Imperial College London
Department of Computing

MEng Individual Project

Logic-based Approach to Machine Comprehension of Text

Author:
Piotr Chabierski

Project Supervisors:
Prof. Alessandra Russo
Mark Law

June 19, 2017

Abstract

Machine comprehension of text is a long-term open problem in Artificial Intelligence and can be assessed by machine's ability to answer questions about passages of text. In this project we combine ideas from the fields of logic-based learning, knowledge representation and computational linguistics to develop a domain-independent system capable of both answering questions about text based on Answer Set Program representation and of acquiring background knowledge automatically using Inductive Logic Programming.

The approach uses the Combinatory Categorial Grammar and Montague-style semantics, expressed with λ-ASP calculus, to perform domain-independent semantic analysis of text and derive Answer Set Program representations. The system can use the semantic representation to automatically derive bounds on the hypothesis space in the form of mode declarations and generate an Inductive Logic Programming task suitable as an input to ILASP [31] algorithm.

The system was evaluated against MemN2N [61] and EntNet [22] neural network-based models on a subset of *The (20) QA bAbi Tasks Dataset* [66] and achieved comparable accuracy using a significantly smaller number of training examples.

Acknowledgements

I would like to thank my supervisors, Prof. Alessandra Russo and Mark Law, for supporting me with their expertise in the field of logic-based learning and for their guidance and enthusiasm throughout the project. I want to express my gratitude for the time that they dedicated to our weekly meetings which gave rise to many interesting project discussions and ideas. I would also like to thank Prof. Marek Sergot, my Personal Tutor, for providing me with advice and assistance during the last four years. Finally, I would like to thank my parents.

Contents

Chapter 1

Introduction

Machine comprehension of text, also referred to as machine reading or natural language understanding, has been a central goal of Artificial Intelligence for over sixty years, its origin might be traced back to the Turing test. The task of Machine Comprehension of Text can be defined as follows:

"A machine comprehends a passage of text if, for any question regarding that text that can be answered correctly by a majority of native speakers, that machine can provide a string which those speakers would agree both answers that question, and does not contain information irrelevant to that question." [12]

In this project, approaches from the fields of computational linguistics, knowledge representation and logic-based learning are combined in order to build a fully-automated system capable of answering questions, as well as acquiring common sense knowledge from text.

1.1 Background

The first natural language understanding systems originated in the late 60s and early 70s with notable examples such as LUNAR and SHRDLU. LUNAR [68] was a *natural language interface* to a database. It translated natural language questions to queries to a database of information about lunar geology. SHRDLU [67] was able to sustain a conversation about the state of a *blocks world* as well as modify the environment based on a user's commands and a plan devised by the system. These two systems relied on logical forms derived from the textual input in order to respond to a user's queries. However, the process of deriving logical representations was largely rule based and as such could not be easily extended to other domains.

Achieving domain-independence was one of the motivations for approaches to *semantic parsing* developed from the 90s until now. Semantic parsing is a task of mapping natural language input to a structured representation suitable for manipulation by a machine. Over the last twenty years different approaches to that problem were proposed. Some of the early ones relied on Inductive Logic Programming [69] [63] and were followed by statistical machine learning methods [70] [51]. Most popular approaches to semantic parsing are

supervised and rely on annotated training examples in order to be trained for application to a particular domain.

Developments in linguistics and the theory of natural language grammar, in particular formalisation of Combinatory Categorial Grammar (CCG) [59], had a significant influence on the developments in the domain of semantic parsing. CCG is linguistically expressive and efficiently parsable grammar which offers transparent interface between syntax and semantics of natural language [60]. Many modern approaches to semantic parsing use CCG to represent syntactic structure of text [70] [29].

Creation of the C&C parser [16], which is a wide-coverage syntactic parser for CCG, motivated another approach to semantic parsing, implemented by the Boxer system [10]. In Boxer, representation is not learnt from data but is instead generated from the CCG parse tree by taking advantage of the clear interface between syntax and semantics offered by the CCG and by relying on the idea proposed by Richard Montague that formal semantics of English can be derived compositionally using lambda calculus [27]. Among others, Boxer was applied to the task of recognising textual entailment and to extracting RDF/OWL ontology from text [52].

However, in the recent years we could have observed a departure from the more traditional semantic parsing approaches in favour of models based on neural network architectures [23] [61]. Such approaches rely on large annotated datasets for training and are very effective at answering questions where the correct answer can be identified using lexical clues [13]. Some neural architectures, for example Memory Networks, were evaluated on datasets considered to require more sophisticated reasoning capabilities, such as *The (20) QA bAbi Tasks Dataset*, and the results were favourable [65]. Determining cognitive capabilities of different deep neural network-based approaches is still an open research problem [53].

1.2 Motivations

Logic-based approaches intuitively seem like a good fit for the problem of natural language understanding for two reasons. Firstly, symbolic representations used by logic-based methods are comprehensible by humans and allow specification of more abstract concepts such as temporal relations or negation. Secondly, background knowledge, which is essential for natural language understanding, can be easily added to the logic programs.

Recent advancements in the field of logic-based learning, namely invention of the ILASP algorithm [31], has opened new interesting possibilities regarding the type of common sense knowledge that can be learned from text. ILASP integrates brave and cautious induction within a unified learning framework and allows learning choice rules and constraints, which improves on the previous approaches to Inductive Logic Programming under the Answer Set Semantics in terms of the provided functionality.

Historically, logic-based approaches to natural language understanding have been associated with expert systems. However, advancements in the field of natural language processing, especially development of more accurate wide-coverage syntactic parsers [36], opened a possibility for developing more general approaches, as confirmed by the example of Boxer system, which is a wide-coverage semantic parser [10].

In the light of the recent developments in the fields of logic-based learning and natural language processing, building a general purpose logic-based system capable of both performing inference and learning would offer a new perspective on approaching the problem of machine comprehension of text and provide more insight into applicability of symbolic approaches in that problem domain.

1.3 Objectives

The main goal of the project is to investigate the applicability of logic-based learning approaches, in particular Inductive Logic Programming, to answering questions about natural language text written in English. The following high-level objectives were identified as necessary in order to achieve the main goal:

1. **Research of the current state-of-the-art in semantic parsing.** Inductive Logic Programming algorithms require their input to be specified using a formal representation, which in case of the algorithm that we are using - ILASP [31] are Answer Set Programs. Performing such a mapping exemplifies the more general problem of semantic parsing, which has been studied extensively. Therefore, a critical assessment of the applicability of different approaches to semantic parsing has to be undertaken in the context of our project.
2. **Translation of sentences to ASP.** Robust conversion of sentences written in English to ASP is a prerequisite both for answering questions using existing background knowledge and learning. In the interest of wider applicability of the system, the translation algorithm should be domain-independent.
3. **Automatic generation of learning tasks.** Inductive Logic Programming algorithms usually rely on the user to specify constraints on the hypothesis space and provide other hyper-parameters of the learning task, such as the maximum number of variables that each rule in the hypothesis space can take. In order to enhance our system's learning capabilities, automatic generation of learning tasks should be investigated.
4. **Evaluation of the approach on a popular dataset.** Due to exploratory nature of the project, evaluation of the developed approach on a widely used question answering dataset is crucial in order to gather insights into how it compares to more commonly used techniques, especially the ones relying on statistical machine learning models.

1.4 Project Outline

This section serves as a high-level overview of the tasks undertaken throughout the project. Where necessary, references are given to sections containing more details about specific parts of the developed system. It should be noted that all ideas outlined in this section were implemented as a part of the project.

The project could be conceptually divided into two parts: translation (Chapter 3) whose objective is to derive ASP representation for a given sentence based on the CCG grammar

and λ-ASP calculus, and learning (Chapter 4) in which the derived ASP representation is used to formulate an Inductive Logic Programming task that is run in order to derive additional background knowledge rules.

We begin by describing the ASP representation used in the project to represent text (Section 3.1). Then, λ-ASP calculus [5] which serves an an intermediate representation between English and ASP and allows for compositional derivation of semantic representation of phrases and sentences is introduced (Section 3.2.1). Intuitively, λ-ASP calculus follows the ASP syntax and extends it with abstraction and application, as known from λ-calculus.

Then, a systematic, linguistically-motivated approach to translation from English to ASP, which relies on CCG grammar and λ-ASP calculus, is presented (Section 3.2). To derive ASP representation of a sentence, every leaf node of the corresponding CCG parse tree is assigned a λ-ASP expression by a lexicon, which can be though of as a map between syntax and semantics of the given language (Section 3.2.2), and semantic representation of a complete sentence is built bottom-up by composing λ-ASP expressions corresponding to child nodes via function application (Section 3.2.3). The order of application is dictated by the structure of the parse tree. Then, the process of deriving the final ASP representation from the corresponding λ-ASP expression is explained (Section 3.4).

Example 1.1 (CCG parse tree and ASP representation of a sentence *Jack ate a sandwich*)**.**

Listing 1.1 CCG parse tree.

```
<T S[dcl]>
  <T NP>
    <L N NNP Jack>
  <T (S[dcl]\NP)>
    <L ((S[dcl]\NP)/NP) VBD ate>
    <T NP>
      <L (NP/N) DT a>
      <L N NN sandwich>
```

Listing 1.2 ASP representation.

```
binaryEvent(e0,eat,c1,n0).
unaryNominal(n0,sandwich).
unaryNominal(c1,jack).
metaData(0,e0)
```

We describe the algorithm for automatic generation of a learning task from examples specified as text-question-answer triples (Section 4.1), which relies on the English-to-ASP translation algorithm discussed earlier (Section 4.2). Then, the three-step process of deriving mode declarations, used to specify the hypothesis space, from task context and background knowledge is described in detail (Section 4.3).

First, we describe how inclusions and exclusions of each example are unified with the background knowledge and example context to obtain a list of potential ground hypothesis heads (Section 4.3.1). Secondly, the the algorithm assigning types to predicate arguments based on the context in which they occur is presented (Section 4.3.2). Finally, the algorithm which assigns types to predicates based their arguments is described (Section 4.3.3). Predicate and argument types provide information about number and parametrisation of mode declarations (Section 4.3.4).

Then, we describe a learning process in which multiple learning tasks are generated to explore different hypothesis spaces corresponding to the training examples, which stem from the differences in the parametrisation of the mode declarations. A set of heuristics was invented which order tasks based on certain criteria such as approximate size of the hypothesis space (Section 4.5). The ordered tasks are scheduled in parallel and the learning process is stopped when the first set of rules is learnt (Section 5.3.3).

Finally, we present the results of evaluation of the project on a subset of *The (20) QA bAbi Tasks Dataset* [66] and compare the results to the ones achieved by neural network-based approaches (Section 6.2.8). It can be observed that in its current form the system is less robust than the neural network-based approaches primarily due to errors in the syntactic parser. However, the system is able to achieve similar accuracy to the competing approaches using significantly smaller number of training examples.

1.5 Contributions

The main contribution of the project is the development of a *novel fully automated logic-based approach to natural language understanding* capable of performing inference and learning to answer questions about text. Specific contributions are as follows:

1. **ASP representation of natural language text** that can be used to derive semantics of phrases and sentences compositionally and is suitable for learning using Inductive Logic Programming (Section 3.1).
2. **Adaptation and extension of λ-ASP calculus** used as intermediate representation between English and ASP, to handle more complex linguistic constructions such as control, rising, relativisation and coordination (Section 3.3).
3. **Implementation of English-to-ASP translation algorithm** that is general as it can generate ASP representations for texts from different domains, and linguistically motivated as its design is rooted in the theory of formal grammar (Section 3.2).
4. **Automatic generation of bounds on the hypothesis space** in the form of mode declarations that are domain independent, rely on the context of the learning tasks as well as background knowledge, and offer considerable reduction in the size of the hypothesis space compared to the baseline approach (Section 4.3).
5. **Comparison of the approach to neural network-based models** on *The (20) QA bAbi Tasks Dataset*, which requires a significant reasoning capacity from the learner and is described as a "prerequisite for any system aiming at full text understanding and reasoning" [66].

Chapter 2

Background

2.1 Answer Set Programming

Answer set programming (ASP) is a form of declarative programming oriented towards difficult search problems [38]. ASP relies on the stable model (answer set) semantics of logic programs.

2.1.1 Programming Language

An answer set program is specified by a set of *rules*, which in turn comprise of a set of *literals*. A literal is either an atom p or its *default negation* not p, also referred to as *negation as failure*. In what follows, the subset of ASP language consisting of *normal rules*, *constraints* and *choice rules* is assumed. A *normal rule* r has the following form:

$$r: \quad \underbrace{h}_{head(r)} \leftarrow \underbrace{b_1, b_2, \dots b_m, \text{not } b_{m+1}, \dots \text{not } b_{m+k}}_{body(r)}$$

where $h, b_1, \dots, b_{m+k}$ are atoms. A rule whose body is empty ($m = k = 0$) is called a *fact*. A normal rule for which $k = 0$ is called a *positive rule* or a *definite clause*.

A *constraint* is a rule with an empty head [38]. The general form of a constraint is given by: $\leftarrow b_1, \dots b_m, \text{not } b_{m+1}, \dots \text{not } b_{m+k}$ where $b_1, \dots, b_{m+k}$ are atoms. Adding a constraint has the effect of eliminating all answer sets which both include $b_1, b_2, \dots, b_m$ and exclude $b_{m+1}, \dots b_{m+k}$ from the answer sets of a program.

A *choice rule* has the form: $l\{a_1, \dots, a_n\}u \leftarrow b_1, b_2, \dots b_m, \text{not } b_{m+1}, \dots \text{not } b_{m+k}$ where $a_1, \dots, a_n, b_1, \dots, b_{m+k}$ are atoms and bounds $l, u \in \mathbb{N}$ satisfy $0 \leq l \leq u$. The head of a choice rule is called an *aggregate*. Informally, given a ground choice rule with a head $l\{h_1, \dots h_n\}u$, if the body of the choice rule is satisfied, the choice rule generates all answer sets in which t, where $u \leq t \leq l$, ground atoms from the set $\{h_1, \dots, h_n\}$ are included [31].

A *variable* occurring in a rule r is said to be *safe* if it occurs in at least one positive literal in the body of r. *Grounding* is the process of replacing a program which contains variables with an equivalent program without variables [38].

2.1.2 Stable Model Semantics

In order to define a stable model of a normal logic program P, let us first introduce the definition of the minimal Herbrand model in the context of normal logic programs. The *Herbrand Base* $HB(P)$ is the set of all ground atoms made from the constants, predicate symbols and function symbols in P [30]. An *Herbrand interpretation* of P assigns a truth value to every atom in $HB(P)$. In order to specify an Herbrand interpretation for P, it is sufficient to say for each ground atom $a \in HB(P)$ if it is true or false [56].

Definition 2.1 (Herbrand Model)**.** An *Herbrand model* M of a normal logic program P is an Herbrand interpretation in which for every ground instance r of each rule in P such that $body(r)$ is satisfied by M, $head(r)$ is also satisfied by M.

Definition 2.2 (Minimal Herbrand Model)**.** An Herbrand model M of a normal logic program P is *minimal* if no proper subset of M is an Herbrand model of P [21].

Definite logic programs – programs that do not contain negation, have a unique minimal Herbrand model, also referred to as the *least Herbrand model*. Programs with negation can have multiple minimal Herbrand models [21]. Given the definition of the minimal Herbrand model, a stable model of a normal logic program P can be characterised as follows.

Definition 2.3 (Stable Model)**.** For any set M of ground atoms of a normal logic program P, let us define a *reduct* P^M as a logic program obtained from P by deleting

- each rule that has a negated literal *not* b in its body with $b \in M$
- all negated literals in the bodies of the remaining rules

M is a *stable model* of P if it coincides with the least Herbrand model of P^M [21].

The fact that normal logic programs (which include negation) can have multiple stable models leads to two different definitions of entailment [30].

Definition 2.4 (Cautious and Brave Entailment)**.** An atom a is *cautiously* entailed by a normal logic program P (written $P \models_c a$) if it is true in every stable model of P. An atom a is *bravely* entailed by a logic program P (written $P \models_b a$) if it is true in at least one stable model of P.

2.2 Inductive Logic Programming

Inductive Logic Programming is a research area which lies at the intersection of machine learning and computational logic [46]. Inductive Logic Programming can be considered as a search problem over a hypothesis space for a hypothesis which, together with background knowledge, entails a set of observations.

Definition 2.5 (Generalised inductive problem)**.** Three logic-based languages $\mathcal{L}_O$, $\mathcal{L}_B$, $\mathcal{L}_H$ are provided for expressing, respectively, observations, background knowledge and hypotheses. Given a set of examples $O \subseteq \mathcal{L}_O$ and consistent background knowledge $B \subseteq \mathcal{L}_B$ the

goal is to find hypothesis a $H \in \mathcal{L}_H$ such that

$$B \wedge H \vdash O$$

Usually, additional constraints are put on $\mathcal{L}_O$ and H. $\mathcal{L}_O$ is often constrained to ground literals or atoms and H is a single clause [46] (however, the latter does not apply to the ILASP algorithm). Specification of $\mathcal{L}_H$ is referred to as a *language bias* and constrains the space of possible hypotheses.

Observations are divided into *positive* and *negative* examples and the hypothesis H should *cover* all positive examples and none of the negative ones. The *covers* relation $c : H \times E$ maps from the set H of hypotheses, referred to also as *version space*, to the set E of examples. The precise definition of the *covers* relation is a part of specification of a particular inductive logic programming task and defines the learning setting [54].

The two most popular learning settings are learning from interpretations and learning from entailment. In the former, a model-theoretic approach is taken and examples are treated as complete interpretations and a hypothesis H is related to an example e by the *covers* relation if and only if e is a model of $B \cup H$ (where B is the background knowledge). In learning from entailment examples are ground facts of a theory and a hypothesis H is related to an example e by the *covers* relation if and only if e is entailed by $B \cup H$ [54].

2.2.1 Inductive Logic Programming under Answer Set Semantics

Inductive Logic Programming under answer set semantics can be formalised as follows.

Definition 2.6 (Inductive Logic Programming under Answer Set Semantics)**.** Given:

- background knowledge B specified as an answer set program
- positive examples E^+ and negative examples E^- specified as sets of literals

The task is to learn a hypothesis H such that:

- $B \cup H$ is satisfiable (has at least one answer set)
- $B \cup H \models e_1^+ \wedge \ldots \wedge e_n^+ \wedge \text{not } e_1^- \wedge \ldots \wedge \text{not } e_m^-$ where $e_1^+ \ldots e_n^+$ and $e_1^- \ldots e_n^-$ are all positive and negative examples and 'not' denotes negation as failure

Following the two definitions of entailment in answer set programming, we can distinguish *cautious* and *brave* induction which corresponds to using $\models_c$ and $\models_b$ respectively from Definition 2.4. Under cautious induction setting, every positive example has to be extended by all answer sets and every negative example by none of the answer sets. Under brave induction, there has to exist at least one answer set which extends all positive and none of the negative examples.

2.3 ILASP

ILASP (Inductive Learning of Answer Set Programs) is a system for learning ASP programs from examples, developed at Imperial College London by Mark Law, Alessandra Russo

and Krysia Broda [31] [33] [32]. The system can learn Answer Set Programs from partial interpretations, integrates cautious and brave entailment, and supports weak constraints and context dependent examples.

2.3.1 Learning from Partial Answer Sets

In ILASP the search space S_M is specified using a language bias characterised by *mode declarations*. The language bias is defined by a pair of sets $\langle M_h, M_b \rangle$ called head and body mode declarations, which specify the literals that can occur in the corresponding parts of the rules forming the hypothesis space. Every mode declaration $m_h \in M_h$ and $m_b \in M_b$ is a literal with abstracted arguments and those abstractions can be specified to range over typed constants and variables [31].

Example 2.1 (Mode declarations)**.** Let M be equal to

$$\langle \{go(v, c), sleep(v)\}, \{tired(v), bored(v)\} \rangle$$

and the set of values of the constant type c is given by $\{bedroom, garden\}$. Then, among others, the following rules are in S_M:

$$\{go(P, bedroom) \leftarrow tired(P); \quad go(P, garden) \leftarrow bored(P);$$
$$sleep(P) \leftarrow tired(P); \quad sleep(P) \leftarrow tired(P), bored(P)\}$$

and the following are not in S_M:

$$\{go(P, R) \leftarrow tired(P); \quad sleep(garden) \leftarrow tired(P)\}$$

To describe the ILASP learning task, the notion of partial interpretation under answer set semantics has to be introduced.

Definition 2.7 (Partial Interpretation under Answer Set Semantics)**.** "A partial interpretation E is a pair $E = \langle E^{inc}, E^{exc} \rangle$ of sets of ground atoms, called the *inclusions* and *exclusions*. An answer set A extends E if and only if $(E^{inc} \subseteq A) \wedge (E^{exc} \not\subseteq A)$" [31].

In the learning task, both background knowledge B and hypotheses H are expressed as ASP programs and positive examples E^+ and negative examples E^- are expressed as partial interpretations. B, in addition to standard ASP constructs, can contain constraints. A hypothesis H is an inductive solution of a learning task if and only if:

- $H \subseteq S_M$ where S_M denotes the search space (defined by mode declarations M)
- $\forall e^+ \in E^+ \; \exists A \in AS(B \cup H)$ such that A extends e^+
- $\forall e^- \in E^- \; \nexists A \in AS(B \cup H)$ such that A extends e^-

where $AS(P)$ denotes answer sets of ASP program P [31]. Let us notice that there might exist a different answer set extending every positive example. In the above formulation, brave entailment can be expressed through positive examples and cautious through negative, which contrasts with Definition 2.6. By assuming cautious entailment of negative examples it is possible to learn constraints and choice rules. Brave entailment for positive examples 'relaxes' the constraints on the inductive solutions.

2.3.2 Context-Dependent Examples

Learning from context-dependent examples is an extension of the task of learning from answer sets and allows specification of example-specific background knowledge [32]. The two main advantages of using example contexts are better organisation of the background knowledge and improved performance of the learning algorithm.

Definition 2.8 (Context-dependent Partial Interpretation)**.** Context-dependent partial interpretation is a pair $\langle e, C\rangle$ where e is a partial representation and C is an ASP program with no weak constraints, called *context* [32].

2.3.3 Learning Task Definition

Given the definition of context-dependent examples, the definition of the inductive learning task followed the ILASP system can be specified.

Definition 2.9 (Context-dependent Learning from Answer Sets)**.** The task is defined by a tuple $T = \langle B, S_M, E\rangle$ where [32]:

- B is the background knowledge expressed as an ASP program
- S_M is the hypothesis space defined by a mode bias $M = \langle M_h, M_b\rangle$
- $E = \langle E^+, E^-\rangle$ where E^+ and E^- are positive and negative context-dependent examples

A hypothesis H is an inductive solution of T if and only if [32]:

- $H \subseteq S_M$
- for every positive context-dependent example $\langle e, C\rangle \in E^+$ there exists an answer set of $B \cup C \cup H$ which extends e (brave semantics)
- for no negative context-dependent example $\langle e, C\rangle \in E^+$ there exists an answer set of $B \cup C \cup H$ which extends e (cautious semantics)

Example 2.2 illustrates the use of context-dependent examples to learn a common sense knowledge rule from a narrative.

Example 2.2 (ILASP task containing a context-dependent example)**.** Let us consider a narrative *John travelled to the countryside. Mary went to the garden.* Let us assume that there is an implicit timeline in the narrative - chronological order of the events is given by their order in the text. The last argument of the predicates *go*, *travel* and *in* is the time.

Listing 2.1 Representation of a simple narrative as an ILASP task.

```
% Background knowledge:
time(1..3).
earlier(E,L) :- time(E), time(L), E < L.

% Context-dependent example:
#pos(p1, {
  in(mary,y,3)
}, {
  in(mary,y,1)
}, {
```

```
11:   countryside(x).
12:   garden(y).
13:   travel(john,x,1).
14:   go(mary,y,2).
15: }).
16:
17: % Mode declarations:
18: #modeh(1, in(var(person), var(place), var(time))).
19: #modeb(1, travel(var(person), var(place), var(time))).
20: #modeb(1, go(var(person), var(place), var(time))).
21: #modeb(1, earlier(var(time), var(time))).
22:
23: % Maximum number of variables per rule in the hypothesis space:
24: #maxv(4).
25:
26: % Learnt rule:
27: % in(V0,V1,V3) :- go(V0,V1,V2), earlier(V2,V3).
```

The rule learnt by ILASP (line: 26) can be intuitively read as: *an entity is in a certain place at a certain time if it went to that place earlier.*

2.4 Compositional Semantics of Natural Language

2.4.1 The Principle of Semantic Compositionality

The Principle of Semantic Compositionality, also referred to as Frege's principle, is a fundamental idea underlying the compositional approach to natural language semantics, widely attributed to Gottlob Frege. As pointed out in [50], there are numerous formulations referred to as *the principle of compositionality.* The following will be used in this work:

Definition 2.10 (The Principle of Semantic Compositionality)**.** The meaning of a complex expression is a function of the meanings of its constituents together with the method by which those constituents are combined [62].

As pointed in [50], the Principle of Semantic Compositionality makes no assumptions about any particular definition of meaning and it does not provide a method of determining whether meanings of two expressions are the same. Moreover, the principle does not specify what the constituents of a complex expression are and it does not impose any limitations on the combinatory rules. However, the principle constrains the possible meanings of complex expressions.

It is normally assumed that the Principle of Semantic Compositionality ranges over expressions of some particular language. In order to be able to analyse a complex expression as a function of meanings of its constituents and resolve the questions of structure and constituency, syntactic analysis of the expression has to be considered. The meaning of simple expressions - lexical items, is determined by the lexical semantics of a given language. Therefore, the Principle of Semantic Compositionality implies that syntax together with lexical semantics recursively determines the entire semantics of a given language [62].

2.4.2 Montague Grammar

Montague grammar is a theory of semantics of natural language and its relation to syntax, developed by Richard Montague (1930 - 1971). Montague grammar had a profound impact on the contemporary formal semantics for natural language as it was the first formalism that proposed a *systematic* treatment of the semantics of natural language using logic, analogously to semantics of formal languages [27]. The Principle of Semantic Compositionality, truth conditions and model-theoretic approach to semantics are central ideas underlying the theory.

Montague's grammar introduced a systematic relation between syntax and semantics of natural language. Following the principle of compositionality, in Montague's grammar every syntactic rule has to be paired with a semantic rule which determines how meanings of the constituents are combined. Formally, there exists a homomorphism between the syntactic and semantic algebra [49].

Structure of Montague Grammar

Montague proposed two approaches to semantic interpretation of natural language syntax. In the model-theoretic approach, semantic values of natural language expressions are given directly - English is treated as a formal language. The interpretation via translation approach proceeds by specifying a formal language whose semantics is clearly defined, and provides a compositional translation relation[1] from the natural language to the intermediate formal representation on which truth values can be evaluated [45]. According to [47], the structure of classic Montague grammar can be described by:

Syntactic categories and semantic types In Montague grammar, every syntactic expression has an associated *category* and every semantic expression has an associated *type*. There is also a function mapping categories to semantic types [45].

Lexical items and their semantic interpretations In case of the syntactic categories whose expressions include lexical items of a given language, semantics must assign an expression of a corresponding type [48]. Open class lexical items, such as nouns, adjectives, verbs, or adverbs, are translated into constants of an appropriate type. Closed class lexical items, such as prepositions or determiners, are either treated in the same way as open class items or they are given explicit interpretations, as in case of the determiner *every* [48]. Interpretations assigned to lexical items are treated as primitives in the semantic representation and are contained in a *lexicon* of a given language [48].

Syntactic and semantic rules Due to the principle of compositionality, every syntactic combinatory rule has its semantic counterpart - "syntactic and semantic rules come in pairs" [47].

The formal intermediate language used by Montague in [45] to represent the semantics of a fragment of English was *intensional logic*, which was interesting to contemporary linguists

[1] Mapping from syntactic expressions to their semantic counterparts is a relation and not a function because of ambiguity of natural language, which can cause syntactic expression to map to more than one semantic representation.

due to its use of rich set of types and lambda abstractions.

Intensional logic, being higher-order logic, allows variables to range over expressions of different types, as opposed to first order logic, where they can range only over basic entities. This feature of intensional logic is crucial as it allows specification of compositional rules taking sub-expressions of specific types as arguments and having clearly defined meaning, which is necessary in the derivation of meanings of composed expressions [45].

The logic uses lambda abstractions, which makes it possible to formulate higher order functions. Together with the type theory, lambda abstractions allows function-argument application to be treated as "semantic glue" using which meanings of different expressions are combined [49].

Example Translation

To illustrate the use of lambda abstractions and type system in semantic composition, as well as some aspects of homomorphism between algebras of syntax and semantics, let us consider a derivation of a semantic representation of a sentence *Every dog barks*.

The quantifier-forming determiner *every* has a syntactic category[2] $(t/(t/e))/(t/e)$. Its semantic representation has type[3] $\langle\langle e,t\rangle,\langle\langle e,t\rangle,t\rangle\rangle$ and is given by:

$$\lambda P.\lambda Q.\forall x.(P(x) \rightarrow Q(x))$$

The syntactic expression *dog* has category t/e, denoted as CN (common noun). Its corresponding semantic expression has type $\langle e,t\rangle$ and is given by a unary predicate **dog**. Similarly, the intransitive verb *barks* has category t/e, denoted as IV and its corresponding semantic representation has type $\langle e,t\rangle$ and is given by a unary predicate **bark**.

Semantic representation of a noun phrase *every dog* is composed using β - reduction:

$$\underbrace{\lambda P.\lambda Q.\forall x.(P(x) \rightarrow Q(x))}_{\text{“every”}}(\mathbf{dog}) \quad \equiv \quad \lambda Q.\forall x.(\mathbf{dog}(x) \rightarrow Q(x))$$

β - reduction with a unary predicate **bark** yields the final representation of the sentence:

$$\lambda Q.\forall x.(\mathbf{dog}(x) \rightarrow Q(x))(\mathbf{bark}) \quad \equiv \quad \forall x.(\mathbf{dog}(x) \rightarrow \mathbf{bark}(x))$$

Let us notice that *every*, *dog* and *bark* are terminal (leaf) nodes of the syntactic derivation, therefore their meaning is determined by the lexical semantics.

[2] e and t are basic categories corresponding to, respectively, entities and truth values. Expression of composite category A/B produces expression of category A when combined with expression of category B via one of the predetermined syntactic combinatory rules.

[3] Basic types e and t correspond to, respectively, entities and truth values. Composite type $\langle A,B\rangle$ describes function which produces result of type B when argument of type A is applied to it.

2.5 Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG) [59] is a grammar formalism appealing from the point of view of semantic analysis as it is expressive, efficiently parsable and provides a clear interface between syntax and semantics of natural language. CCG belongs to mildly context-sensitive category of grammars, which are between context-free and context-sensitive grammars in Chomsky hierarchy. It is argued that CCG provides an adequate description of the syntactic structure of natural language [59].

In Combinatory Categorial Grammar words are associated with a syntactic *category*, which identifies them as functions, and specifies the number, type and directionality of their arguments and the type of their result [58]. The categorial lexicon is the sole source of language-specific information. It contains mappings from words to their categories paired with the corresponding logical forms which capture word's semantics.

Example 2.3. In what follows, λ-terms providing the semantic interpretation of a category is associated with the syntactic category using ":" operator. Using that notation, an intransitive verb *swim* has the following entry in the categorial lexicon:

$$swim := S \setminus NP : \lambda x.swim'\ x$$

which identifies the verb as a function from (subject) noun phrases (NP) to sentences (S), and the backward slash indicates that the argument of the verb occurs to the left of *swim* in the sentence. Similarly, a transitive verb *admire* has the entry:

$$admire := (S \setminus NP)/NP : \lambda x.\lambda y.admire'\ xy$$

which identifies it as a function from (object) noun phrases to intransitive verbs and the forward slash specifies that the argument has to occur to the right of *admire* in the sentence.

2.5.1 Combinatory Rules

In Combinatory Categorial Grammar, categories can combine using a number of combinatory operations and assemble the meaning of phrases or sentences from its constituents, following the Principle of Semantic Compositionality. The type of combinatory rule applicable to combine semantic representations depends on constituents' categories.

Application rules

The simplest combinatory rules correspond to functional application. In the following, X and Y range over the syntactic categories [59].

Forward Application: (>)

$$X/Y : f \qquad Y : a \implies X : fa \tag{2.1}$$

Backward Application: (<)

$$Y : a \qquad X \setminus Y : f \implies X : fa \tag{2.2}$$

Example 2.4 (Application rules)**.** The functional application rules is applied to derive a compositional interpretation of a sentence *"Jack runs marathons."* as follows:

$$\dfrac{\dfrac{Jack}{NP : jack'} \quad \dfrac{\dfrac{runs}{(S \setminus NP)/NP : \lambda x \lambda y.run'\ xy} \quad \dfrac{marathons}{NP : marathons'}}{S \setminus NP : \lambda y.run'\ marathons'\ y}>}{S : run'\ marathons'\ jack'}<$$

Coordination rule

Another combinatory rule, coordination, allows constituents of the same category to conjoin and yield a single constituent of the given category [58]. In the following, *conj* denotes conjunction e.g. *and, or, but.*

$$\begin{aligned} &\text{Coordination: } (< \& >) \\ &X \quad conj \quad X \implies X \end{aligned} \tag{2.3}$$

Example 2.5 (Coordination rule)**.** The coordination and application rules is applied to derive a compositional interpretation of a sentence *"Jack tried and succeeded."* as follows:

$$\dfrac{\dfrac{Jack}{NP : jack'} \quad \dfrac{\dfrac{tried}{S \setminus NP : \lambda x.try'\ xy} \quad \dfrac{and}{conj} \quad \dfrac{succeeded}{S \setminus NP : \lambda x.succeed'\ x}}{S \setminus NP : \lambda x.try'\ x \wedge succeed'\ x} < \& >}{S : try'\ jack' \wedge succeed'\ jack'}<$$

Composition rules

CCG also includes composition rules, which allow two functor categories whose domain and range match to combine and form another functor. There are four basic composition rules in order to account for different combinations of argument directionalities [26].

$$\begin{aligned} &\text{Forward Composition: } (> \mathbf{B}) \\ &X/Y : f \qquad Y/Z : g \implies X/Z : \lambda x.f(gx) \end{aligned} \tag{2.4}$$

$$\begin{aligned} &\text{Backward Composition: } (< \mathbf{B}) \\ &Y \setminus Z : g \qquad X \setminus Y : f \implies X \setminus Z : \lambda x.f(gx) \end{aligned} \tag{2.5}$$

$$\begin{aligned} &\text{Forward Crossing Composition: } (>_{\times} \mathbf{B}) \\ &X/Y : f \qquad Y \setminus Z : g \implies X \setminus Z : \lambda x.f(gx) \end{aligned} \tag{2.6}$$

$$\begin{aligned} &\text{Backward Crossing Composition: } (<_{\times} \mathbf{B}) \\ &Y/Z : g \qquad X \setminus Y : f \implies X/Z : \lambda x.f(gx) \end{aligned} \tag{2.7}$$

Type-raising rules

Combinatory grammars include type-raising rules, which turn argument categories into functions over functions over such categories. These rules allow constituents to compose e.g. in coordination. As stated in [59], type raising represents the grammatical notion of

(nominative, accusative, etc.) case and theoretically should be applied only to complements of verbs (such as *Jim* and *Jack* in Example 2.6).

$$\begin{array}{l}\text{Forward Type-raising: } (>T)\\ X:a \implies T/(T\setminus X):\lambda f.fa\end{array} \tag{2.8}$$

$$\begin{array}{l}\text{Backward Type-raising: } (<T)\\ X:a \implies T\setminus(T/X):\lambda f.fa\end{array} \tag{2.9}$$

Let us notice that rather than introducing type-raising rules, the lexicon could be expanded and include all the raised categories that are otherwise defined by the type-raising rules. Also all functions into the categories which can be type-raised will require category of functions into the raised categories. However, such solution considerably increases the size of the lexicon hence, is less efficient from the computational point of view.

Example 2.6 (Composition, type-raising and coordination)**.** Derivation of compositional interpretation for a sentence *"Jim offered and Jack accepted help."* involves application of forward type-raising, forward composition and coordination rules. The example was adapted from [58]. For the interest of clarity, semantic representations for the lexical items are skipped.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Jim}{NP}}{S/(S\setminus NP)}>T\ \dfrac{offered}{(S\setminus NP)/NP}}{S/NP:\lambda x.offer'x\ jim'}>B\quad\dfrac{and}{CONJ}\quad\dfrac{\dfrac{\dfrac{Jack}{NP}}{S/(S\setminus NP)}>T\ \dfrac{accepted}{(S\setminus NP)/NP}}{S/NP:\lambda x.accept'x\ jack'}>B}{S/NP:\lambda x.offer'x\ jim'\wedge accept'x\ jack'}<\&>\dfrac{help}{NP}}{S:offer'help'jim'\wedge accept'help'\ jack'}$$

2.5.2 Comparison of CFGs and CCGs

Due to the wide applicability of context free grammars, for example for specifying syntax of programming languages, it would be instructive to compare CFGs and CCGs and see why the latter formalism might be more convenient for parsing natural language.

In linguistics terms, both context free and combinatory categorial grammars fall under broader category of *phrase structure grammars*. The term was introduced by Noam Chomsky as a term for grammars defined by phrase structure rules - rewrite rules of the form $\alpha \to \beta$, where $\alpha \subseteq V^*\ N\ V^*$ (V - finite set of symbols, vocabulary of the grammar, N - non-terminal grammar symbols) and $\beta \subseteq V^*$ is a sequence of labels with which α can be replaced when generating constituent structure of a sentence [20]. In a phrase-structure grammar every word belongs to a certain lexical category. Constituents are words or groups of words that function together as a unit or phrase and form larger grammatical structures [64].

Categorial grammars put into the lexicon most of the information that in context-free grammars is captured by the phrase structure rules. Context free grammars are constrained to the rewrite rules which belong to $N \times V^*$, heads of the rules must be (single) non-terminal grammar symbols. In categorial grammars, categories associated with constituents identify them as either functions or arguments, specification of grammatical rules is transferred from

	CFGs	CCGs
Combination operations	Many	Few
Parse Tree Nodes	Non-terminals	Categories
Syntactic symbols	Few dozen	Handful, but can combine
Paired with words	POS tags	Categories

Table 2.1: Comparison of the most important properties of context free and combinatory categorial grammars [2].

phrase structure rules to the lexicon [59]. The size of the lexicon is an important measure of a grammar's complexity.

When the combinatory rules of categorial grammar are limited to functional application rules, the categorial grammar is "nothing more than a context-free grammar" [59, p. 34] written in a different form – rather than specifying how constituents are produced, the category and application order of arguments of each constituent is given. However, additional combinatory rules (coordination and composition) increase the generative power of CCGs relative to CFGs and allow the former to represent natural language phenomena, such as unbounded long-range dependencies,[4] that the latter, in general, cannot tackle. The main differences between context free and combinatory categorial grammars are summarised in Table 2.1.

2.5.3 Syntactic Parsing with CCGs

Syntactic parsing is a process of deriving a representation of a structure of a sentence from a raw textual input. Parsing with combinatory categorial grammar can be divided into three stages:

- Assignment of part-of-speech tags and CCG categories to words in a sentence. Process of assigning CCG categories is called *supertagging* and often constitutes a bottleneck on parser's performance, especially for bottom-up parsing algorithms. Supertagging is done using statistical models, such as maximum entropy models [15] or long-short term memory recurrent neural networks [36].
- Combination of categories using the combinatory rules. A standard algorithm for performing this step is bottom-up CKY chart-parsing algorithm. The algorithm uses a chart, which is a tabular data structure, for storing constituents spanning each subsequence of words of a parsed sentence [14].
- Highest scoring derivation selection is performed according to a parsing model, such as perceptron or maximum entropy model, trained over a large data corpus.

A notable example is the C&C parser by Clark and Curan [15] which used log-linear CCG parsing model to discriminate between multiple parses of a sentence. The parser was trained on CCGbank, which is a translation of Penn Treebank[5] corpus into a corpus of CCG

[4]Dependencies within the same sentence with arbitrary distance between the head and the argument. Unbounded long-range dependencies can be introduced by relative clauses where the missing NP is arbitrarily deeply embedded, for example: *The sushi that [you told me [John saw [Mary eat]]].*

[5]Large annotated text corpus of American English containing over 4.5 million words, created at the University of Pennsylvania. Words in the corpus are annotated with part-of-speech tags and sentences with

derivations [25]. The EasySRL parser, which uses LSTM recurrent neural networks for supertagging, achieved state-of-the-art F1 measure (88.1%) for parsing word dependencies when evaluated on the CCGBank test set [36].

2.6 Semantic Parsing

Semantic parsing is a task of deriving a semantic representation of a sentence, usually expressed in some logical formalism, from textual input. Over the years, a number of approaches to this problem have been proposed, varying in the overall system architecture, statistical models used and type of supervision.

2.6.1 Boxer System

Boxer system[6] proposed by Bos et al. [10] uses CCG derivation structure produced by the C&C parser [15] to generate a first-order logic representations using λ-calculus. In order to derive the semantic representation, 245 most frequent CCG categories were annotated with the corresponding λ-expressions, following Montague-style semantics. For example, transitive verbs received the following annotation (*lexeme* is replaced with a lexeme of a specific transitive verb):

$$(S \setminus NP)/NP : \lambda x.\lambda y.lexeme' xy \tag{2.10}$$

Moreover, the combinatory rules of the CCG grammar were reformulated in terms of the target semantic representation.

To derive a semantic representation of a given sentence, semantic representation is assigned to each lexical item and β-conversion is applied to the constituents according to the structure of the CCG derivation in a bottom-up manner. The authors in [10] point out that the same methodology could be applied to formalisms different than first-order logic.

dependency structures. The corpus contains text from many different categories, most of it however was drawn from newswire stories. The corpus is very valuable for construction of statistical models for the grammar and evaluation and comparison of parsing models [41].

[6] A further overview of the Boxer system and a comparison to the approach developed in this project are provided in Section 7.1.2.

Chapter 3

Translation

In this chapter, the process of translating from English to a set of ASP rules is described in detail. First, an overview of the target semantic representation is given together with motivations for the choice of the target formalism. Then, λ-ASP expressions [5], which serve as an intermediate representation in the translation process, are introduced. The introduction is followed by a description of how λ-ASP expressions allow for building the target semantic representation compositionally in a bottom-up fashion. Presentation of λ-ASP Calculus is concluded with a description of some more complicated translation problems, such as handling non-local word dependencies. Finally, the process of translating from λ-ASP Calculus intermediate form to the target ASP representation is described.

3.1 Semantic Representation

The objective of the translation procedure is to represent an input text, written in English, as a set of ASP rules. In order to enable the use of the semantic representation both for inference and learning, it had to be constructed in a way that would allow for composition of meaning using multiple predicates in the translation phase and for flexible formulation of the hypothesis space using mode declarations when used in the learning setting. These two requirements were addressed by constraining the set of predicates used in the translation process, dividing predicates into categories, and adding meta-level information to each predicate in order to allow for composition of meaning.

The chosen representation consists of four categories of predicates: **nominals**, **events**, **modifiers** and **prepositions**, which can have different arities (from one to four). Predicate categories are closely tied to the elements in the structure of a sentence. Intuitively, regarding correspondence to parts of speech, nominals roughly correspond to nouns, events to verbs and modifiers to adjectives or adverbs. The main motivation for such selection of predicate categories was to have a small set of predicates (in practice, only around 10 predicate names are used in our translation) which provide high-level information about the semantics of the corresponding word or phrase.

Predicate names are formed by concatenating an arity prefix (*unary*, *binary*, *ternary*, *quadrary*) with a category name. The arity of each predicate is equal to the number of arguments of the CCG category of the corresponding word plus extra meta arguments used

to enrich predicate semantics and allow for composition of meaning. The meta arguments are *lemma* and *identifier*. For a given predicate, *lemma* is the lemma of the corresponding word and, except from certain modifiers, lemma is the second argument of a predicate. Identifier, if present, is the first argument of a predicate and is used to reference one predicate from another. The only predicates which do not take an identifier as their argument are modifiers whose meaning is not built compositionally.

Example 3.1 (Predicate corresponding to a ditransitive verb)**.** For a ditransitive verb *gave*, as used in a sentence:

Jack ***gave*** *a flower to Jenny.*

the corresponding predicate is: $ternaryEvent(e0, give, c0, n0, c1)$, where $e0$ is the identifier, *give* is the lemma and $c0, n0, c1$ are constants corresponding to *Jack*, *flower* and *Jenny* respectively. Let us note the use of *ternary* as the arity prefix. This is caused by the fact that, in this case a ditransitive verb *give* has a CCG category $((S[dcl] \setminus NP)/PP)/NP)$ which takes three arguments.

Example 3.2 (Modifier with non-compositional semantics)**.** Predicates corresponding to an adverb *quickly* and transitive verb *eat*, as used in a sentence:

Jack ***ate*** *a sandwich* ***quickly****.*

are given by: $unaryModif(quickly, e0)$ and $binaryEvent(e0, eat, c0, n0)$ where $c0, n0$ are constants corresponding to *Jack* and a *sandwich* respectively. Let us notice how semantics of *eat quickly* is built compositionally by referencing the predicate corresponding to *ate* from within the predicate corresponding to *quickly*. A complete ASP representation of the above sentence is given by the following set of facts:

$$\begin{aligned} C = \{&unaryModif(quickly, e0).\\ &binaryEvent(e0, eat, c0, n0).\\ &unaryNominal(n0, sandwich).\\ &unaryNominal(c0, jack).\\ &metaData(e0, 0).\} \end{aligned}$$

Let us notice the use of *metaData* predicate in the representation of the sentence in Example 3.2. Currently, this is the only category of predicate generated by the translation algorithm other than the four categories already described. The aim of this predicate is to add information about relative position of events in the text, which is significant when analysing temporal relations in narratives. Similar predicates could be generated to further enrich the representation and embed other type of syntactic information, such as tense, however they were not found useful at the current stage of the project. In the next examples, *metaData* predicates will be omitted for the sake of clarity.

The use of lemmas as predicate arguments makes the representation more expressive than if different predicate names were used for different word lemmas.[1] An example of a rule that the current representation allows us to formulate, and that we would not be able to

[1] If the second option was followed (different predicate names for different word lemmas) then *eat* in Example 3.2 could translate to $eat(e0, c0, n0)$.

formulate if we used the other representation is:

$$unaryNominal(C1, T) : -eq(C1, C2), unaryNominal(C2, T).$$

which roughly states that *if two entities are equal to each other, then they are of the same class*, where T is the class in question. From the learning perspective, the chosen representation allows for elegant and efficient specification of mode declarations encapsulating multiple words of the same type (e.g. synonyms), which is significant for predicate clustering (section 4.3.4). For example, to specify that transitive verbs *get*, *take* or *grab* could occur in a body of a hypothesis, the following mode would be sufficient:

$$modeb(1, binaryEvent(var(v0), const(c1), var(v2), var(v3)))$$

where $c1$ assumes values from the set $\{get, take, grab\}$. If predicates with different names were used, three different mode declarations would be required.

3.2 CCG Parse Tree to λ-ASP Calculus Translation

As described in Section 2.5, one of the main advantages of using CCG for semantic parsing is its clear interface between syntax and semantics. However, in order to be able to emulate the combinatory rules that CCG uses to compose meaning, an intermediate representation close in syntax to ASP, however allowing for such compositions, was required. This was the primary motivation for using λ-ASP expression, idea proposed in [5], which has been adapted and significantly extended for the purpose of the current project.

3.2.1 λ-ASP Calculus Primitives

In the version of λ-ASP expressions used throughout this project, λ-ASP expressions have the following structure:

$$\underbrace{\#X_1.\#X_2.\ldots.\#X_k.}_{k\ \ abstractions}\underbrace{p_1(X_{i_{1,1}},\ldots X_{i_{1,a_1}})\ldots p_n(X_{i_{n,1}},\ldots X_{i_{n,a_n}})}_{n\ \ predicates}$$
$$\underbrace{X_{j_{1,0}}@(X_{j_{1,1}},\ldots X_{j_{1,b_1}})\ldots X_{j_{m,0}}@(X_{i_{m,1}},\ldots X_{j_{m,b_m}})}_{m\ \ applications}$$

where a_i is the arity of the i^{th} predicate and b_j is the number of arguments taken by the j^{th} application. Primitives from which λ-ASP expressions are constructed can be described as follows:

Abstractions Abstractions provide a parametrisation of a λ-ASP expression. Syntactically, they are represented by a list of variables prefixing the formula. Semantically, the list specifies the arguments that a given λ-ASP expression takes and the order in which they are applied. Abstractions enclosed in $[\cdot]$ are the heads of the expression. Intuitively, expression's heads are identifiers of an expression used when the semantics is composed.

Predicates Predicates, together with applications (described next) form the body of a λ-ASP expression. Intuitively, predicates convey the expected semantics of the λ-ASP expression as eventually they are, almost directly, translated to ASP predicates forming the final representation.

Application Applications are the key mean used for semantic composition. They allow precise specification (at the level of predicate arguments) of how λ-ASP expressions should be combined to form the expected semantics. In general, applications' syntax can be expressed by: $X@(A_1, \ldots, A_n)$ where X can be thought of as a function and $A_1, \ldots, A_n$ as arguments. An application is said to be instantiated when the values of X and $A_1, \ldots, A_n$ are supplied. When the function and arguments are supplied, semantics is obtained by applying the latter to the former.

Example 3.3 (λ-ASP expressions)**.** Let us consider the following sentence:

Jack decided to ***give*** *Jane a flower.*

The ditransitive verb *give* has CCG category $(((S[b]\backslash NP)/NP)/NP)$ and the corresponding λ-ASP expression is given by:

$$\#[I].\#L.\#X2.\#X1.\#X0.\ ternaryEvent(I, L, X0, X1, X2), (X2), (X1), (X0) \tag{3.1}$$

Let us notice that the abstraction I is a place holder for the identifier of the expression, similarly as L is a place holder for the lemma. Moreover, I plays the role of a head of the expression. When used as predicate arguments, abstractions $X2, X1, X0$ are replaced with lexical heads of the provided expressions. Let us notice that $X2, X1, X0$ also occur as applications with no arguments, in which case they are simply treated as place holders for the corresponding predicates. If identifier $e0$ and lemma *give* (in that order) are applied to the expression 3.1, the following expression is obtained:

$$\#X2.\#X1.\#X0.\ ternaryEvent(e0, give, X0, X1, X2), (X2), (X1), (X0) \tag{3.2}$$

When instantiated[2] λ-ASP expressions:

$$unaryNominal(c1, jane),\ unaryNominal(n0, flower),\ unaryNominal(c0, jack)$$

are applied (in that order) to 3.2, the following instantiated λ-ASP expression is obtained:

$$ternaryEvent(e0, give, c0, n0, c1), unaryNominal(c1, jane),$$
$$unaryNominal(n0, flower), unaryNominal(c0, jack)$$

Example 3.4 (λ-ASP expression with application)**.** Let us consider the same sentence as in Example 3.3, however let us now focus on the verb *decide*, which requires coindexation (Section 3.3.4). In such case, use of argument-taking application is required. The CCG category of the verb *decide* is $((S[dcl] \setminus NP)/(S[to] \setminus NP))$ and the corresponding λ-ASP expression is:

$$\#[I].\#L.\#X1.\#X0.\ binaryEvent(I, L, X0, X1), (X1)@(X0) \tag{3.3}$$

After applying identifier $e1$ and lemma *decide* to 3.3, the following expression is obtained:

$$\#X1.\#X0.\ binaryEvent(e1, decide, X0, X1), (X1)@(X0) \tag{3.4}$$

[2] λ-ASP expression is said to be instantiated if it has no abstractions. Otherwise, it is said to be abstract.

When λ-ASP expression:

$$\#X0.ternaryEvent(e0, give, X0, n0, c1), unaryNominal(c1, jane), \\ unaryNominal(n0, flower)$$

corresponding to the partially instantiated λ-ASP expression for the verb *give* is applied to 3.4, the following expression is obtained:

$$\#X0.binaryEvent(e1, decide, X0, e0), \\ (\#X0'.ternaryEvent(e0, give, X0', n0, c1), unaryNominal(c1, jane), \\ unaryNominal(n0, flower), X0')@(X0) \tag{3.5}$$

Let us notice that both the predicate and the application will be passed the same argument - the value of $X0$. Hence, when instantiated λ-ASP expression $unaryNominal(c0, Jack)$ is applied to 3.5, a fully instantiated λ-ASP expression is derived:

$$binaryEvent(e1, decide, c0, e0), ternaryEvent(e0, give, c0, n0, c1), \\ unaryNominal(c1, jane), unaryNominal(n0, flower), unaryNominal(c0, Jack)$$

Two facts should be noted about λ-ASP expressions. First of all, λ-ASP expressions used in this project are not typed and their semantics does not distinguish between variables and constants. The only distinction is made between *abstractions* and *instances*, with the latter being provided in place of the former using application. Secondly, recursive applications *are* allowed, application primitives can take as an argument a λ-ASP expression which has application primitives in its body, and so on. However, the expressive power of such formulation was not checked beyond the immediate requirements of the project.

3.2.2 Lexicon

In the context of the current project, lexicon refers to an *algorithm* for deriving the (intermediate) semantic representation for a single word given its corresponding syntactic information. More precisely, given a word associated with annotations such as lemma, part-of-speech tag, CCG category and co-reference information, lexicon returns a λ-ASP expression for such word. Let us notice that in other places, like for example [70], lexicon is a *mapping* between lexemes and pairs of CCG category and semantic representation. The different treatment of the term *lexicon*, i.e. algorithm rather than static mapping, results from the fact that in the current project the semantic representation is derived algorithmically rather than learned from annotated data like in [70]. However, the main role of a lexicon in both cases, namely a mechanism for deriving semantics given single words as input, is the same, hence the same term is used.

Following the Principle of Semantic Compositionality, the meaning of phrases and sentences is built in a bottom-up fashion starting from the leaf nodes of a CCG parse tree. Leaf nodes correspond to single words from a sentence, and their semantic representation is obtained from the lexicon. In most cases, λ-ASP expressions are assigned to words based on their lemma, part-of-speech tag and CCG category. However, in some cases, like for example the determiner *the*, further information like co-reference annotations are required.

In order to make the lexicon more robust to parser errors, part-of-speech tags are used to assign a word to one of five categories, namely *nominals*, *verbs*, *modifiers*, *prepositions* or *interrogative words*. Within each of these sub-lexicons, the CCG category of a word and lemma are used to derive the semantic representation. Relying on part-of-speech tags to perform coarse division of words has the advantage that state-of-the-art part-of-speech taggers have significantly higher accuracy than CCG supertaggers. When the word gets assigned to a correct sub-lexicon, in many cases it is still possible to retrieve a correct semantic representation even if the CCG category is incorrect (however, the CCG category needs to have correct arity). In Table 3.1, lexical entries for some words occurring in the *The (20) QA bAbi Tasks Dataset* are provided.

Word	**POS tag**	**Lemma**	**CCG Category**	**λ-ASP expression**
cats	NNS	cat	N	$[X0]\#X0.unaryNominal(X0, cat), (X0)$
are	VBP	be	$(S[dcl] \setminus NP)/(S[adj] \setminus NP)$	$[X0]\#X0.(X0)$
blue	JJ	blue	$S[adj] \setminus NP$	$[blue]\#X0.unaryModif(blue, X0), (X0)$
went	VBD	go	$(S[dcl] \setminus NP)/PP$	$[e0]\#X1.\#X0.(X1), (X0),$ $binaryEvent(e0, go, X0, X1)$
to	TO	to	PP/NP	$[X0]\#X0.(X0)$
inside	IN	inside	$((S \setminus NP) \setminus (S \setminus NP))/NP$	$[X1]\#X2.\#X1.\#X0.(X1)@(i0), (X2), (X0),$ $binaryPrep(i0, inside, X2, X0)$
a	DT	a	NP/N	$[X0]\#X0.(X0)@(n0)$

Table 3.1: Lexicon entries for selected words from *The (20) QA bAbi Tasks Dataset*.

3.2.3 Semantic Composition

As mentioned in Section 3.2.2, the guiding principle when deriving a semantic representation of a sentence is the Principle of Semantic Compositionality. The order in which λ-ASP expressions are composed, using application, is dictated by the CCG parse tree. The semantic representations for the leaf nodes of a CCG parse tree are derived using the lexicon. In order to derive the semantics for internal nodes, two steps have to be taken. First, a combinatory rule (Section 2.5.1) used to combine CCG categories of child nodes of the internal node has to be determined. This can be done by analysing the left and right subcategories of child nodes' categories together with their *separator* (/ or \). Such information allows to identify a child node whose λ-ASP expression should serve as a *function*. If the other child node is present (CCG parse tree is a binary tree), its λ-ASP expression serves as an *argument*. Then, the *argument* is applied to the *function* and λ-ASP expression for the internal node is derived. Such procedure is applied bottom-up, starting from the leaf nodes, until the λ-ASP expression for the root node is derived.

Example 3.5 (Bottom-up derivation of internal node's λ-ASP expression)**.** Let us consider a sentence: *Jack gave Jane a flower*, which has the following CCG parse tree:

Listing 3.1 CCG parse tree of a sentence *Jack gave Jane a flower*

```
1: <T S[dcl]>
2:   <L NP NNP Jack>
3:   <T (S[dcl]\NP)>
```

```
4:      <T ((S[dcl]\NP)/NP)>
5:        <L (((S[dcl]\NP)/NP)/NP) VBD gave>
6:        <L NP NNP Jane>
7:      <T NP>
8:        <L (NP/N) DT a>
9:        <L N NN flower>
```

In Listing 3.1, nodes labelled with `L` (lines: $2, 5, 6, 8, 9$) are leaf nodes and and nodes labelled with `T` (lines: $1, 3, 4, 7$) are internal nodes. Let us consider node `<T NP>` (line: 7).

From the CCG categories of the child nodes (lines: $8, 9$) it can be deduced that the categories are combined using *forward application* (Section 2.5.1). λ-ASP expression for determiner *a* assumes the role of a function and semantics of the noun *flower* plays the role of an argument. Based on Table 3.1, it is known that the corresponding λ-ASP expressions are: $\#X0.(X0)@(n0)$ and $\#X0.unaryNominal(X0, cat), (X0)$. The resultant expression is given by:

$$\begin{aligned}
&(\#X0.(X0)@(n0))@(\#X0'.unaryNominal(X0', flower), (X0')) \\
&\quad \equiv \ (\#X0.unaryNominal(X0, flower), (X0))@(n0) \\
&\quad \equiv \ unaryNominal(n0, flower)
\end{aligned}$$

Let us notice that the application mechanism of λ-ASP expressions allows uniform treatment of different combinatory rules of CCG in a sense that semantic composition is always realised using application, which is important as, according to the presentation in [26], there are thirteen different combinatory rules. The only element of semantic composition mechanism described above which changes per each rule is identification of *function* and *argument*.

3.3 Specific Translation Problems

Let us now consider some more involved translation problems that were encountered during the project. In all cases, it will be described how λ-ASP expressions can be used to provide the (intermediate) semantic representation.

3.3.1 Noun Definiteness

It is important to distinguish between definite and indefinite nouns as the former can provide additional semantic information to a given set of sentences. For example, for the narrative:

*Jack went to **the** kitchen to grab some milk.*
*Jim went to **the** kitchen to check if his dinner was ready.*

the use of definite article *the* allows us to conclude that Jack and Jim are in the same location. In order to handle the semantic difference between definite and indefinite nouns, part-of-speech tag, co-reference information and, if present, article were taken into account.

A noun is classified as definite either when it is a proper noun, which can be determined

based on its part-of-speech tag,[3] or when definite article *the* is a sister of the given node or any of its ancestors in the CCG parse tree. λ-ASP expression for a definite noun is given by: $\#L.\#[I].unaryNominal(I, L)$. L is given by the lemma of the corresponding word and I is equal to either c_n where n is the co-reference cluster index, when the co-reference information is present for the given noun phrase, or $I = L$ when no co-reference information is absent. Using c_n or L as identifier reflects the intended semantics of a definite noun phrase as referring to a specific entity in the domain of discourse.

Common nouns can be distinguished by their part-of-speech tag[4] and the λ-ASP expression that they are assigned by the lexicon is $\#L.\#[I].unaryNominal(I, L), (I)$. Let us notice the use of (I) place holder after the predicate, which was introduced to allow compositional derivation of semantics for phrases like *traveller Jack*. When indefinite article *a* is a sister of a common noun node or of any of its ancestors in the CCG parse tree, it has an effect of applying a Skolem constant to the noun phrase, as it could be seen in Example 3.5 for a noun phrase *a flower*. Application of a Skolem constant conveys the expected semantics of indefinite noun referring to *some* entity in the domain of discourse.

Let us notice that in the project semantics of articles different than *a* and *the* was not addressed. Another fact worth pointing out is that, unless a node corresponding the a determiner is a sister of some node on a path from the common noun node to the root of the parse tree, the semantic representation of the common noun will not be fully instantiated - identifier I will be a variable. Such nouns are implicitly assumed to be *generic nouns* and are important when considering generic statements.

3.3.2 Generic Sentences

Generic sentences communicate generalisations about the world. Examples of such such statements are *Turtles live long*, *Tigers are carnivorous* or *Jack likes turnips for dinner*. They often express some form of common knowledge about the world and are a subject of studies both among linguists and philosophers.

In the project, generic statements are translated both as (unground) rules, which corresponds to universal quantification, and sets of facts, which correspond to the Skolemised version of the aforementioned rules. The main difficulty with translating generic sentences was selection of a rule head from a set of predicates returned as an output of the translation algorithm. To that aim, information about the head of the λ-ASP expression was utilised - a predicate whose identifier is equal to the head of a fully-composed λ-ASP expression is chosen as a head of the corresponding rule. Let us notice that due to the use of *semantic* predicates (Section 4.3.1), exceptions to generic statements can be specified by providing a rule whose head is a relevant abnormality predicate.

Example 3.6 (Translation of a generic statement)**.** Let us consider a sentence:

Lions eat meat.

[3] In Penn Treebank Project, singular proper nouns are assigned tag NNP and plural are assigned NNPS. This type of annotations is used by Stanford CoreNLP.

[4] In Penn Treebank Project, singular common nouns have tag NN and plural common nouns have tag NNS.

Transitive verb *eat* has CCG category $(S[dcl] \setminus NP)/NP$ and λ-ASP expression:

$$[e0]\#X1.\#X0.binaryEvent(e0, eat, X0, X1), (X0), (X1) \tag{3.6}$$

As discussed in section 3.3.1, *lions* and *meat* are generic nouns, and their λ-ASP expressions are: $\#[X0'].unaryNominal(X0', lion), (X0')$ and $\#[X1'].unaryNominal(X1', meat), (X1')$ respectively. When the former and the latter (in that order) are applied to 3.6, the following λ-ASP expression is obtained:

$$[e0]\#X1'.\#X0'.binaryEvent(e0, eat, X0', X1'), unaryNominal(X0', lion), \\ unaryNominal(X1', meat), (X0'), (X1') \tag{3.7}$$

Let us notice that the head of expression (3.7) is $e0$, which is an identifier of the predicate corresponding to the word *eat*. Therefore, the following ASP rule (not λ-ASP expression) and a set of facts corresponding to the Skolemised rule are obtained (using the method described in Section 3.4):

$$binaryEvent(e0, eat, X0, X1) :- unaryNominal(X0, lion), unaryNominal(X1, meat)$$
$$binaryEvent(e0, eat, sk0, sk1).\ unaryNominal(sk0, lion).\ unaryNominal(sk1, meat).$$

Let us notice that for the rule translation mechanism to work, uniqueness of names of predicate identifiers within a sentence is required. Let us also notice that the translation mechanism assumes that the entire sentence is generic, sentences like:

Jack knows that turnips are healthy.

were not considered in the project. Moreover, truth conditions of generic statements are a separate topic that was outside the scope of the project.

3.3.3 Coordination

The first of the more complex syntactic structures supported by the translation algorithm that will be discussed is coordination. In linguistics, coordination refers to a syntactic structure in which two or more *conjuncts* are linked together. In English, coordination is often introduced by a *coordinator*, such as *and*, *or*, *but*, which in are assigned category CONJ in CCG.

In order to support generating λ-ASP expressions for sentences including coordination, λ-ASP formalism was extended to allow expressions to have multiple heads, which essentially reflects the basic semantics of symmetric coordination. When argument with n heads is applied to a function (both of which are λ-ASP expressions), the resultant λ-ASP expression consists of n copies of the function, each one with a different head applied to it. The head of the resultant expression is given by the head of the function.

In the project, we focused on coordination of constituents that have the same category, as in sentences: *[Jack] and [Jim] went fishing* or *Jack [applied for] and [received] a bonus pay.* In such cases, the CCG category of the parent node of the coordinator in the CCG parse tree has form $X \setminus X$, where X is the category of the coordinator's sister node, which is one of the conjuncts. Therefore, the coordinator needs to have a λ-ASP expression that functionally behaves the same as expressions corresponding to category $X \setminus X$, but also takes the other

conjunct as an argument, hence has one argument more than the parent node. Pseudocode of lexicon's function responsible for generating λ-ASP expressions for coordinators in given in Listing 3.2.

Listing 3.2 Pseudocode of a function generating λ-ASP expressions for coordinators

```
1: procedure coordinator_semantics(node):
2:   n_args = node.parent().category().n_args()
3:   abstractions = empty_list()
4:
5:   for i := 1 to n_args − 1:
6:     abstractions.append(var(i))
7:
8:   return lambda_asp(
9:     #[C2].#[C1].#abstractions.C1@(abstractions), C2@(abstractions))
```

Example 3.7 (Verb phrase coordination)**.** Let us consider the sentence: *Jack washed and ironed a shirt*, whose CCG parse tree is presented in Listing 3.3.

Listing 3.3 CCG parse tree of a sentence *Jack washed and ironed a shirt.*

```
1: <T S[dcl]>
2:   <L NP NNP Jack>
3:   <T (S[dcl]\NP)>
4:     <T ((S[dcl]\NP)/NP)>
5:       <L ((S[dcl]\NP)/NP) VBN washed>
6:       <T (((S[dcl]\NP)/NP)\((S[dcl]\NP)/NP))>
7:         <L CONJ CC and>
8:         <L ((S[dcl]\NP)/NP) VBN ironed>
9:     <T NP>
10:      <L (NP/N) DT a>
11:      <L N NN shirt>
```

Using the procedure from Listing 3.2, the λ-ASP representation for node `<L CONJ CC and>` (line: 7) is given by:

$$\#[C2].\#[C1].\#X1.\#X0.(C1)@(X1,X0),(C2)@(X1,X0) \tag{3.8}$$

Applying the λ-ASP expressions corresponding to nodes in lines 8 and 5, we get:

$$\begin{aligned}[e0,e1]\#X1.\#X0.(&binaryEvent(e1,iron,X2',X3'),(X2'),(X3'))@(X1,X0),\\ &(binaryEvent(e0,wash,X0',X1'),(X0'),(X1'))@(X1,X0)\end{aligned} \tag{3.9}$$

The final λ-ASP expression is obtained by applying expressions corresponding to nodes in lines 9 and 2 (in that order) and is given by:

$$\begin{aligned}&binaryEvent(e1,iron,c0,n0),binaryEvent(e0,wash,c0,n0),\\ &unaryNominal(c0,jack),unaryNominal(n0,shirt)\end{aligned} \tag{3.10}$$

Let us notice that the method of handling coordination works the same for all kinds of coordinators, which means that λ-ASP representation can be produced for all coordinators linking conjuncts of the same CCG category. However, some coordinators come with additional meaning, like for example *or*, which presents an alternative, as in a sentence:

Jack wanted to order steak or ribs.

Handling of additional semantic information corresponding to different coordinators is a possible area for future work.

3.3.4 Non-local Dependencies

As opposed to local dependiencies, where arguments are attached directly to predicates in the dependency structure, non-local dependencies can be thougt of as the ones where argument of a predicate is attached to some other predicate. Non-local dependencies are a subject of extensive studies among linguists due to their significance for capturing predicate-argument structure of more complex sentences. Non-local dependencies occur due to different linguistic constructions, such as control, raising and relativisation. In order to generate λ-ASP expressions for such constructions, the idea of co-indexation, as described in [25] is used. The co-indexation mechanism is implemented using application primitives of λ-ASP expressions.

Raising constructions involve movement of an argument from a subordinate clause to the main clause. Raising verbs, such as *seem* or *appear*, take infinitival complement and explative *there* can appear as their subject or object, which distinguishes them from control verbs. Control verbs, such as *try*, *want* or *refuse*, co-reference the subject of their infinitival complement with one of their arguments. What follows are example sentences with raising and control verbs respectively.

*Jack **seems** to enjoy the weather.*
*Jack **wanted** to buy a car.*

In order to tackle non-local dependencies introduced by raising and control, as well as by other linguistic constructions, arguments of complex arguments of a category mediating the non-local dependence are assigned indices to establish co-reference. As an example, let us consider the verb *seem* in the first of the two sentences listed above. The CCG category of the verb annotated with co-indexation information is $(S[dcl] \setminus NP_i)/(S[to] \setminus NP_i)$, where subscript i denotes that the two arguments should be the same. As *Jack* is a subject of the raising verb *seems*, and given the co-indexation information, *Jack* is also used as a subject of the verb *enjoy*. The λ-ASP expression for a verb *seem*, which embeds the co-indexation information is given by: $[e0]\#X1.\#X0.binaryEvent(e0, seem, X0, X1), (X1)@(X0)$. Co-indexed λ-ASP expressions are provided by the verb sub-lexicon (section 3.2.2). For an example derivation of a semantic representation of a sentence containing a control verb please see Example 3.4 and the verb *decide*. Example 3.8 provides a sentence with a relative clause.

Example 3.8 (Co-indexation of relative clauses)**.** Let us consider the sentence *Jack has a car that is expensive*, which has the CCG parse tree given in Listing 3.4.

Listing 3.4 CCG parse tree of a sentence *Jack has a car that is expensive.*

```
1: <T S[dcl]>
2:   <L NP NNP Jack>
3:   <T (S[dcl]\NP)>
4:     <L ((S[dcl]\NP)/NP) VBZ has>
5:     <T NP>
```

```
 6:        <L (NP/N) DT a>
 7:        <T N>
 8:          <L N NN car>
 9:          <T (N\N)>
10:            <L ((N\N)/(S[dcl]\NP)) WDT that>
11:            <T (S[dcl]\NP)>
12:              <L ((S[dcl]\NP)/(S[adj]\NP)) VBZ is>
13:              <L (S[adj]\NP) JJ expensive>
```

Co-indexed category of a relative pronoun *that* (line: 10) is: $(N_i \setminus N_i)/(S[dcl] \setminus NP_i)$ and the corresponding λ-ASP expression is given by: $\#X1.\#[X0].(X1)@(X0)$. The semantic representation of a copula *is* (line: 12) is given by: $\#[X0].(X0)$ and λ-ASP expression for adjective phrase *expensive* is: $[expensive]\#X0.unaryModif(expensive, X0), (X0)$. Therefore, the semantic representation of the bottom part of the parse tree (lines: 10 - 13) is:

$$\#[X0].(\#X0'.unaryModif(expensive, X0'), (X0'))@(X0)$$

which, when combined with the semantics of a common noun *car* (line: 8) gives:

$$\begin{aligned}&(\#X0'.unaryModif(expensive, X0'), (X0'))@(\#[X1'].unaryNominal(X1', car), (X1'))\\ &\quad\equiv\ \#X1'.unaryModif(expensive, X1'), unaryNominal(X1', car), (X1')\end{aligned}$$

which represents the semantics of a bare noun phrase *car that is expensive*. The derivation of the sentence semantics can be completed in analogous way as in the Example 3.5, and the final λ-ASP expression is:

$$\begin{aligned}&[e0]binaryEvent(e0, have, c0, n0), unaryModif(expensive, n0),\\ &\quad unaryNominal(n0, car), unaryNominal(c0, jack)\end{aligned}$$

Raising, control and relativisation are three instances from a list of linguistic constructions that require co-indexation, for a more complete presentation please refer to [25]. It is worth mentioning that in the project co-indexation for all constructs listed in [25] was implemented, which shows expressiveness and wide applicability of λ-ASP expressions.

3.3.5 Processing Questions

In the project we focused on polar questions and a subset of *wh*-questions, namely the ones starting with *wh*-words *what*, *where*, and *who* and which can be answered using a single-word nominal. Two separate approaches to handling the two types of questions were developed. Polar questions are transformed into declarative sentences and their truth value is evaluated in the context of the associated text. For the specified subset of *wh*-questions, an *answer predicate* is introduced - one of its arguments gives an answer to the question.

In order to convert polar question to declarative sentences, subject-auxiliary inversion had to be "reversed", which was achieved by analysing the CCG parse tree of the question, identifying the subject of the question and the corresponding set of tokens and switching the positions of the subject and the auxiliary verb in the question. The set of tokens for a subject of a question is derived by flattening a sub-tree of a CCG parse tree corresponding to the subject. The most compelling benefit of converting polar questions to declarative sentences is that co-reference resolution works better for the latter.

Example 3.9. Let us consider a polar question: *Is the runner who won the marathon famous?*, which has the CCG parse tree given in Listing 3.5.

Listing 3.5 CCG parse tree for a question: *Is the runner who won the marathon famous?*

```
 1: <T S[q]>
 2:   <T (S[q]/(S[adj]\NP))>
 3:     <L ((S[q]/(S[adj]\NP))/NP) VBZ Is>
 4:     <T NP>
 5:       <L (NP/N) DT the>
 6:       <T N>
 7:         <L N NN runner>
 8:         <T (N\N)>
 9:           <L ((N\N)/(S[dcl]\NP)) WP who>
10:           <T (S[dcl]\NP)>
11:             <L ((S[dcl]\NP)/NP) VBD won>
12:             <T NP>
13:               <L (NP/N) DT the>
14:               <L N NN marathon>
15:   <L (S[adj]\NP) JJ famous>
```

Nodes of the CCG parse tree in lines: 4-14 correspond to the subject of the question. The set of subject's tokens can be obtained by flattening the sub-tree whose root is in line 4, which gives a noun phrase: *[the runner who won the marathon]*. Finally, positions of auxiliary *is* and the subject are switched, which gives a declarative sentence: *[The runner who won the marathon] [is] famous.*

The selected subset of *wh*-questions was intentionally constrained so that the questions can be answered using a single word, which is a nominal. Therefore, the ordinary translation of the question is parametrised by an answer variable and an answer predicate is added to the set of generated λ-ASP expressions, which is achieved by using the following λ-ASP expression for *wh*-words introducing questions:

$$\#X.unaryNominal(C, ANS), X@(C) \tag{3.11}$$

where C corresponds to identifier of the answer and ANS corresponds to the lemma.

Example 3.10. Let us consider a question: *Who broke the window?*, which has a CCG parse tree given in Listing 3.6.

Listing 3.6 CCG parse tree for a question: *Who broke the window?*

```
1: <T S[wq]>
2:   <L (S[wq]/(S[dcl]\NP)) WP Who>
3:   <T (S[dcl]\NP)>
4:     <L ((S[dcl]\NP)/NP) VBD broke>
5:     <T NP>
6:       <L (NP/N) DT the>
7:       <L N NN window>
```

λ-ASP expression for the sub-tree rooted at the node `<T (S[dcl]\NP)>` (line: 3) is:

$$[E0]\#X0.binaryEvent(E0, break, c1, X0), unaryNominal(c1, window), (X0)$$

As semantic representation for node `<L (S[wq]/(S[dcl]\NP)) WP Who>` is given by the expression 3.11, λ-ASP expression for the entire question is given by:

$$
\begin{aligned}
&(\#X.unaryNominal(C, ANS), X@(C))@(\\
&\quad \#X0.binaryEvent(E0, break, c1, X0), unaryNominal(c1, window), (X0)\\
&\equiv \; unaryNominal(C, ANS), (\\
&\quad \#X0.binaryEvent(E0, break, c1, X0), unaryNominal(c1, window), (X0))@(C)\\
&\equiv \; unaryNominal(C, ANS), binaryEvent(E0, break, c1, C), unaryNominal(c1, window)
\end{aligned}
$$

Let us notice that in the example 3.10, identifier of *binaryEvent* predicate is a variable $E0$. When semantic representation for questions is generated, identifiers of all predicates apart from noun phrases with co-referents in the text and present tense verbs are replaced with variables.

3.4 λ-ASP Calculus to ASP Translation

Due to the syntactic similarity between λ-ASP expressions and ASP, conversion from the former to the latter essentially requires keeping only the *predicate* primitives of the λ-ASP expression which, when the list of abstractions of the corresponding λ-ASP expression is empty, get translated to a set of ASP facts. A complication occurs when after generating semantic representation of the entire sentence, the set of abstractions of the λ-ASP expression is non-empty. Such λ-ASP expressions are converted to ASP rules and information about the head of the λ-ASP expression is used to select a head of the rule. The remaining predicates constitute rule's body. Moreover, a Skolemised version of the rule, which is a set of facts, is generated by applying to the λ-ASP expression number of Skolem constants equal to the number of abstractions in semantic representation of a sentence.

Example 3.11 (Translation of a sentence to ASP rule)**.** Let us consider a sentence: *Apples are healthy.* The corresponding λ-ASP expression is:

$$[healthy]\#X0.unaryModif(healthy, X0), unaryNominal(X0, apple), (X0)$$

Given that applications are discarded when converting λ-ASP expressions to ASP and that the head of the above expression is constant *healthy*, the corresponding ASP rule is give by:

$$unaryModif(healthy, X0) :- unaryNominal(X0, apple)$$

and the set of ASP facts corresponding to Skolemisation is:

$$\{unaryModif(healthy, sk0), unaryNominal(sk0, apple)\}$$

Let us notice that the Skolemisation allows answering questions such as *What is healthy?*

One issue that has not been discussed so far is ordering of arguments within a predicate, which is resolved when λ-ASP expression is converted to ASP. A predicate identifier, if present, is the first argument of a predicate and lemma of the corresponding word is the second (if no identifier is present, lemma is the first argument). The remaining predicate

arguments, which are provided by the arguments of the corresponding word, are ordered according to indices of their semantic roles as returned by a semantic role labeller. In case of missing semantic roles for more than one argument, the ordering of arguments is kept the same as in the corresponding λ-ASP expression. By using this approach, the same ordering of arguments in a predicate is derived even in presence of inversion or passive movement.

Example 3.12 (Predicate ordering)**.** Let us consider the sentence: *The kitchen was cleaned by Jack.* Semantic role tags of noun phrases *Jack* and *the kitchen* with respect to the transitive verb *clean* are: $ARG0$ and $ARG1$. Therefore, the following ASP fact corresponds to the verb *clean*: $binaryEvent(e0, clean, c0, n0)$, where constants $c0$ and $n0$ stand for *Jack* and *kitchen* respectively.

Chapter 4

Learning

The system developed in the project relies on Inductive Logic Programming, more precisely Inductive Learning of Answer Set Programs (ILASP) algorithm [31] [32], to learn hypotheses that would allow the system to answer question requiring common sense knowledge. For description of the ILASP algorithm please refer to Section 2.3. The main challenge associated to learning which had to be solved during the course of the project is automatic generation of learning tasks from textual input provided to the system, which is described in detail in this chapter.

First, the format of the training examples is presented together with a mechanism of converting the textual input to ILASP examples. Then, automatic approach to generation of mode declarations is described in detail. Finally, generation of mode bias constraints and heuristics related to creation and scheduling of learning tasks are outlined.

4.1 Example Format

In the learning setting, input to the system consists of a list of examples, each of which follows one of two formats. In both formats, list of sentences forming the text is provided. In the first format, the text is followed by a question associated with correct and incorrect answers. In the second format, statements entailed and not entailed by the text are specified.

Example 4.1 (Learning input formats)**.** The same learning example specified using the two different input formats.

Text: Jack gave Jim the box. Jim handed Jack the ball. **Question:** Who received the ball? **Correct:** Jack **Incorrect:** Jim	**Text:** Jack gave Jim the box. Jim handed Jack the ball. **Correct:** Jack received the ball. **Incorrect:** Jim received the ball.

Textual input specified in the above manner allows for direct conversion to a learning task suitable for use with ILASP. In case of learning examples specified as question-answer pairs, question together with answer is translated to a set of ASP rules and, depending on whether the answer is correct or incorrect, the predicates are added to inclusions or

exclusions respectively. Analogously, in case of the second format, the correct and incorrect statements serve as inclusions and exclusions respectively.

Listing 4.1 ILASP example corresponding to the textual inputs presented in Example 4.1.

```
% Background knowledge rules:
semBinaryEvent(E,L,Y,Z) :- binaryEvent(E,L,Y,Z),
        not abBinaryEvent(E,L,Y,Z).

#pos(p0, {
  rule(r0)
}, {
  rule(r1)
}, {
  ternaryEvent(e0,give,c1,c3,c2).
  unaryNominal(c1,jack).
  unaryNomnial(c3,box).
  unaryNominal(c2,jim).
  ternaryEvent(e1,hand,c2,c6,c1).
  unaryNominal(c6,ball).
  metaData(0,e0).
  metaData(1,e1).
  rule(r0):-semBinaryEventH(E0,receive,c1,c6).
  rule(r1):-semBinaryEventH(E1,receive,c2,c6).
}).
```

When background knowledge is specified, the same set of background knowledge rules is used by all examples and hence they are added outside the example context, which can be observed in Listing 4.1.

4.2 Context Generation and Background Knowledge Inclusion

In the learning task an implicit assumption is made that multiple examples were provided to the system in order to learn a single underlying concept. Due to the fact that the text part of each example, which forms example context, could translate to multiple ASP rules, examples are translated to context dependent partial interpretations [32], as shown in Listing 4.1. ASP rules forming the example context are derived using the translation procedure described in Chapter 3.

4.2.1 Generation of Inclusions and Exclusions

Inclusions and exclusion require special treatment as far as translation is concerned. Please recall from Chapter 3 that majority of predicates has an *identifier* which allows for semantic composition. In case of sentences forming inclusions and exclusions, such constants require separate treatment, which depends on the function on the corresponding predicate.

In most cases, the head constants are replaced with the corresponding variables. Currently, the only exceptions are present tense verbs and nominals whose coreferents are present in the text. Example of such replacement can be seen in Listing 4.1 for rules headed by predicates `rule(r0)` and `rule(r1)`. Let us notice that if constants were used instead of variables `E0` and `E1`, the inclusion could be covered as such constants would not occur as an argument of any predicate present in the context.

In case of the nominals whose coreferents are present in the context, like for example *the ball* in Example 4.1, introduction of a variable is not necessary. The same is true for present tense verbs due to the assumption about the linearity of time in the narratives. Present tense verbs occurring in the inclusions and exclusions are assumed to relate to the state of the world *after* the events from the narrative, therefore it is implicitly assumed that they do not have corresponding facts in the context and that they should be entailed by the context and the background knowledge.

Let us notice that the names of predicates for all facts and bodies of all rules present in the inclusions and exclusions have `H` suffix. This modification was introduced in order to prevent the increase in the size of the grounding in presence of recursive rules. However, the modification prevents our system from learning recursive concepts.

4.2.2 Background Knowledge Inclusion

In order to increase the capability of the learner, it is possible to include background knowledge in the learning task. The same set of background rules is used for all examples within one task. Due to the fact that predicates in inclusions and exclusions have `H` suffix, it was necessary to make the same modification to the background knowledge rules.

In order to be able to use the modified (`H` suffix) background knowledge in combination with rules included in the example context, a set of additional background knowledge rules was generated, in which for every predicate $p(a_0, \ldots, a_n)$ from the example context there is a corresponding rule of the form:

$$p_h(a_0, \ldots a_n) \ :- \ p(a_0, \ldots, a_n)$$

where $p_h(a_0, \ldots a_n)$ is the corresponding predicate with `H` suffix. Let us point out that the predicates in the heads of the learnt rules also have the `H` suffix, which is removed by a post-processing step before the learnt hypothesis is returned by the system.

4.3 Automatic Generation of Mode Declarations

As outlined in Section 2.3, in ILASP mode declarations are used to bound the hypothesis space. Automatic generation of head and body modes based on the background knowledge and context of all examples included in the task was the most challenging part of automating the learning process. The chosen approach was inspired by the method presented in [37], namely using distributional word semantics, i.e. relying on contextual information, to derive types of entities and predicates occurring in the text.

The overall objective of deriving mode declarations is to generate a set of modes which are expressive enough to define a hypothesis space containing a solution to the learning problem and conservative enough not to generate a prohibitively large hypothesis space rendering the learning task computationally infeasible.

Derivation of mode declarations is divided into four stages:

- unification of inclusions and exclusions with background knowledge

- assignment of types to predicate arguments
- assignment of types to predicates

At the end of the process, a set of typed head and body mode declarations is obtained together with an assignment of predicate arguments to types and sets of values that each predicate argument can take.

4.3.1 Unification of Inclusions and Exclusions with Context and Background Knowledge Rules

The main aim of unifying the inclusions and exclusions with the background knowledge is to derive *instances* (groundings) of head modes that would in turn allow generation of head modes that lead to derivation of the correct hypothesis when the background knowledge is included in the task. The unification algorithm uses the *subsumption* relation to compute, in a top-down fashion, a set of facts which are the aforementioned instantiated head modes.

Definition 4.1 (Subsumption Relation)**.** Let $P(a_1, a_2, \ldots, a_n)$ and $Q(b_1, b_2, \ldots, b_n)$ be n-ary predicates. Let V denote the set of all variables and C the set of all constants. Then, $P(a_1, a_2, \ldots, a_n)$ subsumes $Q(b_1, b_2, \ldots, b_n)$ if and only if the following conditions are met:

- $P = Q$
- $\exists_{\phi \in V \times C} \ \forall_{1 \leq i \leq n} \ \phi(a_i) = b_i$

where ϕ is a substitution - a partial function from V to C.

Given the above definition, the set of predicates which will be considered as heads of hypotheses is computed recursively by unifying candidate heads, which are initialised with inclusions and exclusions, with rules in the background knowledge and extending the candidate set with the bodies of the unified rules.

More precisely, for each example in turn, the *heads* predicate set is initialised with the inclusions and exclusions of the given example. For each predicate in the set and for each rule in the background knowledge and the context it is checked whether the head of the rule subsumes the candidate predicate. For all candidates subsumed by a head of some rule, predicates from the body of the rule, on which substitution is performed, are added to the *heads* set. The algorithm returns the set of *relevant* heads - predicates visited by the procedure whose semantics is not fixed. Pseudocode of the algorithm in presented in Listing 4.2.

Listing 4.2 Pseudocode of algorithm generating candidate head predicates for positive example e and background knowledge rules B.

```
1: procedure unify(e, B):
2:   result := ∅
3:
4:   for ex ∈ E_e^inc ∪ E_e^exc:
5:     visited := {ex}
6:     heads := {ex}
7:     relevant := ∅
8:
```

```
 9:     while head ≠ ∅:
10:       curr := heads.pop()
11:
12:       for r ∈ B_clause ∪ C_clause:
13:         if head(r) subsume curr:
14:           φ := substitution(head(r), curr)
15:
16:           if (¬ fixed_semantics(p_s)):
17:             relevant = relevant ∪ {p_s}
18:
19:           for p ∈ body(r):
20:             # Skolemisation is performed only for v ∉ head(r)
21:             p_s := skolemise(φ(p))
22:
23:             if (p_s ∉ visited):
24:               visited := visited ∪ {p_s}
25:               heads := heads ∪ {p_s}
26:
27:     result := result ∪ relevant
28:   return result
```

The algorithm presented in Listing 4.2 performs a form of abductive inference, given the observations, which in our case are inclusions and exclusions, it derives *possible* explanations using the context and background knowledge rules. The algorithm was inspired by the SLDNF proof procedure as applied in case on abduction - when a subgoal fails to unify with the head of any rule, the subgoal is viewed as a hypothesis [28], which in our case translates to an instance of a head mode. Let us notice an important distinction, hypotheses leading to the one that is non-unifiable are also treated as instances of head modes, unless they are fixed semantics predicates.

Let us notice the use of `fixed_semantics` predicate on line 23 in 4.2, which returns a true value for predicates with predetermined semantics - ones for which no new rules should be derived. Currently, among such predicates are *semantic* equivalents of the predicates output by the translation algorithm when parsing questions. These predicates have a name prefix `sem` and are used to represent exceptions.

Example 4.2 (*Semantic* Predicate)**.** For binary event predicates, the corresponding *semantic* predicate is defined by the following rule.

$$semBinaryEvent(E, L, X, Y) :- binaryEvent(E, L, X, Y), \\ not\ abBinaryEvent(E, L, X, Y).$$

Such a predicate, when used for a binary verb *be*, can convey the meaning that X *is* Y unless *not* modifies *is* by inclusion of the following rule:

$$abBinaryEvent(E, be, X, Y) :- binaryEvent(E, be, X, Y), unaryModif(not, E)$$

As can be sees, *semantic* predicates were introduced to convey the *true* meaning of the underlying text.

Let us notice that the algorithm in Listing 4.2 analyses both inclusions and exclusions. The motivation for doing so is the fact that for $e \in E^{exc}$ there might exist a rule $r \in B_{rule} \cup C_{rule}$

(where B_{rule} and C_{rule} are sets of rules occurring in the background knowledge and context of a given example) such that there exists a negated atom $not\ \ p \in body(r)$. In such case, adding p to the set of candidates from which the head modes are derived might prevent e from being entailed.

Example 4.3. Let us assume that a learning task consists of a single positive example e, with inclusions E^{inc}, exclusions E^{exc} and context C given by:

$$\begin{aligned} E^{inc} &= \{semBinaryEvent(e3, close, c1, c2)\} \\ E^{exc} &= \{semBinaryEvent(e3, close, c0, c2)\} \\ C &= \{binaryEvent(e0, forget, c0, e1). \\ &\quad binaryEvent(e1, close, c0, c2). \\ &\quad binaryEvent(e2, close, c1, c2). \\ &\quad unaryNominal(c0, jack). \\ &\quad unaryNominal(c1, jim). \\ &\quad unaryNominal(c2, door).\} \end{aligned}$$

The background knowledge B is given by:

$$\begin{aligned} B = \{semBinaryEvent(E, L, Y, Z) :- &binaryEvent(E, L, Y, Z), \\ &not\ \ abBniaryEvent(E, L, Y, Z).\} \end{aligned}$$

Unification, using algorithm in Listing 4.2, of atom $semBinaryEvent(E0, close, c1, c2)$ with the background knowledge B yields the following set of ground head modes:

$$C_1 = \{binaryEvent(E0, close, c0, c2), abBinaryEvent(E0, close, c0, c2)\}$$

Similarly, for $semBinaryEvent(E1, close, c0, c2)$:

$$C_2 = \{binaryEvent(E1, close, c1, c2), abBinaryEvent(E1, close, c1, c2)\}$$

Let us notice that in Example 4.3 *semantic* predicates $semBinaryEvent(E0, close, c0, c2)$ and $semBinaryEvent(E1, close, c0, c2)$ were not included in C_1 and C_2 as they are assumed to have fixed semantics.

4.3.2 Argument Typing

Our argument typing algorithm processes every example and analyses predicates which form rules, facts and constraints included in the example context, inclusions, exclusions as well as the predicates obtained by running the algorithm from Listing 4.2 which do not include Skolem constants as arguments. The general idea is to memorise for every constant or variable c triples $(predicate\_name, lemma, position)$[1] for every predicate in which c occurs as an argument. Such triples are referred to as *descriptors* and arguments with the same descriptor are assigned to the same type. The intuition behind such typing algorithm follows from the heuristic assumption that words with similar semantics occur in similar contexts.

[1] *lemma* refers to the argument of the predicate, as described in Section 3.1, and *position* denotes the index of the given argument in the predeicate.

The idea that *words which are similar in meaning occur in similar contexts*, also referred to as the Distributional Hypothesis, is a basics for statistical approaches to word semantics [55].

Due to the chosen representation, every predicate occurring in context rules and inclusions or exclusions takes a lemma of the corresponding word as one of its arguments. The same holds for a subset of predicates derived form background knowledge using procedure outlined in Listing 4.2. Consequently, for arguments of these predicates, descriptor of the aforementioned form $(predicate\_name, lexeme, position)$ can be constructed. The steps of our algorithm are as follows:

- For the i^{th} example collect a set P_i of predicates occurring in context rules, inclusions, exclusions and the ones following from background knowledge by abducing inclusions and exclusions (running algorithm presented in Listing 4.2).
- For every predicate $p(a_1, \ldots, a_n) \in P_i$ and for every argument a_j of such predicate create a descriptor (p, l, j), where p is the predicate name, l is the lemma associated with the predicate and j is the position (index) of the argument.
- Let A_i be a set of all arguments (constants and variables) that predicates in P_i can take. Let $A = \bigcup_{i=1}^{M} A_i$ where M is the total number of examples, i.e. A is the set of all predicate arguments in all examples (let us assume that different constant names are used across the examples and different variable names across rules). For every pair of arguments $a_k, a_l \in A$ let D_k, D_l be their sets of descriptors computed in the previous step. Then, a_k, a_l are assigned to the same type t_m iff $D_k \cap D_l \neq \emptyset$.

The last step of the algorithm is optimised by considering only *supersets* of descriptors. A descriptor superset contains all descriptors of arguments of the given type. In case a new argument is determined to belong to an existing type, the corresponding superset is extended with the argument's descriptors. Such optimisation allows to perform $\mathcal{O}(T)$ descriptor set intersections per argument, rather than $\mathcal{O}(N)$, where T is the total number of argument types and N is the total number of different predicate arguments in all examples.

Example 4.4. Let us consider a simple narrative:

Jack went to the garden.
He got a football.
Jim went to the kitchen and got a sandwich.

Let us assume that this narrative constitutes a context of an example. In ASP, it is represented by the following set of predicates (meta predicates were omitted for the sake of clarity as they are irrelevant for argument clustering). The derived argument types are presented in Table 4.1.

$$
\begin{aligned}
C = \{&binaryEvent(e0, go, c0, c1).\ binaryEvent(e1, get, c0, n0).\\
&binaryEvent(e3, get, c2, n1).\ binaryEvent(e2, go, c2, c3).\\
&unaryNominal(c0, jack).\ unaryNominal(c1, garden).\\
&unaryNominal(n0, football).\ unaryNominal(c0, he).\\
&unaryNominal(n1, sandwich).\ unaryNominal(c2, jim).\\
&unaryNominal(c3, kitchen).\}
\end{aligned}
$$

Argument	Descriptors	Type
$c0$	$\{(binaryEvent, go, 0), (binaryEvent, get, 0)\}$	t_0
$c1$	$\{(binaryEvent, go, 1)\}$	t_1
$c2$	$\{(binaryEvent, go, 0), (binaryEvent, get, 0)\}$	t_0
$c3$	$\{(binaryEvent, go, 1)\}$	t_1
$n0$	$\{(binaryEvent, get, 1)\}$	t_2
$n1$	$\{(binaryEvent, get, 1)\}$	t_2

Table 4.1: Descriptor values and types assumed by arguments of predicates in set C. Three different types are recognised by the argument typing algorithm. Type t_0 approximately corresponds to a *person*, t_1 to *location* and t_2 to an *object*.

4.3.3 Predicate Typing

The main aim of predicate typing is to group predicates based on their meaning in the context of the narrative and hence be able to treat them uniformly when generating the hypothesis space - use a single mode declaration for all predicates grouped together. By assigning types to the relations between object in the domain of discourse we try to extract the underlying word semantics by removing syntactic variability introduced by phenomena such as synonymity. Predicate typing relies on the argument types and predicate naming convention. In general, predicates with the same name and argument types are assigned the same predicate type.

Let us recall from Chapter 3 that every predicate occurring in the context, inclusions and exclusions, as well as some predicates derived from the background knowledge by abducing inclusions and exclusions, takes *lemma* as one of its arguments. Therefore, predicate typing is equivalent to assigning types to words taking multiple arguments, such as transitive verbs.

Example 4.5. Let us consider the following narrative:

Jack went to the garden and grabbed a football.
He moved to the kitchen and left the football
Jim went to the kitchen and picked up the football.

The narrative is represented by the following set of ASP facts used as example context:

$$\begin{aligned} C = \{&binaryEvent(e0, go, c0, c1).\ binaryEvent(e1, grab, c0, c2).\\ &binaryEvent(e2, move, c0, c5).\ binaryEvent(e3, leave, c0, c2).\\ &binaryEvent(e4, go, c3, c4).\ binaryEvent(e5, pick\_up, c3, c2).\\ &unaryNominal(c0, jack).\ unaryNominal(c0, he).\\ &unaryNominal(c1, garden).\ unaryNominal(c2, football).\\ &unaryNominal(c3, jim).\ unaryNominal(c4, kitchen).\} \end{aligned}$$

Using the argument typing algorithm described in section 4.3.2, $\{c0, c3\}$ are assigned type t_0, $\{c1, c4\}$ are assigned type t_1 and $\{c2\}$ is assigned t_2. Based on such type assignment, predicates with non-zero arity occurring in the context (*binaryEvent* predicates) can be typed, result of which is shown in Table 4.2.

Predicate	Signature	Type
$binaryEvent(e0, go, c0, c1)$	(t_0, t_1)	p_0
$binaryEvent(e1, grab, c0, c2)$	(t_0, t_2)	p_1
$binaryEvent(e2, move, c0, c5)$	(t_0, t_1)	p_0
$binaryEvent(e3, leave, c0, c2)$	(t_0, t_2)	p_1
$binaryEvent(e4, go, c3, c4)$	(t_0, t_1)	p_0
$binaryEvent(e5, pick\_up, c3, c2)$	(t_0, t_2)	p_1

Table 4.2: Types assigned to predicates based on the types of their argument. *Signature* is a tuple of types that the arguments of a predicate take. *Type* is the final type of the predicate. To avoid confusion, predicate types are denoted with a letter p and a subscript.

One complication regarding predicate typing occurs for predicates derived from the background knowledge which take Skolem constants as their arguments. This situation occurs for all background knowledge rules which have variables in their body that do not occur in their heads, an example of such rule is:

$$unaryNominal(C1, L2) :- eq(C1, C2), unaryNominal(C2, L2).$$

which states that if two entities are equal, then they belong to the same class of entities.[2] The variable $C2$ does not occur in the head, hence when, for example, observation $unaryNominal(c4, human)$ is abduced, the result will be:

$$\{eq(c4, sk1), unaryNominal(sk1, human)\}$$

where $sk1$ is an introduced Skolem constant, whose type is unknown. In such case, a separate new type is introduced for the Skolem constant. However, in some cases it can be inferred that the type of such Skolem needs to be the same as some different argument type based on the types of predicates in which the Skolem constant occurs. That is the case for the background knowledge rule:

$$binaryTerm(E, be, P, L) :- binaryEvent(E, go, P, L2), unaryNominal(L, C).$$

which roughly expresses the fact that *an entity terminates to be at a certain location when it moves to another location.* In such situation, when, for example $\{binaryTerm(e1, be, c1, c2)\}$ is abduced, the resulting explanations are:

$$\{binaryEvent(e1, go, c1, sk1), unaryNominal(c2, sk2)\}$$

where $\{sk1, sk2\}$ are Skolem constants. Consequently, knowing the types of arguments that *binaryEvent* predicates with lexeme *go* take in the example contexts, which can be determined using algorithm outlined in section 4.3.2, allows the type of $sk1$ to be inferred.

In order to systematise making the aforementioned inferences, the problem of typing Skolem constants is translated to a problem of finding strongly connected components in an undirected graph. The set of types of all arguments in all predicates maps to vertices of a graph. By default, every Skolem constant introduced during abduction is assigned a new type.

[2] For example, it allows us to conclude that if an object corresponding to the constant $c1$ is a human, and $c2$ is equal to $c1$, then $c2$ is a human as well. Such inferences are necessary when attempting to learn coreference rules.

Whenever a Skolem constant occurs as an argument of some predicate, undirected edge between the original type of the Skolem constant and the type of the predicate argument is added. Such edges correspond to *equality constraints* between types. After processing all examples, strongly connected components are found in the graph and all types within one component are unified into a single type.

4.3.4 Using Type Information to Generate Modes

The main objective of argument and predicate typing is generation of head and body mode declarations. Each predicate type translates to a separate mode declaration and argument types allow to specify the arguments of such modes together with constant values that each of such arguments can take.

Body modes are derived from *facts* that constitute the example contexts, whereas head modes are generated by abducing the inclusions and exclusions with respect to the background knowledge rules and *rules* from the context, using the algorithm from Listing 4.2. In both cases, the obtained predicates are assigned types and for each predicate type a corresponding mode is created. For every predicate type, a set of lemmas that predicates of the given type can take is kept. Similarly, values of arguments assuming a certain type are collected and if any of them is a lemma, it gets added to a constant value set for the given argument type.

Example 4.6. Let us revisit the Example 4.3. Three different predicate types can be

Predicate	Signature	Type
$binaryEvent(e0, forget, c0, e1)$	(t_0, t_2)	p_0
$binaryEvent(e1, close, c0, c2)$	(t_0, t_1)	p_1
$binaryEvent(e2, close, c1, c2)$	(t_0, t_1)	p_1
$abBinaryEvent(E1, close, c0, c2)$	(t_0, t_1)	p_2

Table 4.3: Types of predicates occurring in the context of Example 4.3 and abduced using the background knowledge.

distinguished for predicates occurring in the Example 4.3. Predicate of type p_2 was obtained by abducing the exclusion, therefore it contributes a head mode:

$$modeh(abBinaryEvent(var(t2), const(c3), var(t0), var(t1)))$$

The remaining two predicate types yield body mode declarations:

$$modeb(binaryEvent(1, var(t2), const(c4), var(t0), var(t2)))$$
$$modeb(binaryEvent(1, var(t2), const(c5), var(t0), var(t1)))$$

The constant sets are: $const(c3) = \{close\}$, $const(c4) = \{forget\}$, $const(c5) = \{close\}$. Let us notice that types $\{c3, c4, c5\}$ are treated as constants in this example, however they could be variables as well. The former approach was taken in order to illustrate creation of constant sets.

4.4 Mode Bias Constraints

One of the primary concerns related to learning common sense knowledge was scalability of the generated learning tasks. This issue was partially addressed by careful derivation of mode declarations. However, in order to further reduce the size of the hypothesis space and remove hypotheses that, due to the representation used throughout the project, could upfront be deemed incorrect, the mode bias constraints mechanism was utilised.

The notion of inductive learning through constraint-driven bias was introduced in [3] and it allows to constrain the hypothesis space, induced by mode declarations, by domain-specific denials. In ILASP, bias constraints can be expressed as ASP program that uses predicates *head* and *body* to specify constraints on the corresponding parts of the rules in the hypothesis space.

Two types of domain-specific constraints that significantly reduced the size of the generated hypothesis spaces were constraints on the use of nominal and modifiers in the hypothesis body. Essentially, nominals should be included in the hypothesis only when accompanied by predicates whose arguments they are. Examples of such predicates are verbs and modifiers. Similarly in case of modifiers, they are allowed to occur in the hypothesis body only when the predicate corresponding to the part of speech that they modify is included as well. A subset of the bias constraints used for each learning tasks is presented in Listing 4.3.

Listing 4.3 Subset of bias constrains used in the learning tasks.

```
 1: #bias("nominal_allowed(V):-body(binaryEvent(_,_,V,_)).").
 2: #bias("nominal_allowed(V):-body(binaryEvent(_,_,_,V)).").
 3: #bias("nominal_allowed(V):-head(binaryPrepH(_,_,V,_)).").
 4: #bias("nominal_allowed(V):-head(binaryPrepH(_,_,_,V)).").
 5: #bias("nominal_allowed(V):-head(unaryEventH(_,_,V)).").
 6: #bias("nominal_allowed(V):-head(unaryModifH(_,V)).").
 7: #bias("nominal_allowed(V):-head(ternaryModifH(_,V,_,_)).").
 8: #bias("nominal_allowed(V):-head(ternaryModifH(_,_,_,V)).").
 9: #bias(":-body(unaryNominal(V,_)),not nominal_allowed(V).").
10:
11: #bias("modif_allowed(V):-body(unaryNominal(V,_)).").
12: #bias("modif_allowed(V):-body(binaryEvent(V,_,_,_)).").
13: #bias(":-body(unaryModif(_,V)),not modif_allowed(V).").
```

The rules in lines 1 and 2 allow nominals to occur in hypotheses in which binary verbs occur in the body. Similarly, rules in lines 3 and 4 allow nominals in rules in which binary preposition (e.g. *behind*) occurs in the head. The rules in lines 11 and 12 allow unary modifiers, such as *fast*, in bodies of hypotheses in which the modified noun or verb occurs as well. Lines 9 and 13 express the constraints on the hypothesis space. They eliminate hypotheses that have *unaryNominal* or *unaryModif* predicate in their body and for which the use of such predicate is not *validated* by any of the previously specified rules. Let us notice that the same approach, namely specifying the conditions under which a predicate is allowed to occur in a hypothesis body and enforcing them with a constraint, could be used in order to provide mode bias constraints for other predicates.

4.5 Mode Declaration Selection Heuristics

Let us notice that modes declarations derived using heuristics described in Section 4.3 take multiple hyper-parameters that have to be set in order for the learning task to complete successfully. The parametrisation of a set of modes includes:

- The maximum number of variables in each rule,
- The recall of each body mode declaration,
- The specification of which mode arguments should be kept constant and which should be variables.

The three aforementioned sources of variability contribute to the exponential increase in the number of different sets of modes when the number of *blueprint* mode declarations generated by algorithms in Section 4.3 increases. In order to tackle this combinatorial explosion while still being able to explore different parametrisations of modes, a series of heuristics was introduced. In what follows it is assumed that the set of mode declarations is constant and the previously mentioned parameters of modes change.

Number of Variables per Rule The maximum number of variables per rule was estimated by taking the minimum of the number of variable types in all modes and preconfigured upper bound on the number of variables. The upper bound was set to five as rules including more than five variables were rarely observed.

Number of Constant Types Inclusion of a constant type which can assume values from a set C approximately causes the size of the hypothesis space to increase $|C|$ times, which in practice renders the learning task not feasible even when relatively small number (around five) of constant types, which can assume more than one value, is included in the mode declarations. Moreover, mode declarations which have multiple constant types often correspond to enumeration of different cases, which might lead to overfitting. Therefore, there was introduced a hard limit on the number of constant types in each set of mode declarations (equal to three), which has an additional effect of reducing the number of different combinations of modes.

Constant Type Selection As indicated in the Example 4.6, for types that can assume constant (lemma) values, there is a choice of making them variables or constants in the set of modes, which can be closely associated with generality of the learned rules. Different selections of constant types were explored exhaustively within the bound on the maximum number of constant types.

Recall of Body Mode Declarations In order to learn certain rules, like for example transitivity of a given relation, multiple instances of one predicate need to occur in the body of a learned rule. In order include such rules in the hypothesis space, the recall of the corresponding body mode declarations has to be increased. In order to avoid substantial increase in the size of hypothesis space, in a given set of mode declarations only one body mode declaration was allowed to have recall equal to two. Values of recall greater than two were not considered as they are rarely observed in the rules.

Mode Declarations Ordering and Selection The order in which different sets of modes declarations were explored is significant for the overall runtime of the learning pro-

cess. Therefore, sets of modes which generate smaller hypothesis spaces should be explored first. In order to avoid the overhead of enumerating the entire hypothesis space every time usability of a set of mode declarations is estimated, a *weight* of a mode was calculated based on factors like cardinality of sets of constant values corresponding to constant types, number of variable types and recall of body modes. Sets of parametrised modes were sorted in increasing order of their weights. Out of all sets of more declarations, only a certain number of modes with the smallest weight was kept. The heuristic tended to prioritise sets of modes generating smaller hypothesis spaces.

Let us notice that due to the use of the aforementioned heuristics, by principle some types of rules were impossible to learn as discussed in case of individual heuristics. However, in order to improve the runtime and feasibility of the learning process, heuristic assumptions about the form of rules to be learned were required.

Chapter 5

Implementation

In this chapter, the implementation of the ideas presented in Chapters 3 and 4 is described. First, an overview of the functionality and the main implementation choices is provided. Secondly, the design of the system is presented and constituent modules described on a high level. The main contribution of this chapter is an outline of practical consideration related to implementing general-purpose language processing systems.

5.1 System Overview

The objective of the project was to develop a natural language processing system capable of answering questions and learning common sense knowledge from textual examples. This twofold goal guided the design of the system and led to an architecture where two separate modes, namely *learning* and *answering* are supported. These two modes co-exist in the final implementation and share large part of their functionality.

The core part of the project is implemented in Java. The resultant code base comprises of approximately eight thousand lines of code spread across over one hundred classes in fifteen packages. Therefore, for the sake of clarity, in the following sections the implementation will be outlined mostly on a package level.

5.2 Input Translation

In the implementation, a significant amount of functionality was shared between the*learning* and *answering* modes, which is the subject of this section. Common modules are responsible for parsing input English sentences to ASP representations. The exchange of messages between the modules during the translation process is illustrated in Figure 5.1. For a visual presentation of roles that each module plays in generating ILASP learning tasks or ASP programs please refer to Figure 5.3 and Figure 5.4 respectively.

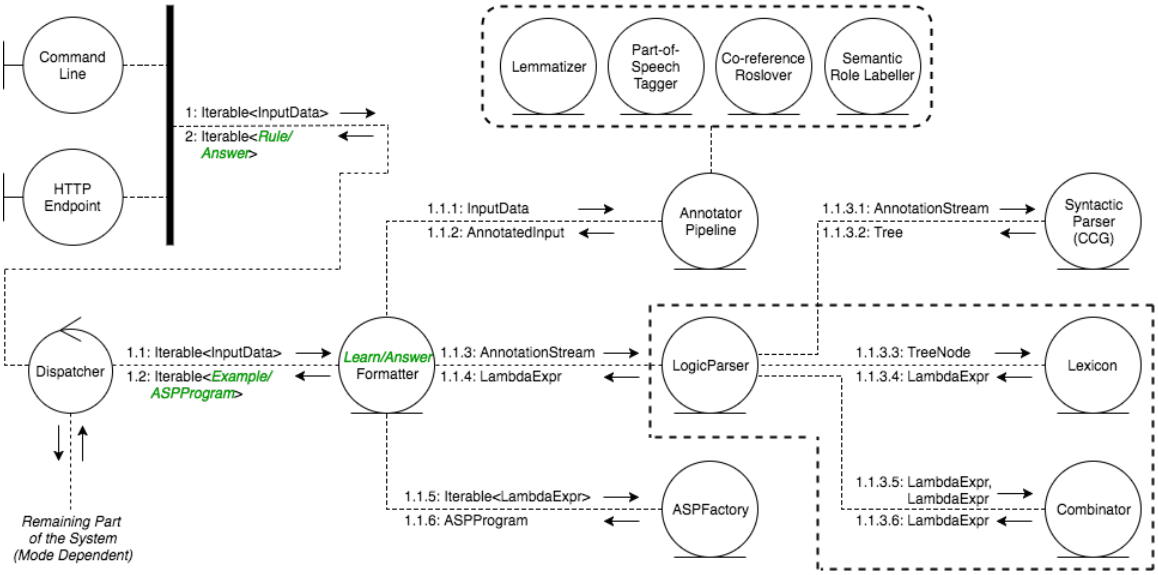


Figure 5.1: UML Communication Diagram of modules used both in *learning* and *answering* modes. In green are highlighted components and messages that are mode-dependent.

5.2.1 User Interface

The system supports two forms of user interface: command line interface, whose intended use case is processing larger amounts of input data, and a graphical interface, implemented in a form of a Web app, which can be used for small learning or question-answering experiments. These two interfaces provide entry points to the system, in the former case the input is read from a file and in the latter it is passed through an HTTP endpoint.

```
[{
    "text": [ "Jack went to the kitchen.",
        "Jim went to the garden." ],
    "questions": ["Where is Jack?"],
    "positive": ["kitchen"],
    "negative": ["garden"]
}]
```

Figure 5.2: The JSON input format used by the system.

The same JSON input format is used in *learning* and *answering* mode, example of which can be seen in Figure 5.2. In case of the *answering* mode, the *positive* and *negative* example fields are redundant and hence left blank. Uniform input format significantly simplifies the implementation of parts of the system responsible for deserialisation and annotation.

The output of the system differs depending on the mode. In the case of the *learning* mode, the system returns a set of learned ASP rules. In the case of the *answering* mode, the system returns a list of answers ordered according to the occurrence of corresponding questions in the input.

5.2.2 Annotator Pipeline

The annotator Pipeline performs morphological, linguistic, and preliminary semantic analysis of the input text for which it relies on two external dependencies, namely Stanford CoreNLP [40] and EasySRL [35]. The following types of annotation are performed:

Sentence Splitting and Tokenisation Although this tasks might appear simple, in practice, in case of deterministic approaches, a significant number of heuristics is required to handle different problematic cases, such as names including punctuation signs or quoted fragments inside sentences. Hence, relying on external libraries to perform these tasks is desirable.

Lemmatisation Removes a part of syntactic variability in the input text by grouping together inflected forms of a word and assigning them word's lemma. This step is important for limiting the number of different predicates generated by the lexicon

Part-of-speech Tagging Assigns to each word a part-of-speech tag as defined in the Penn Treebank Project. This information is used by the lexicon to improve robustness of λ-ASP expression generation.

Named Entity Recognition Identifies sequences of input words which are names and labels them with a relevant class. It is used by the system to detect compound nouns, such as *Barrack Obama*.

Coreference Resolution Determines referents of words occurring in the text. It is used in the system to derive semantic representation of nominals. Let us note that coreference resolution for a particular piece of input data has to be performed on combined text, questions and positive/negative examples, which is why `AnnotatorPipeline` accepts `InputData` object as input rather than separate sentences.

Semantic Role Labelling Assigns roles to arguments of predicates (verbs). EasySLR uses PropBank style role annotations. The only type of annotation used in the project is $ARG[x]$, where x has fixed semantics and, depending on its value, can denote that the argument is *agent*, *recipient*, *theme* or others. The annotations are used to order predicate arguments.

Currently, all annotations apart from semantic role labelling are performed by Stanford CoreNLP. Annotators included in the pipeline have to implement the same interface, which allows uniform treatment and relatively simple replacement or addition of an annotator. Such design has the advantage of reduced coupling between the system and its external dependencies.

5.2.3 Logic Parser

The `LogicParser` module performs translation from English to λ-ASP expressions, theoretical details of which are described in Chapter 3. The module derives translation for sentences represented as `AnnotationStream` objects, which encapsulate the annotation information and text of the original sentence. For each sentence, CCG parser is used to obtain a parse tree, λ-ASP representations of the leaf nodes are derived using the `Lexicon` and representations for the internal nodes of the tree are computed in a bottom-up fashion

using the `Combinator` module. `LogicParser` returns λ-ASP expression corresponding to the entire sentence.

5.2.4 Syntactic Parser

In the project, the implementation of the CCG parser presented in [36] is used. It achieves state-of-the-art performance when evaluated on CCGBank [25]. The parser is implemented in Java and hence could be imported into the project. However, we decided against it for two reasons.

Firstly, running the parser in a server mode greatly improves the development speed of the translation algorithm as repetitive loading of a large language model to memory is not necessary. Let us point out that the original implementation of the parser did not support server mode, hence we had to introduce the necessary modifications to the source code.

Secondly, parsing the textual output produced by the parser, rather than relying on its internal representation of the parse tree, reduces the coupling between our system and the dependency. Let us notice that the parser uses a standard CCGBank output format which is implemented by all CCG parsers, hence by following the chosen approach the parser can be freely replaced.

5.2.5 Lexicon

As described in Chapter 3, in our implementation lexicon is an *algorithm*, which given a leaf node of a parse tree derives a corresponding λ-ASP expression. `Lexicon` module is divided into five submodules, namely: `NominalLexicon`, `VerbLexicon`, `ModifierLexicon`, `PrepositionLexicon` and `WhLexicon`, that produce λ-ASP expressions for different parts-of-speech, as indicated by their names.

In addition to improved robustness (discussed in section 3.2.2), modularising the lexicon allows to achieve better separation of concerns. Certain elements of linguistic analysis have very specific implementation details for some categories of words and are irrelevant for others. For example, detection and handling of co-indexation in case of verbs is performed by pattern matching on a list of CCG categories. However, for nominals co-indexation is not required. Being able to separate such implementation details across classes makes the implementation more maintainable.

The semantic representation of certain words differs depending on whether the word is used in the text, question or example section of the input data (Figure 5.2). An example of such category are interrogative words such as *what*, *who*, *which*. However, as such special cases are rare, in order to avoid code duplication `Lexicon` can be set to operate in one of three modes: text, question or example. Each of the modes introduces necessary adjustments in the λ-ASP representations for the relevant categories of words.

5.2.6 System Configuration

In order to allow tuning our system's performance for different workstations and problem settings, a configuration file is provided which allows specification of parameters responsible

for generating and running learning and question answering tasks. The configuration file provides control over, among others, the number of concurrent learning tasks, number of examples used for learning, maximum allowed size of the hypothesis space and noise tolerance. Location of a configuration file can be specified using an environment variable, which facilitates managing multiple configurations to run the system on different machines.

5.3 Learning Mode

Figure 5.3 illustrates the messages exchanged between different modules of the system when a learning task is generated from textual input. In what follows, the design and implementation of the most important modules used for generating learning tasks is outlined.

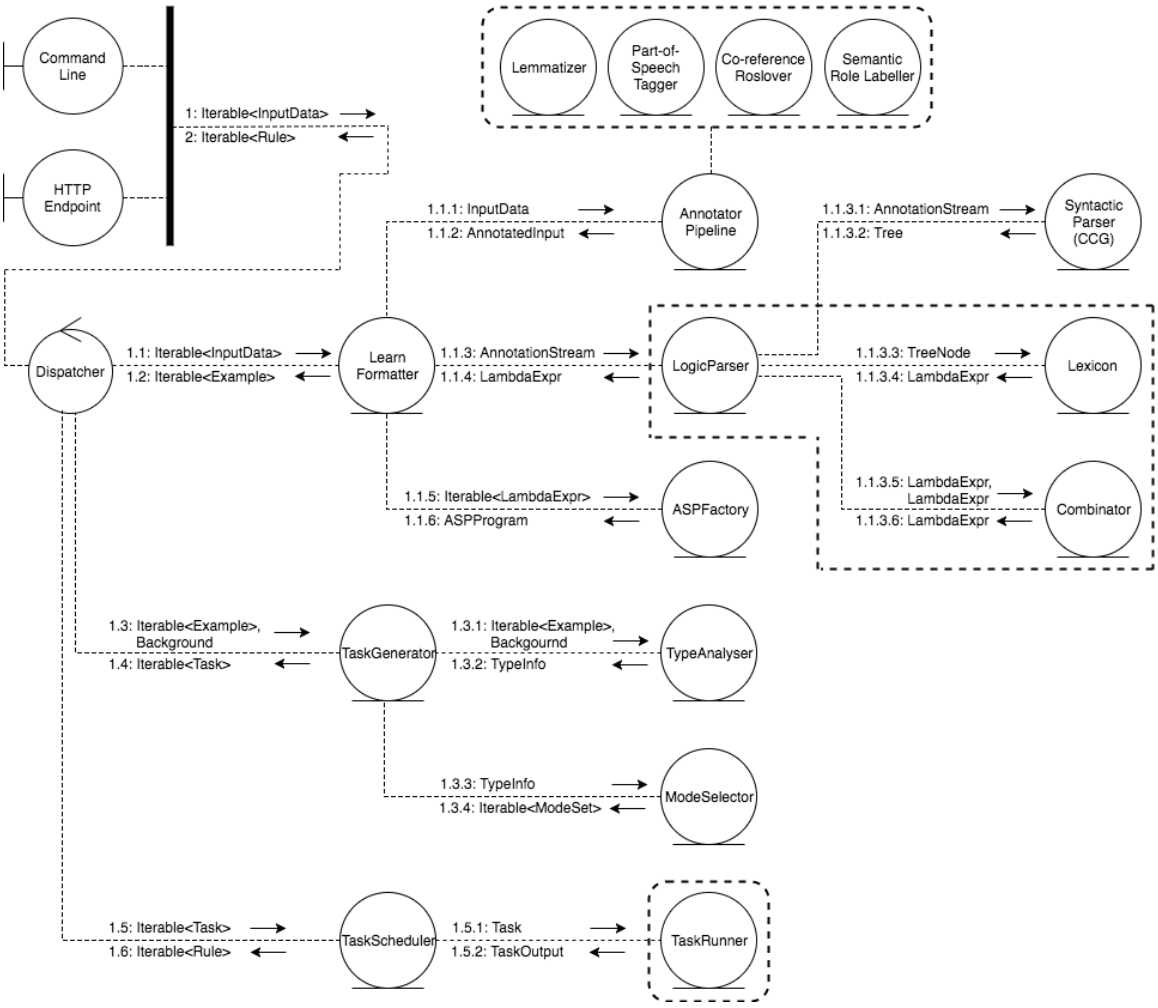


Figure 5.3: UML Communication Diagram of the system used in *learning* mode.

5.3.1 Type Analyser

The `TypeAnalyser` module is responsible for inferring types of predicate arguments and predicates in order to allow generation of mode declarations tailored for a particular learning task. In accordance with the algorithm description provided in section 4.3, implementation was divided into three major submodules: `ArgumentClustering`, `PredicateClustering` and `PredicateTyping`, each providing input to the subsequent submodule it the list. The `TypeAnalyser` module is fully self-contained and could be seamlessly replaced with a different implementation that provides the same interface to the rest of the system. Achieving such effect was one of the design goals as predicate typing is a crucial part of automatic derivation of mode declarations and implementation of this module might need to be changed in order to accommodate more difficult question answering datasets.

5.3.2 Mode Selector

The `ModeSelector` module is responsible for generating mode declarations and also implements the mode declaration selection heuristics outlined in section 4.5. `ModeSet` object, list of which is returned by the module, encapsulates a particular instance of head and body mode declarations for a given learning task. These modes are used in `TaskGenerator` to infer hyper-parameters of the learning task, such as the maximum number of variables used in a rule and hence, together with the examples, form the final learning task.

5.3.3 Task Scheduler

The input to the `TaskScheduler` comprises of a set of learning tasks ordered (by the `ModeSelector` module) according to their relative priority. The main responsibility of the module is scheduling and management of the pool of concurrent learning task.

The `TaskScheduler` relies on concurrency features of Java programming language and uses a `ScheduledExecutorService` to schedule `TaskRunners`. Each task runner takes as input a path to the file containing the learning task and relies on a configuration file specified by the user to create a new process which invokes ILASP with the given learning task as input. If learning succeeds, task runner returns a set of learned rules together with an exit status informing the scheduler that all other currently active task runners could be stopped and task scheduling procedure terminated.

A complication is brought by the fact that, due to completeness of ILASP, running all learning tasks till completion is not always optimal. This is especially true for learning tasks which include noise and have relatively large hypothesis space as ILASP performs an exhaustive search to determine that the task is infeasible. However, empirical observations show that when the correct set of rules is present in the hypothesis space, then for majority of problems attempted throughout the course of the project ILASP would terminate within 10 minutes. Therefore, a timeout value is used, which can be modified via a configuration file, and after which the learning task is forcefully terminated.

5.4 Answering Mode

In *answering* mode, the steps following the translation of input to ASP representation are significantly simpler than in case of the *learning* mode. Essentially, after the list of ASP programs corresponding to input data is generated by the `AnswerFormatter`, the only remaining step is to run multiple concurrent invocations of `clingo` to compute the answers, which are aggregated into a list and returned to the user. Figure 5.4 illustrates interactions between system components in the *answering* mode.

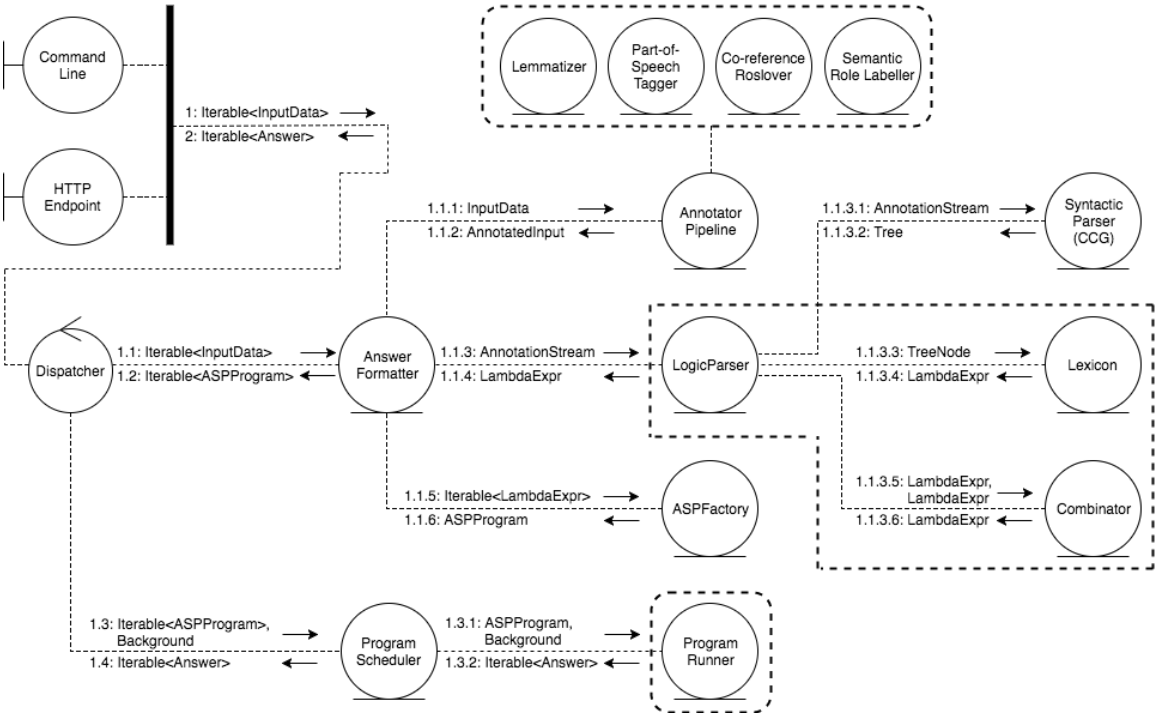


Figure 5.4: UML Communication Diagram of the system used in the *answering* mode.

5.5 External Dependencies

In this section, a list of external dependencies used in the project together with short description and licensing information is provided.

Stanford CoreNLP Provides a set of tools to perform natural language analysis. The framework is implemented in Java and provides a simple web API together with a Java client. The NLP tools provided by the framework achieve nearly state-of-the-art performance on a variety of tasks such as part-of-speech tagging and coreference resolution. Stanford CoreNLP is licensed under the GNU General Public License (v3 or later) and available on GitHub.[1]

EasySRL Provides a semantic role labeller and a state-of-the-art CCG parser [36]. The set of tools is implemented in Java. EasySRL is licensed under Apache License 2.0

[1] https://github.com/stanfordnlp/CoreNLP

and available on GitHub.[2]

ILASP A logic-based learning system developed at Imperial College London by Mark Law, Alessandra Russo and Krysia Broda. Executable is available on SourceForge.[3] Throughout the project it was used with the flag `--2i`.

clingo Answer set solver developed at the University of Potsdam. It is licensed under The MIT License (MIT) and available on GitHub.[4]

Apache Commons Open-source project which provides all kinds of reusable Java components. It is licensed under Apache License 2.0.

Gson Java library used to convert Java object to JSON representation and vice versa. It is licensed under Apache License 2.0 and available on GitHub.[5]

Jetty Lightweight Java based web server and servlet container. Jetty is licensed under Eclipse Public License - v 1.0 and available on GitHub.[6]

[2] https://github.com/uwnlp/EasySRL
[3] https://sourceforge.net/projects/spikeimperial/files/ILASP
[4] https://github.com/potassco/clingo
[5] https://github.com/google/gson
[6] https://github.com/eclipse/jetty.project

Chapter 6

Evaluation

6.1 Translation Evaluation

Despite some previous work on English-to-ASP translation, to the best of our knowledge, no official benchmark has been published to date. Therefore, in order to evaluate the performance of the translation algorithm, a set of 22 sentences of varying length and exemplifying different grammatical constructions was created (Table 6.1). Examples $1-10$ were handcrafted by the author. Examples 11 - 15 were used in [6] where certain aspects of English to ASP translation were addressed. However, as the the publication was mainly concerned with using ASP for solving logical puzzles and the representation that it presented was very task-specific, a direct comparison is not possible. Examples $16-20$ were selected from news articles available on-line [57] [11]. Finally, examples $21-22$ were used to evaluate correctness of the Boxer system [8]. Every example sentence in Table 6.1 is associated with a comment on the correctness of the generated representation. The representations themselves are presented in Appendix C.

Task	**Sentences**	**Comments**
1	*Jack knows a man whose brother won a lottery.*	+ Correct co-indexation of a noun-taking relative pronoun *whose*
2	*Jack ordered and paid for the dinner.*	+ Correct coordination of conjuncts with the same CCG category
3	*Jack is a programmer and happy with his job.*	+ Correct coordination of conjuncts with different CCG categories
4	*Jack bought an expensive car.*	+ Correct representation of a simple sentence with a transitive verb and a simple adjective
5	*Jack bought a car that is expensive.*	+ Correct representation of a sentence with a (non-restrictive) relative clause + The same ASP representation as in Task 4
6	*An expensive car was bought by Jack.*	+ Correct representation of a passive sentence + The same ASP representation as in Tasks 5
7	*Jack is very stubborn.*	+ Correct representation of a sentence containing a compound modifier

8	*Jack insisted on us staying longer.*	+ Correct *lemma* of a predicate corresponding to the phrasal verb *insist on* (*insist_on*)
9	*Jack persuaded Jim to decide in favour of the new agreement.*	+ Correct co-indexation for control verb *persuade* − Incorrect representation of a phrase *in favour*
10	*Does Jack enjoy playing football?*	+ Correct predicate structure + Correct use of a constant for *Jack* and a variable for *football*
11	*Earl arrived immediately before the person with the Rooster.*	− *Arrive* incorrectly takes two nominal arguments − *Before* not included in the representation due to an incorrect CCG parse
12	*Jack did not get a haircut at 1.*	+ Auxiliary *did* correctly skipped + *not* correctly translated as *escnot* (*escape* + *not*) as clingo does not support *not* as a name of a constant
13	*Pete talked about government.*	− *About* incorrectly skipped. Hence, no distinction can be made between *talk to* and *talk about*
14	*The candidate surnamed Waring is more popular than the PanGlobal.*	+ Correct representation of a comparative adjective *more popular* + Correct attachment of *popular* to *Waring*
15	*Miss Hanson is withdrawing more than the customer whose number is 3989.*	− Incorrect predicate structure due to errors in the dependency structure returned by the CCG parser. Parser incorrectly assigned the relative clause *whose ...* as a modifier of *more* and phrase *than the customer* as a modifier of *more*.
16	*Scientists have mostly stopped arguing about whether humans are warming the planet.*	+ Correct predicate structure + Correct attachment of the adverb *mostly* to the verb *stopped*
17	*Yet the arguments that crippled the Kyoto Protocol have hardly changed.*	+ Correct semantic representation of an intransitive verb *change* − *Kyoto Protocol* should be represented as a single entity *kyoto_protocol* rather than distributively as a nominal and a modifier
18	*This time, nations made voluntary commitments, with China agreeing that its emissions will peak in about 2030.*	− *make* takes three arguments and it should take two, which is caused by a parser error - phrase *with China ...* is incorrectly classified as an argument − *its* is not attached to *China*.

19	*The measure was criticized by opponents of the original tax proposal and experts predicted it would be cut.*	+ Generally correct predicate structure − Compound noun phrase *opponents of the original tax proposal* should be analysed distributively as *opponents* and *original tax proposal* with *of* joining the two + *it* correctly coreferenced with *the measure*
20	*Peter Uebelhart, head of tax at KPMG Switzerland, said he did not expect the higher tax on dividends to put off multinationals.*	+ Generally correct predicate structure + *Peter Uebelhart* and *KPMG Switzerland* correctly analysed as compounds − Compound noun phrase *Peter Uebelhart, head of tax at KPMG Switzerland* should be analysed distributively as *Peter Uebelhart*, *head of tax* and *KPMG Switzerland*
21	*Cervical cancer is caused by a virus. That has been known for some time and it has led to a vaccine that seems to prevent it. Researchers have been looking for other cancers that may be caused by viruses.*	+ Correct predicate structure for the first and the third sentence − In the second sentence, *that* is not coreferenced with the verb *cause* in the previous sentence − Incorrect coreference of *it* in the second sentence + Correct co-indexation of a raising verb *seem* in the second sentence − Incorrect representation of a phrasal verb *look for* in the third sentence
22	*John went into a restaurant. There was a table in the corner. The waiter took the order. The atmosphere was warm and friendly. He began to read his book.*	+ Correct predicate structure for all sentences − Incorrect handling of locative *there* in the second sentence + Correct distributional treatment of adjective phrase *warm and friendly* in the fourth sentence − Incorrect coreference of pronoun *he* in the fifth sentence to *waiter* rather than *John* − Possessive pronoun *his* is not attached to the subject *John*

Table 6.1: Results of the correctness analysis of ASP representations generated for the 22 sets of sentences from the prepared dataset. The ASP representations supporting the analysis are presented in Appendix C.

6.1.1 Strengths of the Approach

Below, advantages of the current implementation of English-to-ASP translation, as indicated by the evaluation on examples presented in Table 6.1 are described.

Systematic Translation The main advantage of the approach to English-to-ASP translation outlined in the report is that it offers a systematic way of deriving predicate structure from textual input, which is achieved primarily thanks to using CCG grammar. λ-ASP expressions serve as a *semantic glue* which allows building semantics of more complex statements from their constituents.

Modularity The tree main components of the translation mechanism are: CCG parser, lexicon and annotators (coreference, part-of-speech, lemma) all of which can be freely replaced when a better alternative is available. Let us notice that achieving such outcome required considerable effort at conceptual level and was, among others, achieved by introducing λ-ASP expressions and intermediate representation between syntax and semantics.

Simple Sentences Translation algorithm satisfies the main objective of the project, namely is capable of translating simple sentences (provided correct output of the CCG parser) which is a precondition for achieving high accuracy on any of the bAbI tasks. The algorithm derives correct ASP representation for nouns, which takes into account definiteness, simple adjectives, and verbs with both local and non-local dependency structure, which capitalises on the use of CCG grammar.

Non-local dependencies Regarding non-local dependencies, coordination works correctly for different types of sentences, even in certain cases when categories of CCG conjuncts are different. Similary, the approach successfully produces ASP representation for sentences containing relative clauses. Passive constructions are handled seamlessly thanks to relying on CCG grammar.

Uniformity of Representation Simple sentences with, using transformational grammar nomenclature, different surface structure but the same deep structure are assigned similar or identical ASP representations, which showcases method's ability to extract the underlying meaning of a sentence.

Rules with Exceptions Most of common sense knowledge rules have exceptions. Therefore, being able to specify them on translation level, which was achieved by using *semantic* predicates, greatly enhances capabilities of the system both in question answering and learning setting.

Polar Questions Present tense polar questions are correctly represented in ASP thanks to the preprocessing step converting them to declarative statements, which improves the accuracy of co-reference resolution.

6.1.2 Outstanding Translation Tasks

During the evaluation, certain aspects of the translation algorithm were discovered that are not supported to a sufficient extent and constitute the main obstacle to representing *real*

life text in ASP. Those issues were not identified as weaknesses of the approach *per se*, but rather as issues that were not addressed due to the time constraints of the project.

Explative Pronouns Grammatical constructions involving expletive pronouns, such as *there* or *it* are commonplace in everyday language, however generating correct semantic representation for them is non-trivial due a requirement to find a word in the remaining part of the sentence that the pronoun refers to. For example, in order to generate a semantic representation for a sentence: *It is difficult to outrun Jack* we have to know that *it* refers to *outrun*. Moreover, in case of *there* pronoun, a distinction between locative meaning (*There is an apple on the table*) and expletive (*There was some misunderstanding of the issue*) has to be made.

Contractions Contractions such as *aren't*, *she'll* or *I've* are problematic because parser generates a single tree node for them rather than two nodes. In order to cope with that issue, a preprocessing step should be added in which contractions are expanded to two words. This opens another issue of distinguishing between contraction and possessive (case of *'s*), which could be solved by relying on part-of-speech tag, which however is not always correct. Similar difficulty occurs with contraction *'d*.

Possessive Pronouns Possessive pronouns are most of the time assigned a CCG category $NP/(N/PP)$. In order to support their semantics correctly, the current approach to predicate generation should be changed so that all possessive pronouns take the same *fixed* lemma and a constant which points to the *owner*, which is not the case for now. Such approach would require co-indexation and correct co-reference resolution.

6.1.3 Open Problems

In the following section, major issues related to English-to-ASP translation are outlined, all of which are subjects of research in Natural Language Processing or Linguistics.

Semantics of Prepositions Prepositions play an important role in composing the semantic representation of a sentence, they are usually used in front of nouns and pronouns to show their relation to other words in the sentence. However, the CCG parser assigns multiple categories to prepositions, some of which are argument categories like: PP/NP, some are adjunct: $((S \setminus NP) \setminus (S \setminus NP))/NP$ and the two types of categories are confused. Moreover, prepositions can also assume a *phrasal verb complement* category PR which happens to be mistakenly treated as other argument or some adjunct categories, as in case of a sentence *He handed in his homework*, where *in* is incorrectly assigned adjunct category $((S \setminus NP) \setminus (S \setminus NP))/NP$. Incorrect parser output makes reasoning about correct semantics of prepositional phrases difficult and automatic translation less robust. More importantly, prepositions often have context-dependent semantics, such as for example locative meaning of *in* for a sentence *Jack is in the house*, which should be interpreted correctly in order to represent semantics of a sentence.

Compound Nouns Deriving semantic representation of compound nouns compositionally is a challenging problem, which can be exemplified by noun phrases such as *income tax deduction* or *box of chocolates*. The former case *tax* and *deduction* could be merged into a single, non-compositional noun and *income* could act as a modifier. In the

latter case, compositional treatment of the semantics of the phrase, i.e. representing it as an entity that is both a *box* and a *chocolate* is perhaps appropriate. However, even more difficult to represent are idiomatic phrases such as *pair of jeans*. In general, deciding about how to derive meaning of compound nouns based only on the output of the syntactic parser is a difficult task.

Questions In the project we focused on a narrow subset of questions. In order to expand the set of questions for which ASP representations could be derived, the current approach would require greater CCG parser accuracy and extensions to the ASP-to-English translation mechanism. Reliable parsing of questions using CCG grammars has been a problematic task due to smaller sizes of available training corpora [14]. In our opinion different types of questions should be grouped into categories, such as *temporal*, *location*, *reason for* and similar and handled on per case basics so that a knowledge base query corresponding to the question is specific enough to retrieve the expected answer without spurious information. Such approach however is associated with substantial manual effort.

Software Dependencies The current approach relies on multiple NLP utilities, namely: CCG Parser, coreference resolver, part-of-speech tagger and lemmatiser, all of which are required to generate a fully correct semantic representation. State-of-the-art part-of-speech taggers and CCG super-taggers achieve 97% [39] and 95% [36] accuracy respectively, however in case of F1 score for CCG dependency parsing it is 88% [36] and for coreference resolution it is around 60% [34], which gives an idea about the upper bound on the accuracy of the translation algorithm. Let us notice that fully correct semantic representation is not always required as questions might be answered without using all the information present in the text. Nevertheless, performance of the dependencies is a limiting factor on the accuracy of the translation algorithm.

Example 6.1 (Example of CCG parser error)**.** Let us consider CCG parser trees and ASP representations of two sentences: *The bedroom is east of the kitchen* and *The bedroom is west of the kitchen* which differ on one word (*east* vs *west*).

Listing 6.1 CCG parse tree of a sentence *The bedroom is east of the kitchen.*

```
% CCG parse tree:
<T S[dcl]>
  <T NP>
    <L (NP/N) DT The>
    <L N NN bedroom>
  <T (S[dcl]\NP)>
    <L ((S[dcl]\NP)/(S[adj]\NP)) VBZ is>
    <T (S[adj]\NP)>
      <L ((S[adj]\NP)/PP) JJ east>
      <T PP>
        <L (PP/NP) IN of>
        <T NP>
          <L (NP/N) DT the>
          <L N NN kitchen>

% ASP representation:
binaryModif(east,c1,c2).
unaryNominal(c2,kitchen).
unaryNominal(c1,bedroom).

```

Listing 6.2 CCG parse tree of a sentence *The bedroom is west of the kitchen.*

```
% CCG parse tree:
<T S[dcl]>
  <T NP>
    <L (NP/N) DT The>
    <L N NN bedroom>
  <T (S[dcl]\NP)>
    <L ((S[dcl]\NP)/NP) VBZ is>
    <T NP>
      <L (NP/PP) NN west>
      <T PP>
        <L (PP/NP) IN of>
        <T NP>
          <L (NP/N) DT the>
          <L N NN kitchen>

% ASP representation:
binaryEvent(e0,be,c1,c3).
unaryNominal(c3,west).
unaryNominal(c3,kitchen).
unaryNominal(c1,bedroom).
metaData(0,e0).
```

In case of the first sentence (Listing 6.1) phrase *east of the kitchen* is correctly identified as an adjective phrase (line: 8) whereas phrase *west of the kitchen* in the second sentence (Listing 6.2) is incorrectly identified as a noun phrase (line: 8), which leads to an incorrect ASP representation.

6.1.4 Discussion

Evaluation shows that the approach performs favourably on simple sentences, which where the main target of the project. As the algorithm supports generating semantic representation for more complex grammatical structures, it was also evaluated on more complex sentences however, with mixed success.

When evaluated on news articles, the translation gives correct output mostly for short sentences. Longer sentences are more likely to contain parser errors and currently unsupported types of expressions (section 6.1.2) and hence yield incorrect representation. Evaluation on the news articles also showed a necessity to add support for other CCG combinatory rules, such as generalised composition or substitution [26], which are rarely used in case of simple sentences.

Boxer system [8], which takes a similar approach to the one presented in this project and produces a logical representation of text based on a CCG parser and λ-DRS calculus, is capable of deriving well-formed semantic representations for over 97% of unseen sentences taken from Wall Street Journal [10]. The semantic representation used by Boxer (Discourse Representation Structure) differs from the one used in this project as it corresponds more closely to the structure of a CCG parse tree which might make it easier to derive automatically.[1] However, the coverage achieved by Boxer and some successful results achieved by the current approach when evaluated on news stories show that there is a potential in the proposed approach to translation to cope with *real life* texts.

6.2 Learning Evaluation

Performance of the system in the learning setting was evaluated on *The (20) QA bAbi Tasks Dataset* (section 6.2.1) and compared to other approaches to question answering evaluated on the same set of tasks. The reason for choosing that particular dataset is simple structure of narratives yet a significant requirement for different types of reasoning.

6.2.1 The (20) QA bAbi Tasks Dataset

The (20) QA bAbI tasks [66] are a part of bAbI project developed by Facebook AI Research, whose aim is to motivate advancements in automatic text understanding and reasoning [66]. The tasks check general text understanding skills, such as identification of facts supporting a claim, time reasoning, deduction and it is postulated that performing well on all of them (achieving over 95% accuracy) is required for any system whose aim is full text understanding.

[1] The reason why the representation used by Boxer could not be used in the current project is its limited applicability for learning in ILP setting.

There are two versions of the dataset, one containing a thousand and the other ten thousand examples per task, however the former is preferable. In both cases, a test set of a thousand examples is provided per each task. Every example, instances of which are presented in Table 6.2, consists of a relatively short narrative ($2 - 30$ sentences) and a question whose answer can be derived based on the text. Each answer in the training set is accompanied by a set of line numbers referring to the associated text which point to statements relevant for answering the question and provide additional supervision for the model. There is no noise in the data for any of the tasks and human can potentially achieve 100% accuracy on the dataset.

Task 1 - Single Supporting Fact	Task 15 - Basic Deduction
1 Sandra moved to the hallway.	1 Wolves are afraid of cats.
2 Daniel went to the office.	2 Cats are afraid of wolves.
3 Where is Sandra? **hallway 1**	3 Mice are afraid of wolves.
4 Sandra travelled to the kitchen.	4 Gertrude is a cat.
5 John moved to the office.	5 Jessica is a mouse.
6 Where is John? **office 5**	6 What is Gertrude afraid of? **wolf 4 2**

Table 6.2: Training examples for two tasks from *the (20) QA bAbI tasks* dataset. Numbers following the answers point to the relevant statements in the preceding narrative.

The (20) QA bAbI tasks dataset is a synthetic dataset, examples for all tasks were generated with a simulation similar to a text adventure game whose world consists of entities of various types (locations, people, objects etc.) and interactions between them. The set of executable actions is constrained to movement, state changes and inspection of objects. To enhance the lexical variety of the generated short stories, entities and actions were assigned a set of replacement synonyms and some entities were replaced with pronouns. Nevertheless, the resulting narratives have simple grammatical structure and the vocabulary size is small (150 words). Authors hope that by keeping the language simple, tasks would focus on evaluating text understanding skills rather than general linguistic knowledge of an agent. Description of tasks present in the dataset is given in Table 6.3. To date, to the best of our knowledge, only one model managed to solve all 20 tasks [22].

6.2.2 Evaluation Set-up

Learning capabilities of the system developed in the project were evaluated on a subset of seven tasks from *The (20) QA bAbi Tasks Dataset*, namely tasks: $1, 6, 8, 9, 12, 15, 18$, on which the system was able to perform learning in a fully automated manner. Evaluations were performed on a workstation with Core i7-4770 3.40GHz processor, 16 GB of RAM, running Ubuntu 16.04.

Due to the small size of the vocabulary and the repetitive sentence structure in the dataset, CCG parser errors errors rendered learning tasks unsolvable as the same error would be repeated in majority of examples. To mitigate that issue, certain words were replaced. The list of replacements can be seen in Table 6.4.

Task	**Description**
1 - Single Supporting Fact 2 - Two Supporting Facts 3 - Three Supporting Facts	Answer to the question depends on a number of facts from the narrative, which have to be chained together and can be interwoven with irrelevant information.
4 - Two Arg. Relations 5 - Three Arg. Relations	Test agent's ability to extract relations from the text. *Fred gave football to Jeff* → gave(football, Jeff, Fred)
6 - Yes/No Questions	Questions with a single supporting fact are used to test models' ability to answer true/false questions.
7 - Counting 8 - Lists/Sets	Checks model's ability to count and list objects with a certain property, e.g. objects carried by a person.
9 - Simple Negation	Requires the model to handle negation in case one of the supporting facts yields a negative answer to a yes/no question, for example (following layout in Table 6.2): 1 Sandra is no longer in the kitchen. 2 Is Sandra in the kitchen? **no**
10 - Indefinite Knowledge	Tests model's ability to answer yes/no-style questions for which there might be no definite answer, in which case agent should answer **maybe**.
11 - Basic Coreference 12 - Conjunction 13 - Compound Coreference	Tasks 11 and 13 test coreference resolution when referent is a subject of the preceding sentence. In tasks 12 and 13 subjects are compound (joined using *and*).
14 - Time Reasoning	Requires understanding of time expressions like *in the afternoon*, *yesterday* by answering questions about order of events.
15 - Basic Deduction 16 - Basic Induction	Test basic induction and deduction via inheritance of properties - given properties of a class model should deduce properties of a single instance or, given an instance, model derives properties of a class.
17 - Positional Reasoning 18 - Size Reasoning	Checks model's ability to reason about relative position or sizes of different objects by asking yes-no questions about relations between objects.
19 - Path Finding	Given the description of relative position of different locations (e.g. *The hallway is south of the bathroom.*) the agent has to describe the path between locations by listing consecutive cardinal directions.
20 - Agent's Motivations	Tests model's ability to acquire basic common-sense knowledge by asking questions why a certain action was performed by a person in the narrative, for example: 1 Jason is thirsty. 2 Where will jason go? **kitchen 1** 3 Jason went to the kitchen. 4 Why did jason go to the kitchen? **thirsty 1**

Table 6.3: Description of tasks present in *The (20) QA bAbI tasks* dataset. Our system was evaluated on tasks 1, 6, 8, 9, 12, 15, and 18.

Task	Word	Replacement
1, 12	Mary	Jack
6, 9	office	study
8	is carrying	does carry
15	sheep	fox

Table 6.4: Replacements of words and phrases causing parser errors in *The (20) QA bAbI tasks* dataset.

6.2.3 Evaluation Method

In all experiments, for each of the seven learning tasks, the training set was divided into two sets of 800 and 200 examples, the former used for training and the latter for validation. Then, 10 subsets of examples, whose size depended on particular experiment, were sampled from the set reserved for training, learning was performed on each of them and the learned hypothesis was evaluated on the validation set. The set of learned rules that performed best on the validation set (Appendix D) was evaluated on the test set.

The evaluation procedure was motivated by the empirical observation that, in case of most tasks from *The (20) QA bAbi Tasks Dataset*, is not necessary and, in some cases, not feasible to use all available training data. As shown in section 6.2.5, in case of tasks for which the optimal set of rules is not contained in the hypothesis space and hence the learner uses the learning with noise feature of ILASP, the learning times increase sharply. Therefore, training on the full training set was not possible, which the aforementioned evaluation method addressed. As length of the narratives in the dataset varies, multiple samples of training examples were taken to compensate for some of them being less informative.

Let us point out that neural network-based models described in [22] and [61], against which we compare our results in Section 6.2.8, also used a validation set in their training procedure. However, in their case the validation set was used in order to select the optimal model architecture and hyperparameters [61] or the best set of initial values of parameters of the model [22].

6.2.4 General Learning Capabilities

The aim of the first evaluation was to assess the general capabilities of the learning algorithm by training on 25 examples, which was empirically determined to be enough to learn the underlying concept. The experiment was performed using the method described in section 6.2.3.

Task	1	6	8	9	12	15	18
Accuracy	**100.0**	**98.9**	**100.0**	**97.0**	**100.0**	**100.0**	**92.4**

Table 6.5: Accuracies obtained on a subset of tasks from *The (20) QA bAbi Tasks Dataset* using the best set of rules obtained using the approach described in the current section.

Based on table 6.5 and using the criteria outlined in [66] (task qualifies as fully solved if over 95% accuracy is achieved) it can be concluded that the system fully solved six tasks from the

dataset. To the best of our knowledge, this is the only system relying on symbolic machine learning capable of learning on *The (20) QA bAbi Tasks Dataset* in a fully automated manner. The sets of rules learnt for each task are shown in the Appendix D.

6.2.5 Minimum Required Number of Training Examples

The second experiment measured the improvement of learner's performance with the higher number of training examples in attempt to estimate the minimum number of data points necessary to learn the underlying concept. Training was performed using the method outlined in Section 6.2.3. The accuracies achieved on the test set are presented in Table 6.6[2] and the mean accuracies achieved on the validation set, together with the corresponding standard deviation, are shown in Table 6.7.

	Task						
Ex.	**1**	**6**	**8**	**9**	**12**	**15**	**18**
5	64.8	90.7	92.4	74.6	100.0	100.0	81.7
10	100.0	90.7	92.4	81.8	100.0	100.0	87.9
15	100.0	92.0	100.0	90.6	100.0	100.0	91.2
20	100.0	95.8	100.0	94.6	100.0	100.0	88.2
25	100.0	98.9	100.0	97.0	100.0	100.0	92.4

Table 6.6: Accuracy values registered on the *test set*, obtained for a subset of *The (20) QA bAbi Tasks Dataset* tasks with a different number of examples used for training. The number of training examples was increased from 5 to 25 in steps of 5

Results in Table 6.6 show that 10 training examples are sufficient to achieve 80.0% or higher accuracy on the test set and 15 examples are sufficient to achieve over 90.0% accuracy.

	Task						
Ex.	**1**	**6**	**8**	**9**	**12**	**15**	**18**
5	55.4 ± 14.1	94.0 ± 3.0	81.5 ± 26.0	65.4 ± 6.2	71.8 ± 12.7	100.0 ± 0.0	74.5 ± 8.3
10	80.5 ± 16.0	90.9 ± 7.4	76.5 ± 22.1	68.3 ± 7.7	85.4 ± 11.1	100.0 ± 0.0	80.8 ± 9.8
15	88.5 ± 18.0	90.8 ± 6.9	89.0 ± 10.5	78.2 ± 11.3	94.7 ± 11.9	100.0 ± 0.0	90.5 ± 4.5
20	94.4 ± 7.3	91.7 ± 6.0	91.3 ± 12.3	81.8 ± 6.4	100.0 ± 0.0	100.0 ± 0.0	76.8 ± 15.0
25	97.1 ± 6.2	94.8 ± 4.4	96.5 ± 5.1	90.0 ± 5.0	100.0 ± 0.0	100.0 ± 0.0	78.9 ± 29.4

Table 6.7: Mean accuracy values and the corresponding standard deviations registered on the *validation set* when training on *The (20) QA bAbi Tasks Dataset*, according to the method described in the Section 6.2.3, using different number of training examples. The number of training examples was increased from 5 to 25 in steps of 5

Table 6.7 shows that the mean accuracy values for all tasks apart from task 18, in general, increase monotonically with the number of training examples, which is the expected behaviour. The difference for task 18 can be explained by the fact that in order to solve the task completely, recursive concepts have to be learnt, which are currently not supported by our system (Section 4.2.1). Therefore, *learning with noise* feature of ILASP is used and the examples which require recursive rules to be covered are implicitly treated as "noisy".

[2]The number of training examples is specified in the **Ex.** column.

Consequently, when the noisy examples are present in a sample, the runtime increases exponentially with the size of the sample (Section 6.2.6) which in case of larger samples (20 or 25 examples) might lead to a timeout of the the learning task. In such situation, no hypothesis is learnt and the accuracy achieved on the validation set is equal to the one obtained by answering *no* to all questions (in task 18 only polar questions are used), which causes the the decrease of the mean accuracy and increase of the standard deviation.

Let us also notice that in case of tasks 6 and 8, the mean accuracy obtained when 5 examples are used is greater than in case of some larger sample sizes. This is caused by the fact that, in these two cases, using few examples causes overly general hypotheses to be learnt which perform relatively well (over 90% accuracy) on the validation set. However, when more examples are provided, the learnt hypotheses are more specific and, in cases when the training examples are not representative of the entire dataset, the learnt hypotheses do not generalise well to the unseen data.

Regarding the main question behind the current experiment, the minimum number of training examples required to learn a concept is task specific and reflects the complexity of the underlying hypothesis. However, 15 examples allow for all tasks to achieve over 90% accuracy on the test set and for all but one task this number is sufficient to achieve approximately 90% accuracy on the validation set. Moreover, the standard deviation of accuracy values, as reported on the validation set, is relatively large for 15 examples. This indicates that accuracies higher than the reported mean can be achieved if the examples are *well-chosen*, which is confirmed by the results in Table 6.6.

6.2.6 Average Learning Times

In this experiment, average end-to-end learning times recorded during training on the subset of *The (20) QA bAbi Tasks Dataset* were measured. The learning times were gathered alongside the accuracies presented in Table 6.6 and Table 6.7.

	Task						
Ex.	**1**	**6**	**8**	**9**	**12**	**15**	**18**
5	3.125	2.074	3.353	2.431	4.435	3.495	9.871
10	4.999	4.428	12.952	35.947	5.154	7.178	19.226
15	6.367	6.131	40.006	48.043	6.960	7.555	75.259
20	7.190	8.018	82.366	184.046	8.967	9.941	341.197
25	9.829	15.508	107.857	229.595	11.048	12.056	237.296

Table 6.8: Average end-to-end learning times (in seconds) for the given number of examples, corresponding to the results form Table 6.6. The number of training examples was increased from 5 to 25 in steps of 5 and learning times were averaged over 10 runs.

Regarding the average learning times presented in Table 6.8, in case of tasks in which no noise is present (which is the case for all tasks apart from 18), they increase exponentially with the number of training examples until the upper limit on the accuracy is reached, which corresponds to the underlying concept being learnt fully. From that point the increase in the learning time is linear. In case of task 18 the correct hypothesis could not be expressed and hence the learning with noise feature of ILASP is used. In this case, the average learning

time for each sample size depends strongly on the percentage of samples that contain noisy examples, which are distributed randomly across the dataset.

6.2.7 Questions Answering on *The (20) QA bAbi Tasks Dataset* Using Background Knowledge

As described in section 6.2.4, seven tasks could be solved by relying on the rules derived by the learning algorithm. However, there are three tasks, namely 2, 5 and 16 (Table 6.3), for which learning does not succeed because of: lack of support for predicate invention, counter-intuitive expected answers and excessively long correct hypothesis respectively. Nevertheless, the tasks could to some extent be solved by relying on additional background knowledge rules (Appendix B). The results of running the tasks can be seen in Table 6.9.

Task	Accuracy
2	100.0
5	73.8
16	93.6

Table 6.9: Question answering evaluation using manually added background knowledge.

Using a human-readable representation allows for adding extra set of rules necessary to, at least partially, solve the tasks. These tree tasks point to a broader problem of creating and managing knowledge bases to aid the task of question answering, which is a separate topic that could be explored further in context of our system as a part of the future work.

6.2.8 Comparison to Other Approaches

To the best of our knowledge, the approach presented in [44] is the only other system relying on symbolic machine learning that was evaluated on *The (20) QA bAbi Tasks Dataset*. It achieves 100% accuracy on all tasks apart from task 16, on which it scores 93.6%. However, their approach to generation of ASP representation is not general-purpose and the system requires manual specification of mode declarations and task-dependent background knowledge. Therefore, the system is excluded from the further comparison.

Among statistical approaches, Recurrent Entity Network (EntNet) introduced in [22] fully solves all tasks and has mean error of 0.5%, however it requires the version of the dataset with 10k training examples to achieve that result and its accuracy significantly decreases when 1k examples are used. Another neural architecture, End-To-End Memory Network (MemN2N), is a type of a memory network introduced in [61] and it performs better on a smaller dataset (containing 1k examples) than Recurrent Entity Network.

As can be seen from Table 6.10, the approach developed in the project compares favourably with other question answering systems evaluated on the same dataset. It consistently outperforms Recurrent Entity Network trained on 1k examples and performs better than End-To-End Memory Network on all tasks other than 9. A closer examination of task 9 revealed that our system did not learn the optimal set of rules, which could be mitigated by supplying more than 25 examples.

	Task						
System	**1**	**6**	**8**	**9**	**12**	**15**	**18**
Sukhbaatar et al. (MemN2N)	99.9	98.0	93.9	98.5	100.0	98.2	90.8
Henaff et al. (EntNet)	99.3	70.0	80.8	68.5	99.2	42.2	91.2
Report (CCG + ILASP)	**100.0**	**98.9**	**100.0**	**97.0**	**100.0**	**100.0**	**92.4**

Table 6.10: Accuracies achieved on the selected tasks from *The (20) QA bAbi Tasks Dataset* by different question answering systems. MemN2N and EntNet were trained using 1k examples and out system was trained using 25 examples.

A couple of important points should be made regarding the comparison. Firstly, all other systems listed in Table 6.10 can achieve over 90.0% accuracy on the remaining 13 tasks from the dataset, which our system currently cannot do. Moreover, MemN2N and EntNet were also evaluated on other *real life* text comprehension datasets and, as stated in [22], "were able to obtain decent performance", which is currently beyond the reach of our system.

However, the unique feature of our approach is that it can achieve relatively high accuracy (80% or more) using only 10 examples which, to the best of our knowledge, neural network approaches are not capable of. This characteristic might prove valuable in domains where training examples are scarce.

6.2.9 Hypothesis Space Reduction

In this experiment the impact of argument and predicate typing performed when generating mode declarations on the size of the hypothesis space is measured. As to the best of our knowledge there is no work in automatic derivation of mode declarations against which our approach could be compared, a baseline approach was devised to perform the evaluation.

In the baseline approach, no predicate or argument typing is performed. Lack of predicate typing means that mode declarations are derived based only on the predicate names. As no argument typing is performed, all predicate arguments which do not occur at the lemma position are assigned the same *variable* type in the mode declarations. *lemma* arguments are assigned different types based on the names of predicates in which they occur.

Example 6.2 (Baseline mode declarations)**.** Let us compare the mode declarations generated by the algorithm used in the project (Listing 6.3) against the baseline approach (Listing 6.4) on Task 1 from *The (20) QA bAbi Tasks Dataset*. It should be noted that for each set of mode declarations, the corresponding hypothesis space contains the solution to the learning task.

Listing 6.3 Mode declarations generated using predicate and argument typing.

```
1: #modeh(binaryInitEventH(var(v2),const(c3),var(v1),var(v0))).
2: #modeh(abBinaryEventH(var(v2),const(c3),var(v1),var(v0))).
3: #modeh(binaryTermEventH(var(v2),const(c3),var(v1),var(v0))).
4: #modeh(binaryEventH(var(v2),const(c3),var(v1),var(v0))).
5:
6: #modeb(1,binaryEvent(var(v2),const(c4),var(v1),var(v0))).
7: #modeb(1,unaryModif(const(c5),var(v2))).
8: #modeb(1,unaryNominal(var(v0),var(v6))).
9: #modeb(1,unaryNominal(var(v1),var(v7))).
10:
```

```
11: #constant(c3,be).
12: #constant(c4,move).
13: #constant(c4,journey).
14: #constant(c4,go).
15: #constant(c4,travel).
16: #constant(c5,back).
17:
18: #maxv(5).
```

Listing 6.4 Baseline mode declarations.

```
 1: #modeh(binaryInitEventH(var(v0),const(c3),var(v0),var(v0))).
 2: #modeh(abBinaryEventH(var(v0),const(c3),var(v0),var(v0))).
 3: #modeh(binaryTermEventH(var(v0),const(c3),var(v0),var(v0))).
 4: #modeh(binaryEventH(var(v0),const(c3),var(v0),var(v0))).
 5:
 6: #modeb(1,binaryEvent(var(v0),const(c4),var(v0),var(v0))).
 7: #modeb(1,unaryModif(const(c5),var(v0))).
 8: #modeb(1,unaryNominal(var(v0),var(v1))).
 9:
10: #constant(c3,be).
11: #constant(c4,move).
12: #constant(c4,journey).
13: #constant(c4,go).
14: #constant(c4,travel).
15: #constant(c5,back).
16:
17: #maxv(5).
```

In order to perform the evaluation, for each of the seven problems from *The (20) QA bAbi Tasks Dataset* (problems: 1, 6, 8, 9, 12, 15, 18), the successful learning task is modified to conform by the baseline approach (the mode declarations used in the tasks were modified not to rely on type information). The size of the hypothesis space generated by the original task is compared to the one generated by the baseline method. In both cases, the effect of adding the fixed bias constraints, as described in Section 4.4, is inspected. The results of the evaluation can be seen in Table 6.11.

Task	Baseline	Baseline + Bias	Types	Types + Bias	Ratio
1	50345	16942	535	248	203.0
6	29627	9544	2837	1099	26.6
8	183677	35145	4273	1523	120.6
9	3854469	797310	26559	7025	548.7
12	68110	19914	626	248	274.6
15	8364	3696	1995	995	8.4
18	443985	335898	6228	3210	138.3

Table 6.11: Comparison, in terms of the size of the corresponding hypothesis space, of the algorithm generating mode declarations used in the project to the baseline that performs predicate typing based exclusively on the predicate name. For both methods, versions with and without the fixed mode bias are evaluated. The *ratio* is between the sizes of the hypotheses spaces generated using the baseline and the algorithm applied in the project when the mode bias included (i.e. between the second and the fifth column in the table).

There are three points that should be made about the results presented in Table 6.11. First, it is necessary to perform some sort of predicate and argument type inference when generating mode declarations as the sizes of hypotheses spaces generated by the baseline approach

are prohibitively large. Secondly, the benefit of including type information depends greatly on the learning task, however, on average, taking into account type information reduces the size of hypothesis space dramatically. Finally, even though the same fixed mode bias is used for all tasks, it contributes a significant reduction (approximately $2 - 5$ times) in the size of the hypothesis space. We expect that further benefits could be achieved if task-specific bias constraints were generated automatically, which is described in detail in Section 8.1.2.

6.2.10 Discussion

During the course of the project two major bottlenecks to wider applicability of the learning algorithm were identified. First one is the accuracy of the CCG parser and the second is the size of the hypothesis space corresponding to the automatically generated learning tasks.

Incorrect dependency structure and CCG category annotations generated by the CCG parser are a major obstacle to solving more task in *The (20) QA bAbi Tasks Dataset*. These issues are relevant to tasks 3, 4, 11, 13, 14, 17 and 19. However, development of syntactic parsers is an active research area that recently saw much progress due to adaptation of deep learning techniques, LSTM networks in particular [36]. Therefore, the accuracy of CCG parser can be expected to improve in the future.

Excessively large hypothesis spaces are an issue that prevented derivation of the optimal hypothesis for task 16. In this project a lot of thought was devoted to the problem of automatically generating compact hypothesis spaces, yet expressive enough to contain the optimal solution. However, the developed approach could be further refined by automatically generating task-specific mode bias constraints that would account for co-occurrence patterns of predicates constituting example contexts.

Regarding the other unsolved tasks, passing them would require support for additional features in translation algorithm, learning algorithm and ILASP. Task 2 requires object invention, task 7 requires adding support in ILASP for counting aggregates in the rule body, task 10 requires inclusion of choice rules in the automatically generated ASP representation in order to correctly support semantics of logical alternative. Task 20 requires correct handling of future tense in the semantic representation. All these issues could be addressed given a longer project time frame.

Chapter 7

Related Work

In this chapter, other projects related to translation from English to formal representations and answering questions on *The (20) QA bAbi Tasks Dataset* are described.

7.1 Translating from English to Formal Representations

7.1.1 λ-ASP Calculus

λ-ASP calculus was introduced by Baral et al. in [5]. The formalism conceptually follows Montague grammar as it establishes a mapping between syntax and semantics, represented in ASP, of natural language.

λ-ASP calculus relies on a CCG derivation to represent the syntactic structure of a sentence. As presented in [5], the lexicon is specified as a mapping from word - CCG category pairs to λ-ASP expressions. The formalism uses ASP syntax and lambda calculus as "semantic glue" to combine the final semantic representation from constituents via function application (β - reduction). Table 7.1 provides λ-ASP expressions for example words and their CCG categories, as presented in [5].

Word	**CCG Category**	**λ-ASP expression**
fly	$S \setminus NP$	$\lambda x.fly(x)$
penguin	N	$\lambda x.penguin(x)$
fictional	$N \setminus N$	$\lambda x.\lambda y.fictional(y) \wedge x@y$
most	N/N	$\lambda x.\lambda y.(y@X \leftarrow x@X \wedge not \neg y@X)$
do not	$(S/(S \setminus NP)) \setminus NP$	$\lambda x.\lambda y.\neg y@X \leftarrow x@X$

Table 7.1: λ-ASP expressions for example words and their CCG categories [5].

Example 7.1 (Derivation of ASP representation of a sentence *Most fictional penguins fly*)**.** Given the mapping from words and CCG categories to λ-ASP expressions (Table 7.1), ASP representation of a sentence *Most fictional penguins fly.* can be derived as follows:

$$\dfrac{\dfrac{\dfrac{Most}{\lambda x.\lambda y.(y@X \leftarrow x@X \wedge not\neg y@X)} \quad \dfrac{\dfrac{fictional}{\lambda x.\lambda y.fictional(y) \wedge x@y} \quad \dfrac{penguins}{\lambda x.penguin(x)}}{NP : \lambda y.fictional(y) \wedge penguin(y)}>}{NP : \lambda y.(y@X \leftarrow fictional(X) \wedge penguin(X) \wedge not\neg y@X)}> \quad \dfrac{fly}{\lambda x.fly(x)}}{S : fly(X) \leftarrow fictional(X) \wedge penguin(X) \wedge not\neg fly(X)}<$$

In the above derivation, CCG categories for the lexical items are omitted. Every derivation step is annotated with the applied CCG combinatory rule ($>$ stands for forward and $<$ for backward function application).

The version of λ-ASP calculus presented in [5] was described as "a first step towards automatically translating natural language statements to theories in ASP" and as such had significant limitations. Firstly, as authors indicate in [5], the formalism was confined to a set of sentence analysed in their publication. Secondly, the presented version of λ-ASP calculus did not support more complex constructs such as adverbs or conjunction. Both of those problems were to some extent addressed in the future work [7] [6] by using statistical machine learning approach to learning CCG lexicon from annotated examples, similarly as it was done in [70].

However, to the best of our knowledge, λ-ASP expressions as introduced in [5] were only used for generating domain-specific semantic representations [6] and were never evaluated on non-synthetic examples. Moreover, no theoretical underpinnings for solving the more complex translation problems mentioned in [5] without access to annotated data were provided and the annotated corpus mentioned in [6] was not released.

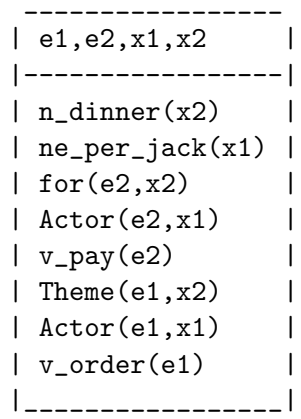
Consequently, the semantics of λ-ASP expressions used in this project was developed mostly independently from the work published by Baral et al. Moreover, version of λ-ASP calculus outlined in this report can handle issues listed as future work in [5], as well as even more advanced grammatical and linguistic constructions such as, among others, relativisation, control and raising.

7.1.2 Boxer

Boxer is a wide-coverge tool for generating semantic representations from text, developed by Bos et al. [10], [8]. The system relies on CCG derivation structure produced by the C&C parser [15] to generate a semantic representation known as Discourse Representation Structures (DRSs) using the idea of λ-DRS calculus. DRSs consist of a set of discourse referents and a set of constraints on their interpretations and they can deal with a number of contextually sensitive phenomena, such as ellipsis and presupposition. DRSs can also be translated to ordinary first-order logic formulas.

Example 7.2 (Discourse Representation Structure)**.** Sentence *Jack ordered and paid for the dinner* has the following corresponding DRS:[1]

[1]DRS is presented in the format used by the Boxer tool.



In order to derive the semantic representations, 245 most frequent CCG categories produced by C&C parser [15] were annotated with the corresponding λ-DRS expressions, following Montague-style semantics, and the combinatory rules of CCG grammar were reformulated in terms of the target semantic representation. Then, to derive semantic representation of a given sentence, semantic representations were assigned to each lexical item and β-reduction was applied to constituents according to the structure of the CCG derivation in a bottom-up fashion.

The system, in combination with a theorem prover, was evaluated on a task of recognising textual entailment (RTE) [9] which requires an agent to find out whether some text T entails a hypothesis H. The semantic representation generated by the system was enriched with three types of background knowledge: generic knowledge added manually, lexical knowledge extracted from WordNet, and geographical knowledge automatically extracted from CIA factbook. However, relying on theorem prover alone was insufficient to perform well on the RTE task, the system managed to find proofs for only 5.8% of examples from the test set and its precision was 76.7%, which overall allowed it to achieve 52.0% accuracy against a 50.0% most frequent class baseline [2]. Relying on theorem proving was found to overgenerate the FALSE class - it was often unable to recognise correct entailment, which was explained by a difficulty to automatically acquire relevant background knowledge [9].

The approach to translating English to a formal representation used in Boxer is similar to the one adopted in this project. Both systems rely on CCG parse tree of a sentence and λ-X intermediate representation, where X is the name of the target formal representation, to derive the final semantic representation compositionally. Also, in both cases the lexicon is an *algorithm* rather than a predefined mapping, which arguably contributes to higher coverage of natural language text by the Boxer system than it is the case for approaches relying on annotated training data to learn the lexicon, as in [70]. Moreover, both systems rely on other resources, such as part-of-speech tags, in addition to the CCG category in order to derive lexical semantics of a word.

The main difference between the two systems is the target formal representation used. In case of our system it is ASP, which enables it to learn new hypotheses from text. To the best of our knowledge, there is no experimental evidence of suitability of the semantic representation produced by Boxer for logic-based learning.

[2] In the test set, 50.0% of example sentence pairs (T, H) were annotated as TRUE (T entails H) and 50.0% as FALSE.

7.2 Lexicon Creation

Different approaches to semantic parsing are distinguished, among others, by the method used for lexicon creation. Lexicons in Boxer (section 7.1.2) and in our project rely on handcrafted set of general translation rules. There are however alternative approaches that learn the lexicon structure from annotated data and are the subject of this section.

7.2.1 Cornell SPF

Cornell Semantic Parsing Framework [1] (previously known as University of Washington Semantic Parsing Framework) removes the dependency on syntactic parser and the need for manual semantic annotations of CCG categories by learning the parse structre (CCG categories of words) and lambda calculus encoding of semantics from a dataset of sentences paired with semantic representations. The framework relies on the approach outlined in [70], which constitutes a basis for many future developments in supervised approaches to semantic parsing.

The approach presented in [70] relies on probabilistic extension of CCGs, namely PCCGs, where every pair of syntactic derivation (CCG parse tree) T and semantic representation (logic formula) L of a given sentence S is assigned a probability:

$$P(L,T|S,\bar{\theta}) = \frac{\mathrm{e}^{\bar{f}(L,T,S)\cdot\bar{\theta}}}{\sum_{(L,T)} \mathrm{e}^{\bar{f}(L,T,S)\cdot\bar{\theta}}} \tag{7.1}$$

where $\bar{\theta} \in \mathrm{R}$ is the parametrisation of the model and $\bar{f}$ is a feature function. Then, parsing under PCCG can be reformulated as a problem of finding the most probable logical form L for a sentence S:

$$\arg\max_{L} P(L|S;\bar{\theta}) = \arg\max_{L} \sum_{T} P(L,T|S;\bar{\theta}) \tag{7.2}$$

where the sum is over all syntactic derivations (parse trees) T that yield a semantic representation L. Given such problem formulation, the task is to learn a CCG lexicon Λ together with parameter vector $\bar{\theta}$ from a set of n training examples $\{(S_i, L_i) : i = 1 \ldots n\}$ where each training example is a sentence S_i paired with a semantic representation L_i. Let us notice that the syntactic derivation T is treated as a hidden variable in the model.

Therefore, the problem can be decomposed into a *structure learning* problem - learning the lexicon, and *parameter estimation* problem - estimating $\bar{\theta}$. The former is achieved by the GENLEX(S_i, L_i) procedure [70], which given a logical form L_i relies on a set of handcrafted rules to derive a set of lexicon entries (CCG category paired with lambda calculus encoding of semantics) for constituents of sentence S_i corresponding to the predicates of L_i. Parameter estimation is performed using stochastic gradient descent.

The approach was evaluated on translating queries from Geo880 (set of 880 queries to a database of U.S. geography) and Jobs640 (set of 640 queries about job listing) both of which were originally specified in Prolog style semantics and had to be manually translated to lambda calculus expressions [70]. The described approach at its time achieved state-of-the-art precision and nearly state-of-the-art recall.

Using a supervised semantic parser, such as Cornell SPF, should allow solving a very specific task, such as parsing English sentences to commands in a robot control language descirbing robot's movement [42], with high accuracy. However, in order to extend the approach to another domain, additional annotated training data would be required to retrain the parser. Lack of a suitable dataset for learning the lexicon coupled with an intention to explore ASP-to-English translation in a broader context were the main reasons why the supervised approaches were not adapted in our project.

7.3 Question Answering on *The (20) QA bAbi Tasks Dataset*

Learning common sense knowledge rules from text was the one of the two main objectives of the project. In this section, other systems evaluated on *The (20) QA bAbi Tasks Dataset* are described.

7.3.1 Simple Knowledge Machine

In [44] a logic-based approach to question answering was proposed, which relies on Abstract Meaning Representation [4], Event Calculus and XHAIL learning algorithm to acquire common sense knowledge from narratives present in *The (20) QA bAbi Tasks Dataset*. When evaluated on the aforementioned dataset, the system achieved 100% accuracy on 19 out of 20 tasks.

The architecture of the system could be divided into two major components. The first one is a semantic parser performing translation from English to ASP. The parser relies on the output of JAMR [19], which itself is a semantic parser producing Abstract Meaning Representation of a sentence (roughly speaking, a graph indicating dependencies between words and annotated with their syntactic roles). Abstract Meaning Representation is converted to ASP program using a rule based approach.

In the second part, XHAIL is used to acquire common sense knowledge from the text in the form of Event Calculus theories. As most of the texts in the *The (20) QA bAbi Tasks Dataset* are narratives with an implicit timeline induced by the order of sentences, and size of vocabulary used in the dataset is small, relying on Event Calculus yields satisfactory results. In [44] it was even successfully applied for a simple case of coreference resolution. However, let us point out that the system presented in [44] required significant user supervision to perform the learning, both in a form of (sometimes extensive) task-specific background knowledge and manually specified mode declarations.

One similarity between our approach and the one proposed in [44] is the conceptual breakdown of the system into independent *parsing* and *learning* parts. Also, both systems rely on ASP representations to perform learning, however details of the representations differ significantly as we strived to make our representation applicable in a more general learning setting where task-specific background knowledge is not required and constraints on the hypothesis space are inferred automatically. Moreover, as our translation algorithm does not depend on a supervised semantic parser but a general-purpose syntactic parser, it can parse sentences from other domains without the need for being retrained with additional annotated examples. However, the increased generality of our approach to translation comes

at a price of lower robustness on *The (20) QA bAbi Tasks Dataset* due to syntactic parser's errors. It is also worth pointing out that the learning algorithm used in our project - ILASP supports both brave and cautious induction, as opposed to XHAIL that supports brave induction only, which allows ILASP to learn Answer Set Programs containing normal rules, choice rules and constraints, the last two of which XHAIL cannot learn [31].

7.3.2 End-To-End Memory Network (MemN2N)

Memory Network was one of the first neural architectures evaluated on *The (20) QA bAbi Tasks Dataset*. The version of the model proposed in [61], called End-To-End Memory Network (MemN2N), achieves over 90% average accuracy across all tasks (when trained with 10k examples) and its modifications were successfully applied to non-synthetic datasets [24]. Current state-of-the-art models for *The (20) QA bAbi Tasks Dataset* [22] rely on theoretical underpinnings of the MemN2N model.

MemN2N model, similarly to other neural network based approaches, uses the idea of *word embedding* and represents words and sentences as d - dimensional real valued vectors. For an ordered set of inputs $\{x_1, \ldots, x_n\}$ and a query q whose words belong to a set V, referred to as *vocabulary*, MemN2N computes a probability vector $\hat{a} \in \mathrm{R}^{|V|}$ whose maximum valued entry corresponds to the answer $a \in V$ to query q.

The parametrisation of the MemN2N model is given by four matrices $I_x, I_q, O \in \mathrm{R}^{d \times |V|}$ and $R \in \mathrm{R}^{|V| \times d}$ whose roles, according to [65], can be intuitively understood as follows:

Input Feature Maps I_x**,** I_q convert the inputs and the query to the internal representation used by the model, which are d-dimensional embeddings

Output Feature Map O produces a new output in the d - dimensional embedding space given the query and the current state of the memory

Response R converts the output from the d - dimensional embedding space to a desired response format

Both inputs $\{x_1, \ldots, x_n\}$ and query q are sequences of words and in [61] they correspond to sentences and a question respectively. Word sequences can be represented in a d-dimensional embedding space using a bag-of-words representation that is obtained by summing embeddings of words forming the sequence. More precisely, given a sequence of words s of length l and embedding matrix $A \in \mathrm{R}^{d \times |V|}$, the corresponding embedding is given by: $\sum_{i=1}^{l} As_i$ where $s_i \in \mathrm{R}^{|V|}$ are embeddings of constituent words represented, for example, using one-hot encoding.

On a high level, a single layer MemN2N model takes the following steps to generate answer a given inputs $\{x_1, \ldots, x_n\}$ and query q:

- for each input x_j, embedding $m_j \in \mathrm{R}^d$, referred to as *memory*, is generated using the input feature map I_x. Similarly, embedding u of query q, referred to as *internal state*, is computed using the feature map I_q
- for each memory m_j, a match p_j between m_j and internal state u is computed using:

$$p_i = \mathrm{softmax}(u^T m_j) = \frac{\mathrm{e}^{u^T m_j}}{\sum_i \mathrm{e}^{u^T m_i}} \tag{7.3}$$

where the summation is taken over all memories m_i. Let us notice that p, as defined in 7.3, is a probability vector

- for each input x_i, an output embedding o_i is computed using output feature map O
- *response vector* r is a sum of the output vectors o_i weighted by matches p_i, $r = \sum_i o_i p_i$, where summation is performed over all inputs
- prediction vector $\hat{a}$ of dimension V is computed using:

$$\hat{a} = \text{softmax}(R(r + u))$$

where *softmax* function is applied to each entry of $|V|$ - dimensional vector $w = W(o + u)$, which gives a probability distribution over the input vocabulary.

- finally, answer a is given by:

$$a = \arg\max_{v \in V} \ \hat{a}(v)$$

a is the word from the input vocabulary with the highest probability assigned by $\hat{a}$

The parameters of MemN2N are jointly learned by minimising the cross - entropy loss between prediction vector $\hat{a}$ and the true label $\bar{a}$. Training is performed using stochastic gradient descent [61].

To improve the expressiveness of the MemN2N model, more layers can be added, which can be achieved as follows (k denotes the layer number):

- for $k > 1$, the internal state u^k is equal to the sum of the output and internal state of the preceding layer, $u^k = u^{k-1} + o^{k-1}$
- each layer k has its own input and output feature maps I_x^k and O^k
- output $\hat{a}$ of a network with K layers is given by $\hat{a} = \text{softmax}(W(o^K + u^K))$

In order to limit the number of parameters of the network and hence facilitate training, various weight-tying schemes can be applied, an overview of which is given in [61].

Let us notice that the approach to question answering implemented by MemN2N is a major departure from the work undertaken in this project. Memory networks, as described in [61], do not rely on any prior linguistic knowledge and produce the answers based solely on the information they learn from annotated data. Moreover, let us notice that in the case of MemN2N model composition of meaning is realised by simple vector addition, yet it is sufficient to capture semantics of more complicated sentences occurring in, for example, children's books [24]. This approach to semantic composition is very different from using a CCG parse tree and λ - calculus expressions as performed in our project, which points to significant differences between logic-based and distributional approaches to representation of meaning. Let us also point out that both in case of MemN2N and our approach, queries require special treatment (in case of MemN2N they have a separate input feature map I_q and in our approach a separate lexicon mode is used) which might indicate that representing semantics of questions is a more universal problem.

Chapter 8

Conclusion

In this project we developed a logic-based approach to natural language understanding that is capable of performing inference and learning to answer questions about text. Developing the system required combining ideas from the fields of logic-based learning, knowledge representation and computational linguistics. Achievements of the project are as follows:

- ASP representation of natural language text suitable both for inference and learning using Inductive Logic Programming
- Extension of λ-ASP calculus, used to compositionally derive ASP representations of text, to handle more complex linguistic constructions
- Implementation of a systematic English-to-ASP translation algorithm that is general purpose and rooted in the theory of formal grammar
- Implementation of argument and predicate typing algorithm used for automatic derivation of mode declarations
- Implementation of a system capable of efficiently learning new background knowledge rules from text by generating multiple learning tasks and running them in parallel
- Comparison of the system to the neural network-based approaches on *The (20) QA bAbi Tasks Dataset*

8.1 Future Work

Given the breadth of the problem of machine comprehension of text, there are many possibilities for future development of the project. In what follows, we will focus on the ones that could bring the most benefit in terms of variety of text that the system can handle and learning capabilities.

8.1.1 Predicate and Object Invention

As stated in [18] "Predicate invention in ILP and hidden variable discovery in statistical learning are really two faces of the same problem". According to the paper, discovering a

hidden (latent) structure in the data is a central problem in machine learning, without such capability learning will always be *shallow*. Let us notice that in case of the representation devised for our system, the problems of predicate and object invention could be handled in exactly the same way, as the set of predicates used in the system is fixed. Consequently, predicate invention in our case amounts to adding a *fresh* constant in the position of the predicate *lemma* and object invention to adding a constant in the argument position.

Example 8.1 (Predicate Invention)**.** Let us illustrate the process using an example narrative and a question: *Jack collected the ball. Jack is in the kitchen. Where is the ball?*, in which the first sentence comes with an implicit (latent) information that Jack *has* or *carries* the ball after *collecting* it. In order to answer the question correctly, apart from the rules about persistence of fluents *be* and *carry*, the system would require rules like:

$$binaryInitEvent(E, be, Obj, Loc) : -binaryEvent(E, be, Per, Loc), \quad (8.1)$$
$$binaryEvent(E, carry, Per, Obj).$$
$$binaryInitEvent(E, carry, Per, Obj) : -binaryEvent(E, collect, Per, Obj) \quad (8.2)$$

that intuitively mean *If a person carries an object, then the object is in the same location as the person* and *If a person collects an object, then the person starts carrying the object*, however the predicate *carry* is implicit in the data. Assuming that we have the following set of modes:

$$modeh(binaryInitEvent(var(e), const(c1), var(p), var(o))). \qquad c1 \in \{be\}$$
$$modeh(binaryInitEvent(var(e), const(c2), var(p), var(l))). \qquad c2 \in \{collect\}$$
$$modeb(1, binaryEvent(var(e), const(c1), var(p), var(l))).$$
$$modeb(1, binaryEvent(var(e), const(c2), var(p), var(o))).$$

in order to be able to learn rules equivalent to 8.1 and 8.2 it is sufficient to add an arbitrary *fresh* constant to the value sets of constant types $c1$ and $c2$, e.g. $c1 \in \{invented1, be\}$ $c_2 \in \{invented1, collect\}$, which allows $invented1$ to function as *carry*. Later, when more examples are given, some of which include a word *carry*, it might be possible to learn a synonymy rules:

$$binaryEvent(E, carry, P, O) : -binaryEvent(E, invented1, P, O)$$
$$binaryEvent(E, invented1, P, O) : -binaryEvent(E, carry, P, O)$$

and hence identify that $invented1$ captures the concept of *carrying*. Once an invented predicate is *tied* to a *real* predicate, the corresponding constant could be replaced with a *lemma*, i.e. $invented1$ replaced with *carry* and the synonymy rules removed.

However, there are practical considerations regarding predicate invention. First is identification of situations when predicate invention is necessary, which could be to some extent addressed by triggering predicate invention whenever the learning task is unsatisfiable. However, unsatisfiability of a learning task can be caused by many factors, such as excessive noise in the training data, in which case predicate invention will not help (or even worse, might cause overfitting) but could significantly affect learning runtime. The second consideration is the number of invented objects that should be added as, unless used sparingly, predicate invention can cause a considerable growth of the size of the hypothesis space.

8.1.2 Automatic Inference of Mode Bias

Currently, a fixed mode bias is added to every learning task and it captures the general constraints on hypothesis space which are rooted in the specifics of the representation used in the project. However, the current approach does not capture constraints specific to a particular text, such as that certain words never form a phrase and hence should not be allowed to co-occur in a hypothesis body.

In order to support contextual mode bias, the current approach to expressing mode bias constraints could be used, however additional constraints should be added by analysing the ASP representations generated from text. Currently, *nominals* and *modifiers* have corresponding mode bias predicates *nominalAllowed* and *modifAllowed* (Section 4.4) which specify very general conditions under which the predicate could be included in a body of a rule in the hypothesis space. The constraints could be made more specific by allowing predicates to occur in the hypothesis body only in the combinations that resemble the ones found in the task context.

Example 8.2 (Inference of mode bias constraints)**.** Let us consider a sentence *Jack performed well but he did not win.* Its ASP representation consists of the following facts:

$$\{unaryEvent(e0, perform, c1)., unaryNominal(c1, jack)., unaryModif(well, e0)., \\ unaryEvent(e1, win, c1)., unaryNominal(c1, he)., unaryModif(not, e1).\}$$

form which we can infer the following conditions under which *modifiers* are allowed to occur in the hypothesis body:

Listing 8.1 Context-dependent mode bias constraints.

```
1: # 'Fixed' mode bias constraint.
2: #bias(":-body(unaryModif(_,V)),not␣modif_allowed(V).").
3:
4: # Inferred mode bias rules.
5: #bias("modif_allowed(V)␣:-␣body(unaryEvent(V,perform,_),
6:         unaryModif(well,V)).").
7: #bias("modif_allowed(V) :- body(unaryEvent(V,win,_),
8:         unaryModif(not,V)).").
```

Being able to infer constraints like the ones presented in Listing 8.1 will be necessary in order to keep the size of the hypothesis space manageable when learning from texts with higher lexical variety.

8.1.3 Enhancing Lexical Knowledge

One frequently quoted disadvantage of symbolic approaches to representation of meaning is their disregard for lexical semantics [17], they leave the meaning of individual words unanalysed and instead are concerned with how meanings compose. Our representation clearly illustrates this problem, for example words *car* and *automobile* have representations $unaryNominal(n1, car)$ and $unaryNominal(n2, automobile)$,[1] which are uninformative with regards to similarity of meaning of the two words. In order to capture such

[1] The names of constants used in this case ($n1$, $n2$) are irrelevant.

similarity the following rule could be added to the background knowledge:

$$unaryNominal(X, W2) : -sense\_of(W1, S), sense\_of(W2, S), \\ unaryNominal(X, W1).$$

Together with facts $sense\_of(car, car)$ and $sense\_of(automobile, car)$. Lexical databases of English, such as WordNet [43], could be used to extract synonymy information, as well as other relations such as antonymy or hypernymy and hence partially mitigate the inadequacies of the symbolic representations of lexical meaning.

References

[1] Yoav Artzi. "Cornell SPF: Cornell semantic parsing framework." In: *arXiv preprint arXiv:1311.3011* (2013).

[2] Yoav Artzi, Nicholas FitzGerald, and Luke Zettlemoyer. *Semantic Parsing with Combinatory Categorial Grammars*. 2013. URL: `http://yoavartzi.com//pub/afz-tutorial.acl.2013.pdf#page=55` (visited on 01/17/2017).

[3] Duangtida Athakravi et al. "Inductive Learning Using Constraint-Driven Bias." In: *Revised Selected Papers of the 24th International Conference on Inductive Logic Programming*. Vol. 9046. ILP 2014. Nancy, France: Springer-Verlag New York, Inc., 2015, pp. 16–32.

[4] Laura Banarescu et al. "Abstract Meaning Representation for Sembanking." In: *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*. Sofia, Bulgaria: Association for Computational Linguistics, 2013, pp. 178–186.

[5] Chita Baral, Juraj Dzifcak, and Tran Cao Son. "Using Answer Set Programming and Lambda Calculus to Characterize Natural Language Sentences with Normatives and Exceptions." In: *Proceedings of the 23rd National Conference on Artificial Intelligence*. Vol. 2. AAAI '08. Chicago, Illinois: AAAI Press, 2008, pp. 818–823.

[6] Chitta Baral and Juraj Dzifcak. "Solving puzzles described in english by automated translation to answer set programming and learning how to do that translation." In: *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press. 2012, pp. 573–577.

[7] Chitta Baral et al. "Using Inverse lambda and Generalization to Translate English to Formal Languages." In: *Proceedings of the Ninth International Conference on Computational Semantics*. IWCS '11. Oxford, United Kingdom: Association for Computational Linguistics, 2011, pp. 35–44.

[8] Johan Bos. "Wide-coverage Semantic Analysis with Boxer." In: *Proceedings of the 2008 Conference on Semantics in Text Processing*. STEP '08. Venice, Italy: Association for Computational Linguistics, 2008, pp. 277–286.

[9] Johan Bos and Katja Markert. "Recognising Textual Entailment with Logical Inference." In: *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*. HLT '05. Vancouver, British Columbia, Canada: Association for Computational Linguistics, 2005, pp. 628–635.

[10] Johan Bos et al. "Wide-coverage Semantic Representations from a CCG Parser." In: *Proceedings of the 20th International Conference on Computational Linguistics*. COLING '04. Geneva, Switzerland: Association for Computational Linguistics, 2004.

[11] Catherine Bosley. *Swiss Eye Scrapping Incentives in Revamped Company Tax Plan*. 2017. URL: https://www.bloomberg.com/news/articles/2017-06-01/swiss-eye-scrapping-some-incentives-in-revamped-company-tax-plan (visited on 06/05/2017).

[12] Chris J.C. Burges. *Towards the Machine Comprehension of Text: An Essay*. Tech. rep. Microsoft Research, 2013. URL: https://www.microsoft.com/en-us/research/publication/towards-the-machine-comprehension-of-text-an-essay/.

[13] Danqi Chen, Jason Bolton, and Christopher D Manning. "A thorough examination of the cnn/daily mail reading comprehension task." In: *arXiv preprint arXiv:1606.02858* (2016).

[14] Stephen Clark. *Practical Linguistically Motivated Parsing with Combinatory Categorial Grammar*. Presentation in JHU Language Technology Summer School. 2009. URL: http://www.cl.cam.ac.uk/teaching/1213/L107/clark_lectures/clark_tutorial.pdf.

[15] Stephen Clark and James R. Curran. "Wide-coverage Efficient Statistical Parsing with CCG and Log-linear Models." In: *Comput. Linguist.* 33.4 (Dec. 2007), pp. 493–552.

[16] Stephen Clark, Julia Hockenmaier, and Mark Steedman. "Building Deep Dependency Structures with a Wide-coverage CCG Parser." In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 327–334.

[17] Stephen Clark and Stephen Pulman. "Combining Symbolic and Distributional Models of Meaning." In: *AAAI Spring Symposium: Quantum Interaction*. 2007, pp. 52–55.

[18] Thomas G Dietterich et al. "Structured machine learning: the next ten years." In: *Machine Learning* 73.1 (2008), p. 3.

[19] Jeffrey Flanigan et al. "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, 2014, pp. 1426–1436.

[20] Jean Gallier. *Phrase-Structure Grammars and Context-Sensitive Grammars*. Lecture Notes: Theory of Computation (CIS 511), Computer and Information Science Department, University of Pennsylvania. 2006. URL: http://www.cis.upenn.edu/~jean/gbooks/cis51104sl13pdf.pdf.

[21] Michael Gelfond and Vladimir Lifschitz. "The stable model semantics for logic programming." In: *ICLP/SLP*. Vol. 88. 1988, pp. 1070–1080.

[22] Mikael Henaff et al. "Tracking the World State with Recurrent Entity Networks." In: *arXiv preprint arXiv:1612.03969* (2016).

[23] Karl Moritz Hermann et al. "Teaching Machines to Read and Comprehend." In: *arXiv preprint arXiv:1506.03340* (2015).

[24] Felix Hill et al. "The Goldilocks Principle: Reading Children's Books with Explicit Memory Representations." In: *arXiv preprint arXiv:1511.02301* (2015).

[25] Julia Hockenmaier and Mark Steedman. "CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank." In: *Comput. Linguist.* 33.3 (Sept. 2007), pp. 355–396.

[26] Julia Hockenmaier and Mark Steedman. *CCGbank: User's Manual*. Tech. rep. Department of Computer and Information Science, University of Pennsylvania, 2005.

[27] Theo M. V. Janssen. "Montague Semantics." In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2016. Metaphysics Research Lab, Stanford University, 2016.

[28] Antonis C Kakas, Robert A Kowalski, and Francesca Toni. "Abductive logic programming." In: *Journal of logic and computation* 2.6 (1992), pp. 719–770.

[29] Tom Kwiatkowski et al. "Scaling semantic parsers with on-the-fly ontology matching." In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. EMNLP 13. Seattle, WA, US: Association for Computational Linguistics, 2013.

[30] Mark Law. *Logic-based Learning in ASP*. Lecture Notes: Logic-based Learning (C304), Department of Computing, Imperial College London. 2016. URL: `https://www.doc.ic.ac.uk/~ml1909/teaching/Unit6.pdf`.

[31] Mark Law, Alessandra Russo, and Krysia Broda. "Inductive learning of answer set programs." In: *European Workshop on Logics in Artificial Intelligence*. Springer. 2014, pp. 311–325.

[32] Mark Law, Alessandra Russo, and Krysia Broda. "Iterative learning of answer set programs from context dependent examples." In: *arXiv preprint arXiv:1608.01946* (2016).

[33] Mark Law, Alessandra Russo, and Krysia Broda. "Learning Weak Constraints in Answer Set Programming." In: *arXiv preprint arXiv:1507.06566* (2015).

[34] Heeyoung Lee et al. "Stanford's Multi-pass Sieve Coreference Resolution System at the CoNLL-2011 Shared Task." In: *Proceedings of the Fifteenth Conference on Computational Natural Language Learning: Shared Task*. CONLL Shared Task '11. Portland, Oregon: Association for Computational Linguistics, 2011, pp. 28–34.

[35] Mike Lewis, Luheng He, and Luke Zettlemoyer. "Joint A* CCG Parsing and Semantic Role Labelling." In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, 2015, pp. 1444–1454.

[36] Mike Lewis, Kenton Lee, and Luke Zettlemoyer. "LSTM CCG Parsing." In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. NAACL '03. San Diego, California: Association for Computational Linguistics, 2016, pp. 221–231.

[37] Mike Lewis and Mark Steedman. "Combining Distributional and Logical Semantics." In: *Transactions of the Association for Computational Linguistics* 1 (2013), pp. 179–192.

[38] Vladimir Lifschitz. "What is Answer Set Programming?" In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*. AAAI '08. Chicago, Illinois: AAAI Press, 2008, pp. 1594–1597.

[39] Christopher D. Manning. "Part-of-Speech Tagging from 97% to 100%: Is It Time for Some Linguistics?" In: *Computational Linguistics and Intelligent Text Processing: 12th International Conference, CICLing 2011, Tokyo, Japan, February 20-26, 2011. Proceedings, Part I*. Ed. by Alexander F. Gelbukh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–189.

[40] Christopher D. Manning et al. "The Stanford CoreNLP Natural Language Processing Toolkit." In: *Association for Computational Linguistics (ACL) System Demonstrations*. 2014, pp. 55–60.

[41] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a Large Annotated Corpus of English: The Penn Treebank." In: *Comput. Linguist.* 19.2 (June 1993), pp. 313–330.

[42] Cynthia Matuszek et al. "Learning to Parse Natural Language Commands to a Robot Control System." In: *Experimental Robotics: The 13th International Symposium on Experimental Robotics*. Ed. by Jaydev P. Desai et al. Heidelberg: Springer International Publishing, 2013, pp. 403–415.

[43] George A. Miller. "WordNet: A Lexical Database for English." In: *Commun. ACM* 38.11 (Nov. 1995), pp. 39–41.

[44] Arindam Mitra and Chitta Baral. "Addressing a Question Answering Challenge by Combining Statistical Methods with Inductive Rule Learning and Reasoning." In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI '16. Phoenix, Arizona: AAAI Press, 2016, pp. 2779–2785.

[45] Richard Montague. "The proper treatment of quantification in ordinary English." In: *Approaches to natural language*. Springer, 1973, pp. 221–242.

[46] Stephen Muggleton. "Inductive Logic Programming." In: *New Gen. Comput.* 8.4 (Feb. 1991), pp. 295–318.

[47] Barbara Partee. "Introduction to Formal Semantics and Compositionality." In: 2006. URL: `http://people.umass.edu/partee/NZ_2006/NZ1.pdf`.

[48] Barbara Partee. *Lambda abstraction, NP semantics, and a Fragment of English*. URL: `http://people.umass.edu/partee/MGU_2005/MGU052.pdf`.

[49] Barbara Partee. "Montague Grammar." In: *International Encyclopedia of the Social & Behavioral Sciences*. Ed. by Neil J. Smelser and Paul B. Baltes. Oxford: Pergamon, 2001, pp. 9995 –9999. URL: `http://people.umass.edu/partee/docs/MontagueGrammarElsevier.PDF`.

[50] Francis Jeffry Pelletier. "The Principle of Semantic Compositionality." In: *Topoi* 13.1 (1994), pp. 11–24.

[51] Hoifung Poon and Pedro Domingos. "Unsupervised Semantic Parsing." In: *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*. Vol. 1. EMNLP '09. Singapore: Association for Computational Linguistics, 2009, pp. 1–10.

[52] Valentina Presutti, Francesco Draicchio, and Aldo Gangemi. "Knowledge Extraction Based on Discourse Representation Theory and Linguistic Frames." In: *Proceedings of the 18th International Conference on Knowledge Engineering and Knowledge Management*. EKAW '12. Galway City, Ireland: Springer-Verlag, 2012, pp. 114–129.

[53] Pranav Rajpurkar et al. "Squad: 100,000+ questions for machine comprehension of text." In: *arXiv preprint arXiv:1606.05250* (2016).

[54] Alessandra Russo. *Introducing Inductive Logic Programming*. Lecture Notes: Logic-based Learning (C304), Department of Computing, Imperial College London. 2016.

[55] Magnus Sahlgren. "The Distributional Hypothesis." In: *Italian Journal of Linguistics* 20.1 (2008), pp. 33–53.

[56] Marek Sergot. *Minimal models and fixpoint semantics for definite logic programs*. Lecture Notes: Knowledge Representation (C491), Department of Computing, Imperial College London. 2005. URL: `https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/Fixpoint_Definite_491-2x1.pdf`.

[57] Jess Shankleman and Alex Morales. *Climate Change*. 2017. URL: `https://www.bloomberg.com/quicktake/climate-change` (visited on 06/05/2017).

[58] Mark Steedman. *A very short introduction to CCG*. 1996. URL: `http://www.inf.ed.ac.uk/teaching/courses/ics/papers/ccgintro.pdf`.

[59] Mark Steedman. *The Syntactic Process*. Cambridge, MA, USA: MIT Press, 2000.

[60] Mark Steedman et al. *Combinatory Categorial Grammars for Robust Natural Language Processing*. 2012. URL: `http://homepages.inf.ed.ac.uk/steedman/papers/ccg/nasslli12.pdf`.

[61] Sainbayar Sukhbaatar et al. "End-To-End Memory Networks." In: *arXiv preprint arXiv:1503.08895* (2015).

[62] Zoltán Gendler Szabó. "Compositionality." In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2013. Metaphysics Research Lab, Stanford University, 2013.

[63] Lappoon R. Tang and Raymond J. Mooney. "Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing." In: *Proceedings of the 12th European Conference on Machine Learning*. EMCL '01. London, UK, UK: Springer-Verlag, 2001, pp. 466–477.

[64] Alicia Beckford Wassink. *Lecture 14 - Syntax*. Lecture Notes: Introduction to Linguistic Thought (LING 200), Department of Linguistics, University of Washington. 2007. URL: `http://faculty.washington.edu/wassink/LING200/lect14_syntax2.pdf`.

[65] Jason Weston, Sumit Chopra, and Antoine Bordes. "Memory networks." In: *arXiv preprint arXiv:1410.3916* (2014).

[66] Jason Weston et al. "Towards ai-complete question answering: A set of prerequisite toy tasks." In: *arXiv preprint arXiv:1502.05698* (2015).

[67] Terry Winograd. "Understanding natural language." In: *Cognitive psychology* 3.1 (1972), pp. 1–191.

[68] W. A. Woods. “Progress in Natural Language Understanding: An Application to Lunar Geology.” In: *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*. AFIPS ’73. New York, New York: ACM, 1973, pp. 441–450.

[69] John M. Zelle and Raymond J. Mooney. “Learning to Parse Database Queries Using Inductive Logic Programming.” In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. Vol. 2. AAAI ’96. Portland, Oregon: AAAI Press, 1996, pp. 1050–1055.

[70] Luke S. Zettlemoyer and Michael Collins. “Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars.” In: *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*. UAI ’05. Edinburgh, Scotland: AUAI Press, 2005, pp. 658–666.

Appendix A

Background Knowledge Used on bAbi Dataset

Listing A.1 Background knowledge rules used on the bAbi dataset.

```
% Background knowledge
% Persistance rules motivated by event calculus.
% Unary.
semUnaryEvent(E,L,Y) :- unaryFluent(E,L,Y).
unaryFluent(E,L,Y) :- unaryInitEvent(E,L,Y).
unaryFluent(E,L,Y) :- unaryFluent(E1,L,Y), previous(E1,E),
        not unaryTermEvent(E,L,Y).

% Binary.
semBinaryEvent(E,L,Y,Z) :- binaryFluent(E,L,Y,Z).
binaryFluent(E,L,Y,Z) :- binaryInitEvent(E,L,Y,Z).
binaryFluent(E,L,Y,Z) :- binaryFluent(E1,L,Y,Z), previous(E1,E),
        not binaryTermEvent(E,L,Y,Z).

% Time points defined by the mata predicates.
previous(E1,E) :- metaData(T1,E1), metaData(T,E), T=T1+1.

% Appendix - mapping to 'semantic' predicates.
semTernaryEvent(E,L,X,Y,Z) :- ternaryEvent(E,L,X,Y,Z),
        not abTernaryEvent(E,L,X,Y,Z).
semBinaryEvent(E,L,Y,Z) :- binaryEvent(E,L,Y,Z),
        not abBinaryEvent(E,L,Y,Z).
semUnaryEvent(E,L,Z) :- unaryEvent(E,L,Z),
        not abUnaryEvent(E,L,Z).

% Equality predicates.
unaryNominal(X,Y) :- eq(X,Z), unaryNominal(Z,Y).
eq(X,Y) :- eq(Y,X).
```

Appendix B

Additional Background Knowledge

Listing B.1 Additional background knowledge rules used for Task 2.

```
synset(go,go).
synset(go,journey).
synset(go,move).
synset(go,travel).

binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,L,V1,V2),
        synset(go,L).
binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,L,V1,V2),
        synset(go,L), unaryNominal(V3,V4).

synset(take,take).
synset(take,get).
synset(take,grab).
synset(take,pick_up).

binaryInitEvent(V0,carry,V1,V2) :- binaryEvent(V0,L,V1,V2),
        synset(take,L).

synset(leave,leave).
synset(leave,discard).
synset(leave,drop).
synset(leave,put_down).

binaryTermEvent(V0,carry,V1,V2) :- binaryEvent(V0,L,V1,V2),
        synset(leave,L).

binaryEvent(E,be,O,L) :- semBinaryEvent(E,carry,P,O),
        semBinaryEvent(E,be,P,L).
binaryInitEvent(E,be,O,L) :- binaryTermEvent(E,carry,P,O),
        semBinaryEvent(E,be,P,L).
binaryTermEvent(E,be,O,L) :- semBinaryEvent(E,carry,P,O),
        binaryTermEvent(E,be,P,L).
```

Listing B.2 Additional background knowledge rules used for Task 5.

```
synset(give,give).
```

```
2: synset(give,hand).
3: synset(give,pass).
4:
5: ternaryEvent(E,V1,X,Y,Z) :- ternaryEvent(E,V2,X,Y,Z),
6:          synset(S,V1), synset(S,V2).
7: binaryEvent(E,V,X,Y) :- ternaryEvent(E,V,X,Y,Z), synset(give,V).
8: binaryEvent(E,receive,R,C) :- ternaryEvent(E,V,G,C,R),
9:          synset(give,V).
```

Listing B.3 Additional background knowledge rules used for Task 16.

```
1: unaryModif(M,Y) :- binaryEvent(E,be,X,C), unaryNominal(C,T),
2:          binaryEvent(E1,be,Y,C2), unaryNominal(C2,T),
3:          unaryModif(M,X).
4:
5: unaryNominal(blue,color).
6: unaryNominal(white,color).
7: unaryNominal(yellow,color).
8: unaryNominal(gray,color).
```

Appendix C

Translation Evaluation

Listing C.1 ASP representation generated for a sentence: *Jack knows a man whose brother won a lottery.*

```
1: binaryEvent(e0,know,c1,n0).
2: unaryNominal(c1,jack).
3: unaryNominal(n0,man).
4: binaryModif(whose,w0,n0).
5: unaryNominal(w0,brother).
6: binaryEvent(e1,win,w0,n1).
7: unaryNominal(n1,lottery).
8: metaData(0,e0).
9: metaData(1,e1).
```

Listing C.2 ASP representation generated for a sentence: *Jack ordered and paid for the dinner.*

```
1: binaryEvent(e0,order,c1,c2).
2: binaryEvent(e1,pay,c1,c2).
3: unaryNominal(c1,jack).
4: unaryNominal(c2,dinner).
5: metaData(1,e1).
6: metaData(0,e0).
```

Listing C.3 ASP representation generated for a sentence: *Jack is a programmer and happy with his job.*

```
1: binaryEvent(e0,be,c1,f0).
2: binaryEvent(e0,be,c1,n0).
3: unaryNominal(c1,jack).
4: unaryNominal(n0,programmer).
5: unaryNominal(f0,job).
6: unaryModif(happy,f0).
7: unaryPrep(he,f0).
8: metaData(0,e0).
```

Listing C.4 ASP representation generated for a sentence: *Jack bought an expensive car.*

```
1: binaryEvent(e0,buy,c1,n0).
```

```
2: unaryNominal(c1,jack).
3: unaryNominal(n0,car).
4: unaryModif(expensive,n0).
5: metaData(0,e0).
```

Listing C.5 ASP representation generated for a sentence: *Jack bought a car that is expensive.*

```
1: binaryEvent(e0,buy,c1,n0).
2: unaryNominal(c1,jack).
3: unaryNominal(n0,car).
4: unaryModif(expensive,n0).
5: metaData(0,e0).
```

Listing C.6 ASP representation generated for a sentence: *An expensive car was bought by Jack.*

```
1: binaryEvent(e0,buy,c1,n0).
2: unaryNominal(c1,jack).
3: unaryNominal(n0,car).
4: unaryModif(expensive,n0).
5: metaData(0,e0).
```

Listing C.7 ASP representation generated for a sentence: *Jack is very stubborn.*

```
1: unaryNominal(c1,jack).
2: unaryModif(m0,stubborn,c1).
3: unaryModif(very,m0).
```

Listing C.8 ASP representation generated for a sentence: *Jack insisted on us staying longer.*

```
1: binaryEvent(e0,insist,c1,i0).
2: unaryModif(longer,e0).
3: unaryNominal(c1,jack).
4: binaryPrep(i0,on,c2,e1).
5: unaryEvent(e1,stay,c2).
6: unaryNominal(c2,we).
7: metaData(0,e0).
8: metaData(1,e1).
```

Listing C.9 ASP representation generated for a sentence: *Jack persuaded Jim to decide in favour of the new agreement.*

```
1: ternaryEvent(e0,persuade,c1,c2,e1).
2: unaryNominal(c1,jack).
3: unaryNominal(c2,jim).
4: binaryEvent(e1,decide,c2,c4).
5: unaryNominal(c4,agreement).
6: unaryNominal(c4,favor).
7: unaryModif(new,c4).
8: metaData(0,e0).
9: metaData(1,e1).
```

Listing C.10 ASP representation generated for a sentence: *Does Jack enjoy playing football?*

```
1: metaData(0,e0).
2: metaData(1,e1).
3: q:-semBinaryEvent(e0,enjoy,c1,e1),semBinaryEvent(e1,play,c1,X0),
4:          unaryNominal(X0,football),unaryNominal(c1,jack).
5: answer(yes):-q.
6: answer(no):-not q.
```

Listing C.11 ASP representation generated for a sentence: *Earl arrived immediately before the person with the Rooster.*

```
1: binaryEvent(e0,arrive,c1,c2).
2: unaryModif(immediately,e0).
3: unaryNominal(c1,earl).
4: unaryNominal(c2,person).
5: binaryPrep(with,c3,c2).
6: unaryNominal(c3,rooster).
7: metaData(0,e0).
```

Listing C.12 ASP representation generated for a sentence: *Jack did not get a haircut at 1.*

```
1: binaryEvent(e0,get,i0,n0).
2: unaryModif(escnot,e0).
3: binaryPrep(i0,at,1,c1).
4: unaryNominal(n0,haircut).
5: unaryNominal(c1,jack).
6: metaData(0,e0).
```

Listing C.13 ASP representation generated for a sentence: *Pete talked about government.*

```
1: binaryEvent(e0,talk,c1,f0).
2: unaryNominal(c1,pete).
3: unaryNominal(f0,government).
4: metaData(0,e0).
```

Listing C.14 ASP representation generated for a sentence: *The candidate surnamed Waring is more popular than the PanGlobal.*

```
1: binaryEvent(e0,surname,c2,c1).
2: unaryNominal(c2,candidate).
3: unaryNominal(c1,waring).
4: ternaryModif(than,c1,m0,c3).
5: unaryModif(m0,popular,c1).
6: unaryModif(more,m0).
7: unaryNominal(c3,panglobal).
8: metaData(0,e0).
```

Listing C.15 ASP representation generated for a sentence: *Miss Hanson is withdrawing more than the customer whose number is 3989.*

```
1: binaryEvent(e0,withdraw,c2,more).
2: unaryModif(miss,c2).
```

```
 3: unaryNominal(c2,hanson).
 4: binaryPrep(i0,than,c4,f0).
 5: unaryModif(more,i0).
 6: unaryNominal(c4,customer).
 7: binaryModif(whose,w0,more).
 8: unaryNominal(w0,number).
 9: binaryEvent(e1,be,w0,3989).
10: metaData(0,e0).
11: metaData(1,e1).
```

Listing C.16 ASP representation generated for a sentence: *Scientists have mostly stopped arguing about whether humans are warming the planet.*

```
 1: binaryEvent(e0,stop,f0,e1).
 2: binaryEvent(e1,argue,f0,e2).
 3: unaryPrep(about,e2).
 4: unaryPrep(whether,e2).
 5: binaryEvent(e2,warm,f1,c3).
 6: unaryNominal(c3,planet).
 7: unaryNominal(f1,human).
 8: unaryNominal(f0,scientist).
 9: unaryModif(mostly,e0).
10: metaData(0,e0).
11: metaData(1,e1).
12: metaData(2,e2).
```

Listing C.17 ASP representation generated for a sentence: *Yet the arguments that crippled the Kyoto Protocol have hardly changed.*

```
 1: unaryModif(yet,e1).
 2: binaryEvent(e0,cripple,c1,c2).
 3: unaryNominal(c1,argument).
 4: unaryNominal(c2,protocol).
 5: unaryModif(kyoto,c2).
 6: unaryEvent(e1,change,c1).
 7: unaryModif(hardly,e1).
 8: metaData(1,e1).
 9: metaData(0,e0).
```

Listing C.18 ASP representation generated for a sentence: *This time, nations made voluntary commitments, with China agreeing that its emissions will peak in about 2030.*

```
 1: unaryModif(this,e0).
 2: unaryNominal(e0,time).
 3: ternaryEvent(e0,make,f0,e1,f2).
 4: unaryNominal(f0,nation).
 5: unaryNominal(f2,commitment).
 6: unaryModif(voluntary,f2).
 7: binaryEvent(e1,agree,c1,e2).
 8: unaryNominal(c1,china).
 9: unaryPrep(that,e2).
10: unaryModif(will,e2).
11: unaryEvent(e2,peak,i0).
12: unaryNominal(f1,emission).
```

```
13: unaryPrep(its,f1).
14: binaryPrep(i0,in,about_2030,f1).
15: metaData(0,e0).
16: metaData(1,e1).
17: metaData(2,e2).
```

Listing C.19 ASP representation generated for a sentence: *The measure was criticized by opponents of the original tax proposal and experts predicted it would be cut.*

```
 1: binaryEvent(e0,criticize,c4,c1).
 2: unaryNominal(c4,opponent).
 3: unaryNominal(c1,measure).
 4: unaryModif(original,c4).
 5: unaryNominal(c4,proposal).
 6: unaryModif(tax,c4).
 7: binaryEvent(e1,predict,f0,e2).
 8: unaryNominal(f0,expert).
 9: unaryModif(would,e2).
10: unaryEvent(e2,cut,c1).
11: unaryNominal(c1,it).
12: metaData(0,e0).
13: metaData(1,e1).
14: metaData(2,e2).
```

Listing C.20 ASP representation generated for a sentence: *Peter Uebelhart, head of tax at KPMG Switzerland, said he did not expect the higher tax on dividends to put off multinationals.*

```
 1: unaryModif(peter_uebelhart,c2).
 2: unaryNominal(c2,head).
 3: unaryNominal(c2,tax).
 4: unaryNominal(c2,kpmg_switzerland).
 5: binaryEvent(e0,say,c2,e1).
 6: unaryModif(escnot,e1).
 7: unaryNominal(c1,he).
 8: binaryEvent(e1,expect,c1,e2).
 9: unaryModif(higher,e2).
10: unaryNominal(e2,tax).
11: unaryNominal(e2,dividend).
12: binaryEvent(e2,put_off,c7,f0).
13: unaryNominal(f0,multinational).
14: metaData(0,e0). metaData(1,e1). metaData(2,e2).
```

Listing C.21 ASP representation generated for Task 2 in STEP 2008: *Cervical cancer is caused by a virus. That has been known for some time and it has led to a vaccine that seems to prevent it. Researchers have been looking for other cancers that may be caused by viruses.*

```
1: % Sentence 1:
2: binaryEvent(e0,cause,n0,f0).
3: unaryNominal(n0,virus).
4: unaryModif(cervical,f0).
5: unaryNominal(f0,cancer).
6: metaData(0,e0).
```

```
 7:
 8: % Sentence 2:
 9: unaryEvent(e1,know,that).
10: binaryPrep(for,f2,that).
11: unaryModif(some,f2).
12: unaryNominal(f2,time).
13: binaryEvent(e2,lead,c1,n1).
14: unaryNominal(c1,it).
15: unaryNominal(n1,vaccine).
16: binaryEvent(e3,seem,n1,e4).
17: binaryEvent(e4,prevent,n1,c1).
18: unaryModif(that,f1).
19: metaData(1,e1).
20: metaData(2,e2).
21: metaData(3,e3).
22: metaData(4,e4).
23:
24: % Sentence 3:
25: binaryEvent(e5,look,f3,f4).
26: unaryNominal(f3,researcher).
27: unaryNominal(f4,cancer).
28: unaryModif(other,f4).
29: unaryModif(may,e6).
30: binaryEvent(e6,cause,f4,f5).
31: unaryNominal(f5,virus).
32: metaData(5,e5).
33: metaData(6,e6).
```

Listing C.22 ASP representation generated for Task 3 in STEP 2008: *John went into a restaurant. There was a table in the corner. The waiter took the order. The atmosphere was warm and friendly. He began to read his book.*

```
 1: % Sentence 1:
 2: binaryEvent(e0,go,c1,n0).
 3: unaryNominal(n0,restaurant).
 4: unaryNominal(c1,john).
 5: metaData(0,e0).
 6:
 7: % Sentence 2:
 8: binaryEvent(e1,be,there,n1).
 9: unaryNominal(n1,table).
10: binaryPrep(in,c4,n1).
11: unaryNominal(c4,corner).
12: metaData(1,e1).
13:
14: % Sentence 3:
15: binaryEvent(e2,take,c5,c6).
16: unaryNominal(c6,order).
17: unaryNominal(c5,waiter).
18: metaData(2,e2).
19:
20: % Sentence 4:
21: unaryModif(friendly,c7).
22: unaryNominal(c7,atmosphere).
```

```
23: unaryModif(warm,c7).
24:
25: % Sentence 5:
26: binaryEvent(e3,begin,c5,e4).
27: unaryNominal(c5,he).
28: binaryEvent(e4,read,c5,f0).
29: unaryPrep(he,f0).
30: unaryNominal(f0,book).
31: metaData(3,e3).
32: metaData(4,e4).
```

Appendix D

Rules Learned on the bAbi Dataset

Listing D.1 Rules learned for Task 1.

```
1: binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,go,V1,V2).
2: binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,journey,V1,V2).
3: binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,move,V1,V2).
4: binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,travel,V1,V2).
5: binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,go,V1,V2),
6:         unaryNominal(V3,V4).
7: binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,journey,V1,V2),
8:         unaryNominal(V3,V4).
9: binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,move,V1,V2),
10:         unaryNominal(V3,V4).
11: binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,travel,V1,V2),
12:         unaryNominal(V3,V4).
```

Listing D.2 Rules learned for Task 6.

```
1: binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,go,V1,V2).
2: binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,journey,V1,V2).
3: binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,move,V1,V2).
4: binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,travel,V1,V2).
5: binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,go,V1,V2),
6:         unaryNominal(V3,V4).
7: binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,journey,V1,V2),
8:         unaryNominal(V3,V4).
9: binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,move,V1,V2),
10:         unaryNominal(V3,V4)
```

Listing D.3 Rules learned for Task 8.

```
1: binaryInitEvent(V0,carry,V1,V2) :- binaryEvent(V0,get,V1,V2).
2: binaryInitEvent(V0,carry,V1,V2) :- binaryEvent(V0,grab,V1,V2).
3: binaryInitEvent(V0,carry,V1,V2) :- binaryEvent(V0,pick_up,V1,V2).
4: binaryInitEvent(V0,carry,V1,V2) :- binaryEvent(V0,take,V1,V2).
5: binaryTermEvent(V0,carry,V1,V2) :- binaryEvent(V0,discard,V1,V2).
6: binaryTermEvent(V0,carry,V1,V2) :- binaryEvent(V0,drop,V1,V2).
7: binaryTermEvent(V0,carry,V1,V2) :- binaryEvent(V0,leave,V1,V2).
8: binaryTermEvent(V0,carry,V1,V2) :- binaryEvent(V0,put_down,V1,V2).
```

Listing D.4 Rules learned for Task 9.

```
binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,go,V1,V2).
binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,journey,V1,V2).
binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,move,V1,V2).
binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,travel,V1,V2).
binaryInitEvent(V0,be,V1,V2) :- not unaryModif(escnot,V0),
        not unaryModif(no,V0), binaryEvent(V0,be,V1,V2).
binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,be,V1,V2),
        unaryNominal(V3,V4).
binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,travel,V1,V2),
        unaryNominal(V3,V4).
```

Listing D.5 Rules learned for Task 12.

```
binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,go,V1,V2).
binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,journey,V1,V2).
binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,move,V1,V2).
binaryInitEvent(V0,be,V1,V2) :- binaryEvent(V0,travel,V1,V2).
binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,go,V1,V2),
        unaryNominal(V3,V4).
binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,journey,V1,V2),
        unaryNominal(V3,V4).
binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,move,V1,V2),
        unaryNominal(V3,V4).
binaryTermEvent(V0,be,V1,V3) :- binaryEvent(V0,travel,V1,V2),
        unaryNominal(V3,V4).
```

Listing D.6 Rules learned for Task 15.

```
eq(V1,V2) :- binaryEvent(V0,be,V1,V2).
```

Listing D.7 Rules learned for Task 18.

```
binaryPrep(V2,in,V0,V4) :- ternaryModif(than,V0,bigger,V1),
        binaryPrep(V2,inside,V3,V4).
binaryPrep(V2,in,V3,V1) :- ternaryModif(than,V0,bigger,V1),
        binaryPrep(V2,inside,V3,V0).
ternaryModif(than,V0,bigger,V2) :- unaryNominal(V2,V3),
        ternaryModif(than,V0,bigger,V1).
ternaryModif(than,V1,bigger,V3) :- binaryPrep(V0,inside,V1,V2),
        unaryNominal(V3,V4).
unaryInitEvent(V0,fit,V1) :- unaryEvent(V0,fit,V1).
unaryModif(bigger,V1) :- binaryPrep(V0,inside,V1,V2).
```