reduced through additional engineering effort. For example, using a faster, linear-time parsing algorithm could allow the parser to consider more lexical entries for each token, reducing the significance of the supertagging bottleneck.

2. The connection of the unrelated formalisms of AMR and CCG established in this thesis is imperfect since there are important grammatical constructions that cannot be modelled with GA-CCG. GA-CCG operators and rules could be extended to treat them better, and in particular, Blodgett and Schneider (2019) have proposed an alternative set of rules to better handle certain phenomena which could be implemented in our parser. However, as our experiments showed, modifications to the grammar interact with the computational requirements of the parser. If the search space is increased by adding new grammar rules, this may lead to the parser not being able to search it as effectively, and could thus neutralise the expected improvements. Ultimately, even with improved grammar rules, it seems unlikely that GA-CCG can be extended in such a way that the entirety of the AMR corpus is covered, given that AMR sometimes offers several acceptable representations and the annotations are not always consistent in this respect, and that AMR relies on non-compositional representations for some phenomena.

3. Some potential engineering improvements are outside the scope of this thesis. This includes improvements of the parser model, where significant improvements might be expected from using, for example, a neural network-based model for scoring, and where further experimentation with alternative parsing algorithms, such as shift-reduce algorithms, would be possible. Similarly, lexicon induction might be improved to reduce reliance on external tools and eliminate computational difficulties associated with brute force search, for example by designing an algorithm based on Bayesian statistics. All of these options have not been explored here due to time constraints and to maintain a focus on exploring the potential of GA-CCG grammars.

In summary, it appears that the GA-CCG approach adds significant engineering complexity while contributing little to benchmark end-to-end performance. While somewhat disappointing, this result is not entirely unexpected as it mirrors developments in other areas of natural language processing, such as machine translation, where systems using neural representations and end-to-end training have displayed better performance than methods relying on linear models, symbolic representations, and rule systems.

It should be kept in mind that this judgment relates purely to Smatch F1 performance on the standardised AMR 1.0 benchmark. While benchmark tasks are important to measure raw algorithmic performance, they do not necessarily represent real-world scenarios. The design and architecture of our system have properties that can be favourable under certain circumstances, and which could be explored in a more realistic setting.

9.3. Outlook

While standardised tasks focus on comparing systems under tightly controlled circumstances, real-world application scenarios are diverse and accompanied by many practical challenges, in addition to the engineering task itself. Real projects often encounter a bootstrapping problem, where task-specific training data does not exist initially and must be created as the project progresses. They may also face limited availability of personnel and computational resources.

In such an environment, systems that can work with limited amounts of data and computation are advantageous. As we have shown in Section 8.1.8, the gramr parser is able to produce useful analyses with a few hundreds of training examples. Its modular, pipelined architecture also makes it adaptable to the concrete setting

of the application. In particular, the grammar can be edited manually in order to further speed up the bootstrapping phase where little training data is available. The parser also produces rich output in the form of GA-CCG derivations, which eases the debugging of the toolchain.

At the same time, some of gramr's limitations can potentially be avoided in a concrete application scenario. For example, when collecting training data for a concrete task, AMRs can be limited to structures that GA-CCG can handle, eliminating an important source of errors.

Taken together, these findings open a perspective for gramr as a construction kit for lightweight AMR-based semantic parsing applications. Using AMR to encode application-specific semantics, and building custom AMR parsers, could be one path towards natural language-based human-computer interfaces that require deep semantic understanding, such as complex question answering systems, natural language database interfaces, analytics systems, and similar applications. For future work, this direction seems well worth investigating.

Appendix A.

Abstract

In this dissertation, I develop a new approach to semantic parsing with graph meaning representations. This approach is implemented in the form of a parser that translates English sentences into Abstract Meaning Representations: graph structures which capture information on entities and events described in the sentence. It is differentiated from other approaches by using a simplified form of Combinatory Categorial Grammar equipped with graph-algebraic operators tailored to the construction of graph meaning representations.

The semantic parsing system described in this thesis consists of a pipeline comprising lexicon induction, delexicalisation, filtering, supertagging, and parsing. Due to the simplified nature of the grammar, the parser works with a compressed, human-readable lexicon, while producing linguistically interpretable derivations of sentence meanings. At the same time, it achieves a Smatch performance comparable to other Combinatory Categorial Grammar-based approaches to Abstract Meaning Representation parsing.

Appendix B.

Kurzfassung

In dieser Dissertation wird ein neuer Ansatz für das semantische Parsing in graphenbasierte Bedeutungsrepräsentationen entwickelt. Dieser Ansatz wird in Form eines Parsers implementiert, welcher englische Sätze in Abstract Meaning Representations übersetzt: Graphenstrukturen, welche Informationen über die in einem Satz beschriebenen Ereignisse und Entitäten abbilden. Das Verfahren unterscheidet sich von ähnlichen Ansätzen, indem eine vereinfachte Form der Combinatory Categorial Grammar verwendet wird und diese mit Graph-algebraischen Operatoren ausgestattet wird, die für die Konstruktion graphenbasierter Bedeutungsrepräsentationen entworfen wurden.

Das in dieser Arbeit beschriebene System folgt einer Pipeline-Architektur und umfasst Lexikon-Induktion, Delexikalisierung, Filterung, Supertagging und Parsing. Dadurch, dass eine vereinfachte Grammatik verwendet wird, kann der Parser mit einem komprimierten, menschenlesbaren Lexikon arbeiten und linguistisch nachvollziehbare Ableitungen der Bedeutungsrepräsentationen produzieren. Dabei wird ein Smatch-Score erreicht, der mit anderen auf Combinatory Categorial Grammar aufbauenden Parsern vergleichbar ist.

Appendix C.

Publications Related to this Thesis

This appendix contains an overview of the peer-reviewed publications that have resulted from the work described in this thesis. In some cases, the publications listed below describe preliminary results that are not reflected in the present text, but which have been important in shaping it.

An overview of each publication's content is provided below. In cases where publications have more than one author, I also state the extent of each author's contribution.

- **Sebastian Beschke. Evaluating Supervised Semantic Parsing Methods on Application-Independent Data.** Pristine Perspectives on Logic, Language, and Computation. ESSLLI 2012 and ESSLLI 2013 Student Sessions. Selected Papers vol. 8607 19–25 (2014).
  I propose a research methodology for expanding Semantic Parsing to broad-coverage data sets. At the time of writing, most published results in Semantic Parsing had been on narrow, application-focused data sets and were achieving increasingly strong results. To expand the scope of semantic parsing to more general and larger data sets, I suggest that computational issues need to be addressed and that the question of the ideal properties of a meaning representation language is still open. A preliminary experiment described in the paper showed that a semantic parsing algorithm with strong performance on application-specific meaning representation performed poorly when presented with more general and extensive meaning representations.

- **Sebastian Beschke, Yang Liu, and Wolfgang Menzel. Large-scale CCG Induction from the Groningen Meaning Bank.** Proceedings of the ACL 2014 Workshop on Semantic Parsing 12–16 (2014).
  We describe a predecessor of the recursive splitting algorithm for the induction of semantic CCG lexica. The algorithm breaks apart meaning representations from the Groning Meaning Bank corpus, given in Discourse Representation

Theory, using λ-calculus. We describe heuristics needed to make the algorithm computationally feasible, and report statistics on the extracted lexical items. This paper has been written by Sebastian Beschke. The underlying implementation and experiments are also due to Sebastian Beschke. Yang Liu and Wolfgang Menzel provided advice and comments both ahead of and during the work on the paper.

- **Sebastian Beschke and Wolfgang Menzel. Graph Algebraic Combinatory Categorial Grammar.** Proceedings of the Seventh Joint Conference on Lexical and Computational Semantics 54–64 (Association for Computational Linguistics, 2018).
  We introduce a predecessor of Graph-Algebraic Combinatory Categorial Grammar to induce CCG lexica for Abstract Meaning Representation parsing. The paper describes a syntax-driven splitting algorithm equipped with alignment constraints, and reports statistics for lexicon induction on the AMR 1.0 corpus. It also includes an analysis of mismatches between CCG and AMR.
  This paper and the underlying implementation and experiments have been created by Sebastian Beschke. Wolfgang Menzel provided advice throughout the work on this topic, as well as comments on early versions of the paper.

- **Sebastian Beschke. Exploring Graph-Algebraic CCG Combinators for Syntactic-Semantic AMR Parsing.** Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2019) 112–121.
  I describe a semantic parsing system based on Graph-Algebraic Combinatory Categorial Grammar, including Expectation Maximisation filtering, a supertagger, and an oracle parser. The system strongly resembles the semantic parser presented in this dissertation; however, the evaluations conducted in this thesis use different GA-CCG rule sets and a different training algorithm.

Appendix D.

Software Created for this Thesis

Throughout this thesis, we report experimental results and outputs of our GA-CCG semantic parsing pipeline. As a key outcome of this research project, the software components underlying our results have been published under free software licenses. In this appendix, we briefly describe the software packages that have been created in the context of this thesis:

- *gramr* is a software package implementing the lexicon induction, filtering, training, and parsing stages of the GA-CCG parsing pipeline.
- *s2tagger* is a standalone tool implementing supertagger training and prediction.
- *gramr-thesis-experiments* is a repository to support reproduction of the reported results, containing all necessary scripts and documentation.

D.1. gramr

Most of the functionality described in this thesis is implemented in the package *gramr*, short for Grammar-Based Graph Meaning Representation Parser. The software is implemented in the Scala programming language[1] and available from `https://gitlab.com/nats/gramr`[2].

D.1.1. Package Contents

The *gramr* repository contains two sub-projects, `base` and `experiments`. The latter implements an experimental user interface that uses dedicated script files to

[1] See `https://www.scala-lang.org/` (retrieved 19 July 2021).

[2] The description of gramr and the experimental results in this thesis are based on Version 306a of gramr (`https://gitlab.com/nats/gramr/-/tree/v306a/`).

Concept / Algorithm	**Package or Module**
Semantic Operators (Section 4.1)	`gramr.graph.edit.sgraph`
GA-CCG Rules (Section 4.2)	`gramr.construction.sgraph.SGraphRule`[3]
Syntax-Driven Splitting Algorithm (Algorithm 5.1)	`gramr.split.RecursiveSplitting`, `gramr.split.PatternBasedSplitGenerator`
Constrained Graph Splitting Algorithm (Algorithm 5.2)	`gramr.split.PatternBasedSplitGenerator`, `gramr.split.labelling`
Delexicalisation (Section 6.1)	`gramr.graph.lexemes.delexicalizeLexicon`
Lexeme Patterns (Section 6.1.1)	`gramr.graph.lexemes.WildcardFiller`
Calculation of Template and Lexeme Counts (Algorithm 6.1)	`gramr.graph.split.em.Iteration`
EM Algorithm for GA-CCG Filtering (Algorithm 6.2)	`gramr.graph.split.em.Estimator`
Beam-Search CKY Algorithm for GA-CCG Parsing (Algorithm 7.1)	`gramr.parse.cky.CkyParser`
Algorithm for Greedy Coreference Resolution (Algorithm 7.5)	`gramr.parse.postprocess.CorefResolver`
Training Loop (Algorithm 7.6)	`gramr.learn.AdadeltaLearner`
Cost-Sensitive Perceptron (Algorithm 7.7)	`gramr.learn.CostSensitiveUpdateStrategy`
Element-Wise Precision (Definition 7.2)	`gramr.parse.ActionPredicates.\ GraphElementsCorrect.precision`
Alignment Coverage (Definition 7.3)	`gramr.parse.ActionPredicates.\ AlignmentsFulfilled.accuracy`
Feature Extraction (Section 8.1.6)	`gramr.learn.features`

Table D.1.: A list of the key algorithms of this thesis, and the Scala modules which implement them.

Reference	Source File
BiLSTM tagger (Section 3.5.1)	`tagger.py`
Masked loss function (Equation 6.2)	`masked_loss.py`
Jackknifing algorithm (Section 6.3.5)	`__main__.py`

Table D.2.: An overview of the algorithms and concepts implemented in s2tagger, along with the corresponding source files.

configure the parser components. The former contains implementations of the algorithms and concepts discussed in this thesis, along with more administrative tools to handle serialisation and deserialisation of AMR corpora, parser models, and the like. Table D.1 shows where implementations of the core algorithms can be found in the *gramr* package.

D.2. s2tagger

The package *s2tagger* (short for *simple supertagger*) implements the supertagging functionality described in Section 6.3. It is implemented in Python[4] using the Keras interface to the Tensorflow 2 deep learning library[5]. s2tagger is available from `https://gitlab.com/nats/s2tagger/`[6].

Table D.2 gives an overview of the functionality implemented in s2tagger that is relevant to the contents of this thesis, and points to the files where the corresponding implementations can be found.

D.3. gramr-thesis-experiments

The repository *gramr-thesis-experiments* contains all scripts and instructions needed to reproduce the results reported in this thesis. It is published at `https://gitlab.com/nats/gramr-thesis-experiments`.

The repository itself contains detailed documentation on running the experiments. This section, gives a brief overview of the process, and documents how the scripts in the repository correspond to the reported experiments.

[4] See `https://www.python.org/` (retrieved 23 July 2021).

[5] See `https://www.tensorflow.org/` (retrieved 23 July 2021).

[6] The experiments in this thesis are based on s2tagger Version 1.2.2 available from `https://gitlab.com/nats/s2tagger/-/tree/v1.2.2`.

D.3.1. Setup and Running Experiments

Running the experiments requires the following steps:

1. All software dependencies must be installed.
2. An enriched AMR corpus must be created by running the *preprocessing pipeline*.
3. Individual experiments can now be run using the corresponding shell scripts.
4. Experimental results can be compiled using the analysis notebooks.

Dependencies

The following software packages are required: `git`, `python3.8`, `sbt`, `docker`, `docker-compose`. Exactly how these are installed varies depending on the operating system and is thus not described in more detail.

The AMR 1.0 corpus[7] and the CCGBank 1.1 corpus[8] must be obtained and extracted into the `data` subdirectory.

Finally, the software dependencies for running the experiments themselves can be installed using the provided scripts.

```
$ ./install.sh
```

Preprocessing

Due to the complex requirements of the various tools, the preprocessing step is implemented using Docker. The following three steps build the required Docker images and run all preprocessing steps:

```
$ ./build.sh
$ ./preprocess.sh
$ ./preprocess-ccgbank.sh
```

[7] Available at `https://catalog.ldc.upenn.edu/LDC2014T12` (retrieved 29 July 2021).
[8] Available at `https://catalog.ldc.upenn.edu/LDC2005T13` (retrieved 29 July 2021).

Running Experiments

Scripts for individual experiments are located in the `experiments` directory. Each subdirectory contains a separate `README.md` file describing the experiment and the steps required to run it.

Most of the experiments contain either a `run.sh` file which runs the experiment, or a series of numbered scripts. In the latter case, the scripts should be run in order. In general, the corresponding `README.md` files should be consulted to understand the requirements of each experiment.

As an example, the full parsing pipeline using the `no-modify` rule set can be run as follows:

```
$ cd experiments/parsing/full-no-modify
$ ./01-induce.sh
$ ./02-extract-test-sentences.sh
$ ./03-tag.20.sh
$ ./04-train.sh
$ ./val.sh
$ ./test.sh
```

Induction and training scripts are set up to be used on machines with at least 64 gigabytes of RAM. On smaller machines, the setting `-Xmx48g` (or similar) may be adapted to the available size[9].

Compiling Results

Scripts for analysing the output of experiments are collected in the `experiments/analysis` directory, mainly in the form of Jupyter notebooks[10]. In addition, some experiment directories also contain scripts for extracting results. As an example, the experiment in `experiments/parsing/full-no-modify` contains a script `eval.sh` which computes Smatch scores on the validation and test sets. Meanwhile, the notebook `experiments/analysis/parsing.ipynb` contains graphs and tables extracted from the outputs of the various parsing experiments.

Experiment	Script Directory
Large Scale Lexicon Induction (Section 5.3)	
Alignment Strategies (Section 5.3.4)	`induce-vary-alignments`
Grammars (Section 5.3.5)	`induce-vary-grammars`
n-Best Parsing (Section 5.3.6)	`induce-vary-ccg-derivation-count`
Coreference Nodes (Section 5.3.7)	`induce-vary-max-corefs`
Grammar Coverage (Section 5.4)	`grammar-coverage`
Delexicalisation (Section 6.1.2)	`parsing/01-induce.sh`
EM Filtering (Section 6.2.3)	`parsing/01-induce.sh`
Supertagging (Section 6.3.6)	
Embeddings	`tagging/vary-embeddings`
Features	`tagging/vary-features`
LSTM Architecture	`tagging/vary-lstm-layout`
Parser Tuning (Section 8.1)	
Baseline Settings (Section 8.1.3)	`parsing/baseline`
Bootstrapping (Section 8.1.4)	`parsing/bootstrap`
Beam Size (Section 8.1.5)	`parsing/beam`
Features (Section 8.1.6)	`parsing/features`
Rule Sets (Section 8.1.7)	`parsing/grammar`
Amount of Training Data (Section 8.1.8)	`parsing/training-data`
Final System Evaluation (Section 8.2)	`parsing/full-no-modify`

Table D.3.: An overview of the experiments reported in this thesis and the scripts in the *gramr-thesis-experiments* repository.

D.3.2. Included Experiments

All experimental results reported in this thesis have been obtained using scripts contained in the *gramr-thesis-experiments* repository. Table D.3 shows the correspondence between each experiment and its script files.

[9] See `https://docs.oracle.com/en/java/javase/16/docs/specs/man/java.html#extra-options-for-java` (retrieved 29 July 2021).
[10] See `https://jupyter.org/` (retrieved 29 July 2021).

List of Figures

List of Tables

List of Algorithms

Bibliography

Artzi, Yoav, Kenton Lee and Luke S Zettlemoyer (2015). 'Broad-coverage CCG Semantic Parsing with AMR'. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, pp. 1699–1710. URL: http://aclweb.org/anthology/D15-1198.

Baker, James K. (1979). 'Trainable grammars for speech recognition'. In: *Speech communication papers presented at the 97th meeting of the Acoustical Society of America*. Cambridge, MA: MIT, pp. 547–550.

Ballesteros, Miguel and Yaser Al-Onaizan (2017). 'AMR Parsing using Stack-LSTMs'. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, pp. 1269–1275. URL: https://www.aclweb.org/anthology/D17-1130.

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer and Nathan Schneider (2013). 'Abstract Meaning Representation for Sembanking'. In: *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*. Sofia, Bulgaria: Association for Computational Linguistics, pp. 178–186. URL: http://www.aclweb.org/anthology/W13-2322.

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer and Nathan Schneider (2019). *Abstract Meaning Representation (AMR) 1.2.6 Specification*. en. URL: https://github.com/amrisi/amr-guidelines (visited on 28/10/2020).

Bangalore, Srinivas and Aravind K. Joshi (1999). 'Supertagging: An Approach to Almost Parsing'. In: *Computational Linguistics* 25.2, pp. 237–265. URL: https://aclanthology.org/J99-2004.

Basile, Valerio, Johan Bos, Kilian Evang and Noortje Venhuizen (2012). 'Developing a large semantically annotated corpus'. In: *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC-2012)*. Ed. by Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Jan Odijk and Stelios Piperidis. Istanbul, Turkey: European Language Resources Association (ELRA), pp. 3196–3200. ISBN: 978-2-9517408-7-7.

URL: http://www.lrec-conf.org/proceedings/lrec2012/pdf/534_Paper.pdf.

Berant, Jonathan and Percy Liang (2015). 'Imitation Learning of Agenda-based Semantic Parsers'. In: *Transactions of the Association for Computational Linguistics* 3, pp. 545–558. URL: https://tacl2013.cs.columbia.edu/ojs/index.php/tacl/article/view/646.

Blodgett, Austin and Nathan Schneider (2019). 'An Improved Approach for Semantic Graph Composition with CCG'. In: *Proceedings of the 13th International Conference on Computational Semantics - Long Papers*. Gothenburg, Sweden: Association for Computational Linguistics, pp. 55–70. URL: https://www.aclweb.org/anthology/W19-0405.

Bos, Johan (2008). 'Wide-Coverage Semantic Analysis with Boxer'. In: *Semantics in Text Processing. STEP 2008 Conference Proceedings*. College Publications, pp. 277–286. URL: https://www.aclweb.org/anthology/W08-2222.

Bos, Johan (2016). 'Squib: Expressive Power of Abstract Meaning Representations'. In: *Computational Linguistics* 42.3. Place: Cambridge, MA Publisher: MIT Press, pp. 527–535. DOI: 10.1162/COLI_a_00257. URL: https://aclanthology.org/J16-3006.

Brown, Peter F., Stephen A. Della Pietra, Vincent J. Della Pietra and Robert L. Mercer (1993). 'The Mathematics of Statistical Machine Translation: Parameter Estimation'. In: *Computational Linguistics* 19.2. Place: Cambridge, MA Publisher: MIT Press, pp. 263–311. URL: https://aclanthology.org/J93-2003.

Cai, Shu and Kevin Knight (2013). 'Smatch: an Evaluation Metric for Semantic Feature Structures'. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics* (*Volume 2: Short Papers*). Sofia, Bulgaria: Association for Computational Linguistics, pp. 748–752. URL: http://www.aclweb.org/anthology/P13-2131.

Clark, Stephen and James R. Curran (2004). 'The importance of supertagging for wide-coverage CCG parsing'. en. In: *Proceedings of the 20th international conference on Computational Linguistics - COLING '04*. Geneva, Switzerland: Association for Computational Linguistics, 282–es. DOI: 10.3115/1220355.1220396. URL: http://portal.acm.org/citation.cfm?doid=1220355.1220396 (visited on 22/02/2021).

Cocke, John and Jacob T. Schwartz (1970). *Programming languages and their compilers: Preliminary notes*. Technical report. New York, USA: New York University.

Collins, Michael (2002). 'Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms'. In: *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*. Association

for Computational Linguistics, pp. 1–8. DOI: 10.3115/1118693.1118694. URL: http://www.aclweb.org/anthology/W02-1001.

Copestake, Ann and Dan Flickinger (2000). 'An Open Source Grammar Development Environment and Broad-coverage English Grammar Using HPSG'. In: *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC'00)*. Athens, Greece: European Language Resources Association (ELRA). URL: http://www.lrec-conf.org/proceedings/lrec2000/pdf/371.pdf.

Courcelle, Bruno and Joost Engelfriet (2012). *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. 1st Edition. New York, NY, USA: Cambridge University Press.

Damonte, Marco and Shay B. Cohen (2018). 'Cross-Lingual Abstract Meaning Representation Parsing'. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, pp. 1146–1155. DOI: 10.18653/v1/N18-1104. URL: https://www.aclweb.org/anthology/N18-1104.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee and Kristina Toutanova (June 2019). 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

Evang, Kilian (2016). 'Cross-lingual Semantic Parsing with Categorial Grammars'. ISBN: 9789036794749 OCLC: 968211067. PhD thesis. University of Groningen. 264 pp. URL: https://kilian.evang.name/phdthesis.pdf (visited on 15/12/2021).

Flanigan, Jeffrey, Sam Thomson, Jaime Carbonell, Chris Dyer and Noah A. Smith (2014). 'A Discriminative Graph-Based Parser for the Abstract Meaning Representation'. In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Baltimore, Maryland: Association for Computational Linguistics, pp. 1426–1436. URL: http://www.aclweb.org/anthology/P14-1134.

Glorot, Xavier and Yoshua Bengio (2010). 'Understanding the difficulty of training deep feedforward neural networks'. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Sardinia, Italy: PMLR, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

Goldberg, Yoav (2017). *Neural Network Methods in Natural Language Processing*. Ed. by Graeme Hirst. Morgan & Claypool Publishers. ISBN: 1-62705-298-4.

Goodman, James, Andreas Vlachos and Jason Naradowsky (2016). 'Noise reduction and targeted exploration in imitation learning for Abstract Meaning Representation parsing'. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, pp. 1–11. DOI: 10.18653/v1/P16-1001. URL: https://aclanthology.org/P16-1001.

Groschwitz, Jonas, Meaghan Fowlie, Mark Johnson and Alexander Koller (2017). 'A constrained graph algebra for semantic parsing with AMRs'. In: *Proceedings of the 12th International Conference on Computational Semantics*. Montpellier, France: Association for Computational Linguistics. URL: http://www.aclweb.org/anthology/W17-6810 (visited on 26/02/2018).

Groschwitz, Jonas, Matthias Lindemann, Meaghan Fowlie, Mark Johnson and Alexander Koller (2018). 'AMR dependency parsing with a typed semantic algebra'. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, pp. 1831–1841. DOI: 10.18653/v1/P18-1170. URL: https://www.aclweb.org/anthology/P18-1170.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). 'Long Short-Term Memory'. In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

Hockenmaier, Julia and Mark Steedman (2007). 'CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank'. In: *Computational Linguistics* 33.3, pp. 355–396. DOI: 10.1162/coli.2007.33.3.355. URL: https://aclanthology.org/J07-3004.

Honnibal, Matthew, James R. Curran and Johan Bos (2010). 'Rebanking CCGbank for Improved NP Interpretation'. In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Uppsala, Sweden: Association for Computational Linguistics, pp. 207–215. URL: http://www.aclweb.org/anthology/P10-1022.

Hovy, Eduard, Mitchell Marcus, Martha Palmer, Lance Ramshaw and Ralph Weischedel (2006). 'OntoNotes: The 90% Solution'. In: *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*. New York City, USA: Association for Computational Linguistics, pp. 57–60. URL: https://www.aclweb.org/anthology/N06-2015.

Howard, Jeremy and Sebastian Ruder (2018). 'Universal Language Model Fine-tuning for Text Classification'. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia:

Association for Computational Linguistics, pp. 328–339. DOI: 10.18653/v1/P18-1031. URL: https://www.aclweb.org/anthology/P18-1031.

Kamp, Hans and Uwe Reyle (1993). *From Discourse to Logic: Introduction to Model-theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Dordrecht: Kluwer. ISBN: 978-0-7923-2403-4.

Kasami, Tadao (1965). *An efficient recognition and syntax-analysis algorithm for context-free languages*. Technical Report. Air Force Research Laboratory.

Kingma, Diederik P. and Jimmy Ba (2017). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].

Knight, Kevin, Bianca Badarau, Laura Baranescu, Claire Bonial, Madalina Bardocz, Kira Griffitt, Ulf Hermjakob, Daniel Marcu, Martha Palmer, Tim O'Gorman and Nathan Schneider (2017). *Abstract Meaning Representation (AMR) Annotation Release 2.0*. Philadelphia: Linguistic Data Consortium. URL: https://catalog.ldc.upenn.edu/LDC2017T10.

Knight, Kevin, Bianca Badarau, Laura Baranescu, Claire Bonial, Madalina Bardocz, Kira Griffitt, Ulf Hermjakob, Daniel Marcu, Martha Palmer, Tim O'Gorman and Nathan Schneider (2020). *Abstract Meaning Representation (AMR) Annotation Release 3.0*. Philadelphia: Linguistic Data Consortium. URL: https://catalog.ldc.upenn.edu/LDC2020T02.

Knight, Kevin, Laura Baranescu, Claire Bonial, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Daniel Marcu, Martha Palmer and Nathan Schneider (2014). *Abstract Meaning Representation (AMR) Annotation Release 1.0 LDC2014T12*. Philadelphia: Linguistic Data Consortium. URL: https://catalog.ldc.upenn.edu/LDC2014T12.

Koller, Alexander (2015). 'Semantic construction with graph grammars'. In: *Proceedings of the 11th International Conference on Computational Semantics*. London, UK: Association for Computational Linguistics, pp. 228–238. URL: http://www.aclweb.org/anthology/W15-0127.

Koller, Alexander, Stephan Oepen and Weiwei Sun (2019). 'Graph-Based Meaning Representations: Design and Processing'. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*. Florence, Italy: Association for Computational Linguistics, pp. 6–11. DOI: 10.18653/v1/P19-4002. URL: https://www.aclweb.org/anthology/P19-4002.

Kuhlmann, Marco and Giorgio Satta (2014). 'A New Parsing Algorithm for Combinatory Categorial Grammar'. In: *Transactions of the Association for Computational Linguistics* 2, pp. 405–418. DOI: 10.1162/tacl_a_00192. URL: https://aclanthology.org/Q14-1032.

Kwiatkowski, Tom, Luke Zettlemoyer, Sharon Goldwater and Mark Steedman (2010). 'Inducing Probabilistic CCG Grammars from Logical Form with Higher-Order Unification'. In: *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*. Cambridge, MA: Association for Computational Linguistics, pp. 1223–1233. URL: http://www.aclweb.org/anthology/D10-1119.

Kwiatkowski, Tom, Luke Zettlemoyer, Sharon Goldwater and Mark Steedman (2011). 'Lexical Generalization in CCG Grammar Induction for Semantic Parsing'. In: *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*. Edinburgh, Scotland, UK.: Association for Computational Linguistics, pp. 1512–1523. URL: https://aclanthology.org/D11-1140.

Lari, K. and S. J. Young (1990). 'The estimation of stochastic context-free grammars using the Inside-Outside algorithm'. In: *Computer Speech & Language* 4.1, pp. 35–56. ISSN: 0885-2308. DOI: https://doi.org/10.1016/0885-2308(90)90022-X. URL: https://www.sciencedirect.com/science/article/pii/088523089090022X.

Le, Phong and Willem Zuidema (2012). 'Learning Compositional Semantics for Open Domain Semantic Parsing'. In: *Proceedings of COLING 2012*. Mumbai, India: The COLING 2012 Organizing Committee, pp. 1535–1552. URL: http://www.aclweb.org/anthology/C12-1094.

Lee, Young-Suk, Ramón Fernandez Astudillo, Tahira Naseem, Revanth Gangi Reddy, Radu Florian and Salim Roukos (2020). 'Pushing the Limits of AMR Parsing with Self-Learning'. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, pp. 3208–3214. DOI: 10.18653/v1/2020.findings-emnlp.288. URL: https://www.aclweb.org/anthology/2020.findings-emnlp.288.

Lewis, Mike, Kenton Lee and Luke Zettlemoyer (2016). 'LSTM CCG Parsing'. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, pp. 221–231. DOI: 10.18653/v1/N16-1026. URL: https://aclanthology.org/N16-1026.

Lewis, Mike and Mark Steedman (2014). 'A* CCG Parsing with a Supertag-factored Model'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, pp. 990–1000. URL: http://www.aclweb.org/anthology/D14-1107.

Li, Bin, Yuan Wen, Weiguang Qu, Lijun Bu and Nianwen Xue (2016). 'Annotating the Little Prince with Chinese AMRs'. In: *Proceedings of the 10th Linguistic Annotation Workshop held in conjunction with ACL 2016 (LAW-X 2016)*. Berlin, Germany: Association for Computational Linguistics, pp. 7–15. DOI: 10.18653/v1/W16-1702. URL: https://www.aclweb.org/anthology/W16-1702.

Lindemann, Matthias, Jonas Groschwitz and Alexander Koller (2019). 'Compositional Semantic Parsing across Graphbanks'. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, pp. 4576–4585. DOI: 10.18653/v1/P19-1450. URL: https://www.aclweb.org/anthology/P19-1450.

Liu, Yijia, Wanxiang Che, Bo Zheng, Bing Qin and Ting Liu (2018). 'An AMR Aligner Tuned by Transition-based Parser'. In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, pp. 2422–2430. URL: https://www.aclweb.org/anthology/D18-1264.

Lowerre, Bruce T. (1976). 'The HARPY Speech Recognition System'. PhD thesis. Pittsburgh, Pennsylvania, USA: Carnegie Mellon University.

Marcus, Mitchell P., Beatrice Santorini and Mary Ann Marcinkiewicz (1993). 'Building a Large Annotated Corpus of English: The Penn Treebank'. In: *Computational Linguistics* 19.2. Place: Cambridge, MA Publisher: MIT Press, pp. 313–330. URL: https://aclanthology.org/J93-2004.

Mikolov, Tomás, Kai Chen, Greg Corrado and Jeffrey Dean (2013). 'Efficient Estimation of Word Representations in Vector Space'. In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1301.3781.

Misra, Kumar Dipendra and Yoav Artzi (2016). 'Neural Shift-Reduce CCG Semantic Parsing'. In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, pp. 1775–1786. URL: http://aclweb.org/anthology/D16-1183.

Montague, Richard (1973). 'The Proper Treatment of Quantification in Ordinary English'. In: *Philosophy, Language, and Artificial Intelligence*. Ed. by James H. Fetzer, Jack Kulas, James H. Fetzer and Terry L. Rankin. Vol. 2. Dordrecht: Springer Netherlands, pp. 141–162. DOI: 10.1007/978-94-009-2727-8_7. URL: http://www.springerlink.com/index/10.1007/978-94-009-2727-8_7.

Noord, Rik van and Johan Bos (2017). 'Neural Semantic Parsing by Character-based Translation: Experiments with Abstract Meaning Representations'. In: *Computational Linguistics in the Netherlands Journal* 7, pp. 93–108. ISSN: 2211-4009. URL: http://www.clinjournal.org/sites/clinjournal.org/files/07.neural-semantic-parsing.pdf.

Pennington, Jeffrey, Richard Socher and Christopher Manning (2014). 'GloVe: Global Vectors for Word Representation'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for

Computational Linguistics, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162.

Peters, Matthew, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee and Luke Zettlemoyer (2018). 'Deep Contextualized Word Representations'. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, pp. 2227–2237. DOI: 10.18653/v1/N18-1202. URL: https://www.aclweb.org/anthology/N18-1202.

Pourdamghani, Nima, Yang Gao, Ulf Hermjakob and Kevin Knight (2014). 'Aligning English Strings with Abstract Meaning Representation Graphs'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, pp. 425–429. URL: http://www.aclweb.org/anthology/D14-1048.

Price, P. J. (1990). 'Evaluation of Spoken Language Systems: the ATIS Domain'. In: *Speech and Natural Language: Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27,1990*. URL: https://www.aclweb.org/anthology/H90-1020.

Rosenblatt, Frank (1958). 'The perceptron: a probabilistic model for information storage and organization in the brain.' In: *Psychological review* 65.6. Publisher: American Psychological Association, p. 386.

Singh-Miller, Natasha and Michael Collins (2007). 'Trigger-Based Language Modeling using a Loss-Sensitive Perceptron Algorithm'. In: *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*. Vol. 4, pp. IV–25–IV–28. DOI: 10.1109/ICASSP.2007.367154.

Sobrevilla Cabezudo, Marco Antonio and Thiago Pardo (2019). 'Towards a General Abstract Meaning Representation Corpus for Brazilian Portuguese'. In: *Proceedings of the 13th Linguistic Annotation Workshop*. Florence, Italy: Association for Computational Linguistics, pp. 236–244. DOI: 10.18653/v1/W19-4028. URL: https://www.aclweb.org/anthology/W19-4028.

Steedman, Mark (2000). *The Syntactic Process*. Cambridge, MA: MIT Press.

Szubert, Ida, Adam Lopez and Nathan Schneider (2018). 'A Structured Syntax-Semantics Interface for English-AMR Alignment'. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, pp. 1169–1180. DOI: 10.18653/v1/N18-1106. URL: http://aclweb.org/anthology/N18-1106.

Vaswani, Ashish, Yonatan Bisk, Kenji Sagae and Ryan Musa (2016). 'Supertagging With LSTMs'. en. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, pp. 232–237. DOI: 10.18653/v1/N16-1027. URL: http://aclweb.org/anthology/N16-1027.

Vijay-Shanker, Krishnamurti and David J. Weir (1994). 'The equivalence of four extensions of context-free grammars'. In: *Mathematical systems theory* 27.6, pp. 511–546. URL: http://link.springer.com/article/10.1007/BF01191624.

Wang, Chuan, Nianwen Xue and Sameer Pradhan (2015a). 'A Transition-based Algorithm for AMR Parsing'. In: *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Denver, Colorado: Association for Computational Linguistics, pp. 366–375. URL: http://www.aclweb.org/anthology/N15-1040.

Wang, Chuan, Nianwen Xue and Sameer Pradhan (2015b). 'Boosting Transition-based AMR Parsing with Refined Actions and Auxiliary Analyzers'. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. Beijing, China: Association for Computational Linguistics, pp. 857–862. DOI: 10.3115/v1/P15-2141. URL: https://aclanthology.org/P15-2141.

Winograd, Terry (1971). 'Procedures as a representation for data in a computer program for understanding natural language'. PhD thesis. Cambridge, Massachusetts, USA: Massachusetts Institute of Technology. URL: https://dspace.mit.edu/handle/1721.1/7095.

Wong, Yuk Wah and Raymond J. Mooney (2007). 'Learning Synchronous Grammars for Semantic Parsing with Lambda Calculus'. In: *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. Prague, Czech Republic: Association for Computational Linguistics, pp. 960–967. URL: http://www.aclweb.org/anthology/P07-1121.

Xu, Dongqin, Junhui Li, Muhua Zhu, Min Zhang and Guodong Zhou (2020). 'Improving AMR Parsing with Sequence-to-Sequence Pre-training'. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, pp. 2501–2511. DOI: 10.18653/v1/2020.emnlp-main.196. URL: https://www.aclweb.org/anthology/2020.emnlp-main.196.

Younger, Daniel H. (1967). 'Recognition and parsing of context-free languages in time n3'. In: *Information and Control* 10.2, pp. 189–208. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(67)80007-X. URL: https://www.sciencedirect.com/science/article/pii/S001999586780007X.

Zeiler, Matthew D. (2012). 'ADADELTA: An Adaptive Learning Rate Method'. In: *CoRR* abs/1212.5701. arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701.

Zelle, John M. and Raymond J. Mooney (1996). 'Learning to parse database queries using inductive logic programming'. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. Portland, Oregon, USA, pp. 1050–1055. URL: http://www.aaai.org/Papers/AAAI/1996/AAAI96-156.pdf (visited on 25/02/2014).

Zettlemoyer, Luke S and Michael Collins (2005). 'Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars'. en. In: *Proceedings of the Twenty-First Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-05)*. AUAI Press, pp. 658–666.

Zhang, Sheng, Xutai Ma, Kevin Duh and Benjamin Van Durme (2019). 'Broad-Coverage Semantic Parsing as Transduction'. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, pp. 3786–3798. DOI: 10.18653/v1/D19-1392. URL: https://www.aclweb.org/anthology/D19-1392.

Zhao, Kai and Liang Huang (2013). 'Minibatch and Parallelization for Online Large Margin Structured Learning'. In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Atlanta, Georgia: Association for Computational Linguistics, pp. 370–379. URL: https://aclanthology.org/N13-1038.

Zhou, Junsheng, Feiyu Xu, Hans Uszkoreit, Weiguang Qu, Ran Li and Yanhui Gu (2016). 'AMR Parsing with an Incremental Joint Model'. In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, pp. 680–689. DOI: 10.18653/v1/D16-1065. URL: https://aclanthology.org/D16-1065.