


Università della Calabria

Dipartimento di Matematica e Informatica

Dottorato di Ricerca in Matematica e Informatica

con il contributo del

Programma Operativo Nazionale R&I 2014-2020

XXXVII ciclo

Tesi di Dottorato

NLP based approaches for modeling in ASP

Settore Scientifico Disciplinare INF/01 – INFORMATICA

Coordinatore: Prof. Giorgio Terracina

Giorgio Terracina
13.01.2025
16:10:31
GMT+02:00

Supervisore: Ch.mo Prof. Francesco Ricca

FRANCESCO RICCA
13.01.2025 12:18:20
GMT+02:00

Co-Supervisore: PhD. Manuel Alejandro Borroto Santana

MANUEL ALEJANDRO
BORROTO SANTANA
13.01.2025 12:22:40
GMT+02:00

Dottorando: Dott. Irfan Kareem

IRFAN KAREEM
11.01.2025
11:59:57
GMT+02:00

Firstly, I would like to extend my heartfelt gratitude to my PhD supervisor, Professor Francesco Ricca, and Dr. Manuel Borroto Santana, for their unwavering support, encouragement, and dedication throughout my PhD journey. Their continuous enthusiasm and commitment, even in reviewing many drafts of this thesis, have been invaluable. I am also deeply grateful to Professor Alessandra Russo from the Department of Computing at Imperial College London for her support and guidance. I also appreciate and acknowledge Ms. Katie Gallagher from the SPIKE group at Imperial for her work on the second part of the thesis. Additionally, I would like to extend my thanks to Antonio Ielo, Giuseppe Mazzotta, and Carmine Dodaro for their steadfast support.

I would also like to express my deep gratitude to my late grandmother and grandfather, whose support and encouragement laid the foundation for my education from a young age. Their belief in my potential continues to inspire me. I am equally thankful to my uncles, especially Dr. Ameer and Maher Nasir, for their encouragement and guidance.

I am immensely grateful to my parents and family, who have always emphasized the importance of education and supported me across continents to pursue my academic goals. Finally, a heartfelt thank you to my friends for their constant encouragement and support throughout this journey.

Contents

Sommario

Gli approcci di elaborazione del linguaggio naturale (NLP) hanno modificato il modo in cui interagiamo con i computer, semplificando una serie di attività che richiedono molto tempo e un certo livello di abilità negli utenti. Un esempio piuttosto noto è la composizione di programmi per computer con strumenti come GitHub Copilot. Tuttavia, gli strumenti esistenti per tale attività si concentrano sui linguaggi di programmazione tradizionali e viene fornito un supporto limitato (o nullo) per la rappresentazione della conoscenza consolidata e il formalismo del ragionamento, come Answer Set Programming (ASP). ASP è un ramo della programmazione logica che offre un approccio dichiarativo per modellare i problemi con applicazioni sia in ambito accademico che industriale. ASP è considerato impegnativo perché si basa sulla rappresentazione di regole logiche, che può essere impegnativa per gli utenti alle prime armi e rimane complicata anche per gli esperti.

Un altro problema impegnativo nell'intelligenza artificiale (IA) è la capacità di dotare le macchine di ragionamento di buon senso e capacità di apprendimento. Nelle attività di domande e risposte (Q&A), i modelli di intelligenza artificiale sono addestrati a rispondere alle domande imparando correlazioni e caratteristiche statistiche e non hanno la capacità di apprendere conoscenze di buon senso ed eseguire ragionamenti di buon senso generali. Nonostante i recenti progressi negli approcci di intelligenza artificiale, come i modelli di linguaggio di grandi dimensioni (LLM), hanno avuto prestazioni inferiori nei benchmark di ragionamento in linguaggio naturale. Gli LLM non hanno la capacità di ragionamento di buon senso e apprendimento dal testo. D'altro canto, gli LLM hanno mostrato un grande potenziale nell'elaborazione e nella generazione di testo. I recenti approcci neurosimbolici hanno iniziato a combinare gli LLM con metodi di ragionamento formale per affrontare le limitazioni degli LLM, ma la conoscenza simbolica del componente di ragionamento è progettata manualmente, il che richiede un

notevole sforzo. Un aspetto impegnativo della programmazione logica induttiva (ILP) riguarda l'apprendimento dal set di risposte e la composizione del programma ASP.

Considerando queste sfide, in questa tesi proponiamo due contributi principali: (i) un primo passo verso la composizione automatica di programmi ASP da specifiche di linguaggio naturale; e (ii) l'apprendimento di conoscenze di buon senso per domande robuste e la risposta con LLM.

Il primo contributo introdotto, il primo passo verso l'automazione della composizione delle specifiche di Answer Set Programming (ASP). In particolare, vengono forniti i seguenti contributi: (i) Un set di dati incentrato sulle specifiche dei problemi correlati ai grafici, progettato per sviluppare e valutare strumenti per la codifica automatica ASP; (ii) Un'architettura in due fasi, implementata nello strumento NL2ASP, per generare programmi ASP da specifiche di linguaggio naturale. NL2ASP utilizza la traduzione automatica neurale per trasformare il linguaggio naturale in istruzioni di linguaggio naturale controllato (CNL). Successivamente, le istruzioni CNL vengono convertite in codice ASP utilizzando lo strumento CNL2ASP. Un esperimento conferma la fattibilità dell'approccio.

Nel secondo contributo presentato, un sistema chiamato LLM2LAS, per apprendere la conoscenza di buonsenso da domande e risposte basate su storie espresse in linguaggio naturale. LLM2LAS combina la capacità di analisi semantica degli LLM con ILASP per apprendere conoscenze di buonsenso espresse come programmi di set di risposte. LLM2LAS richiede solo pochi esempi di domande e risposte per apprendere conoscenze di buonsenso generali e rispondere correttamente a domande invisibili. Una valutazione empirica dimostra la fattibilità del nostro approccio.

Abstract

Natural language processing (NLP) approaches have been modifying the way we interact with computers, by simplifying a number of tasks that are time-consuming and require quite some level of ability in the users. A rather well-known example is the composition of computer programs with tools such as GitHub Copilot. Nonetheless, existing tools for that task focus on mainstream programming languages, and limited (or no support) is provided for well-established knowledge representation and reasoning formalism, such as Answer Set Programming (ASP). ASP is a branch of logic programming that offers a declarative approach to model problems with applications both in academia and industry. ASP is considered challenging because it relies on logical rule representation, which can be challenging for novice users and remains complicated even for experts.

Another challenging problem in Artificial Intelligence (AI) is the ability to endow machines with commonsense reasoning and learning capabilities. In Question and Answering (Q&A) tasks, AI models are trained to answer questions by learning correlations and statistical features and lack the ability to learn commonsense knowledge and perform general commonsense reasoning. Despite recent advances in AI approaches such as Large Language Models (LLMs) have underperformed in natural language reasoning benchmarks. LLMs lack the ability for commonsense reasoning and learning from text. On the other end, LLMs have shown great potential in processing and generating text. Recent neurosymbolic approaches have started combining LLMs with formal reasoning methods to address the limitations of LLMs but symbolic knowledge of the reasoning component is manually engineered, which requires quite some effort. A challenging aspect of Inductive Logic Programming (ILP) involves learning from the answer set as well as composing the ASP program.

Considering these challenges, in this thesis we propose two main contribu-

tions: (i) a first step towards automatic composition of ASP programs from natural language specifications; and (ii) learning commonsense knowledge for robust questions and answering with LLM.

The first contribution introduced is a first step towards automating the composition of answer set programming specifications. In particular, the following contributions are provided: (i) a dataset focused on graph-related problem specifications, designed to develop and assess tools for ASP automatic coding; (ii) a two-step architecture, implemented in the NL2ASP tool, for generating ASP programs from natural language specifications. NL2ASP uses neural machine translation to transform natural language into Controlled Natural Language (CNL) statements. Subsequently, CNL statements are converted into ASP code using the CNL2ASP tool. An experiment confirms the viability of the approach.

In the second contribution a system called LLM2LAS, for learning commonsense knowledge from story based question and answering expressed in natural language is presented. LLM2LAS combines the semantic parsing capability of LLMs with ILASP to learn commonsense knowledge expressed as answer set programs. LLM2LAS requires only a few examples of questions and answers to learn general commonsense knowledge and correctly answer unseen questions. An empirical evaluation demonstrates the viability of our approach.

Chapter 1

Introduction

Natural language processing methodologies [51] have determined impressive changes in the way we engage with computers, streamlining a multitude of tasks that would otherwise demand considerable time and proficiency from users. This thesis offers two contributions within this context, each addressing key tasks in natural language manipulation and understanding: the automated composition of logic programs and the automatic acquisition of commonsense knowledge for robust question answering.

Automated composition of logic programs A noteworthy example of automatic composition of computer programs is the emergence of tools for the automatic composition of computer programs [28, 78], like GitHub Copilot [20, 53].

Nevertheless, prevailing tools of this nature predominantly cater to widely used programming languages, and limited (or no) support is available for many problem-solving formalisms in the area of Knowledge Representation and Reasoning (KR&R). An example of a formalism for KR&R that misses automatic program composition tools is answer set programming [8, 38]. ASP is a declarative programming paradigm that can be used to solve complex AI problems. It originates from research on logic programming, nonmonotonic reasoning and knowledge representation. It became popular for featuring both an expressive declarative language and some efficient implementations [36], such as Clingo [34] and DLV [2]. It has been applied both in academia and industry, and it was effective in several knowledge-intensive applications of AI, such as scheduling, product configuration, robotics, workforce management, and decision support [26, 36].

In the last few years, quite some effort has been spent on providing programming environments and tools for assisting programmers in devising ASP specifications [98, 3, 44, 31, 10], including advanced editors, debuggers, testing frameworks, visualization tools, etc. Nonetheless, coding in ASP still is very challenging for beginners, especially for those having a weak mathematical and logical background, and is a time-consuming (sometimes repetitive) task also for expert programmers. The most recent tool for translating controlled natural language to ASP the CNL2ASP [12], is not effortless. Users must respect the grammar for each rule definition and follow the top-to-bottom dependency structure. On the other hand, once we look for the latest tools that significantly boosted the productivity of developers using mainstream imperative programming languages, it is easy to observe that a great benefit has been brought by the introduction of automatic program composition tools, such as GitHub Copilot. However, no programming environment for ASP (at the time of the start of this project) features automatic code composition from the user's requests in natural language. Streamlining KR formalism with an automatic tool shows promise in boosting productivity for experts and easing entry for newcomers, potentially reducing adoption barriers, even in industrial settings. One might observe that generative AI tools, like ChatGPT, are capable of generating some ASP code snippets from specific prompts. Actually, they have been proficiently used to produce factual statements in an ASP based system for reasoning from text [106]; but, it can be easily verified that these AI models are not robust in generating valid and correct ASP programs. Thus, the problem of (robustly) composing ASP programs from statements provided in natural language remains open.

In this thesis, the first steps towards filling this gap are taken, and the following contributions are provided:

(i) A dataset focused on graph-related problem specifications, that is conceived to develop and assess NLP tools for the automatic composition of ASP programs;

(ii) a two-step architecture, implemented in the NL2ASP tool, aiming at automating the translation of natural language specifications into ASP programs;

(iii) an experiment providing empirical evidence that the dataset and the architecture are a good first step towards automated code generation for ASP.

NL2ASP processes the input in two distinct phases: The first phase draws inspiration from Neural Machine Translation (NMT). NMT [105] methods utilize deep learning algorithms to translate text from one language to another, providing more natural and accurate results compared to traditional machine translation methods [91]. NL2ASP employs NMT techniques to translate the input sentences in natural language to statements conforming to a recently-introduced controlled natural language for ASP [12]. For this task, NL2ASP can resort either to BART-base [62] or T5-small [83]. In the second phase, the controlled natural language statements are converted into ASP code using the CNL2ASP tool [12]. The performance of NL2ASP is measured in an experiment in terms of the quality of the statements produced in each of the steps of the composition process. In particular, the CNL statements produced by NL2ASP are evaluated in terms of Bilingual Evaluation Understudy (BLEU) [77], MEtric for TExt Ordered Reference (METEOR) [5] and Syntax Correctness Accuracy (a measure of the capability of producing syntactically valid CNL statements). Eventually, the overall behaviour of the system is assessed by implementing an end-to-end test in which the capability of producing correct ASP specifications is measured. The obtained results are very promising and confirm the viability of the approach.

Automatic acquisition of commonsense knowledge for robust question answering A challenging problem in AI is the ability to endow machines with commonsense reasoning and learning capabilities [21]. In machine comprehension and question and answering tasks, AI models, trained to answer questions about given texts, may do so through learning correlations and statistical features in the text rather than learning commonsense knowledge and/or performing general commonsense reasoning [1]. Despite recent successes, AI approaches such as LLMs, have also shown to fall short in providing truthful answers [109], and to under perform on natural language reasoning benchmarks [100, 59]. Their lack of transparency and explainability make it difficult to ascertain whether these models genuinely learn commonsense reasoning [86]. On the other end, LLMs have shown great potential in processing and generating text. In particular, they have shown to be effective in performing semantic parsing of natural language, the task of mapping natural language utterances into formal meaning representations [23]. Recent neurosymbolic approaches have started to show that by combining LLMs and

formal reasoning methods it is possible to address some of the limitations of LLMs whilst exploiting their generative capability. LLMs coherence and consistency in story completion tasks can be improved by combining LLM based semantic parsing, to map text into formal representations, with symbolic reasoner to determine which of the LLMs generated completion sentences are correct [75]. Ishay, Yang and Lee [50] combine LLMs with ASP [66] to solve logic puzzles. Yang, Ishay and Lee [106] combines GPT3-based semantic parsing with an ASP knowledge module to perform reasoning, showing state-of-the-art performance on several benchmarks. Although these approaches result in more robust reasoning from text, the symbolic knowledge of the reasoning component is manually engineered, which is time-consuming and requires effort from the user.

In this thesis, we propose a novel system, called LLM2LAS, that learns common-sense knowledge for machine comprehension. LLM2LAS combines an LLM-based semantic parser for generating symbolic representations of a story and questions, with ILASP, a tool for learning common-sense knowledge under the answer set semantics. Our approach automatically generates a ILASP learning task from a given story, questions and answers, written in natural language, and iteratively learns the common-sense knowledge needed for solving the given Q&A task. The general learned knowledge can then be used to answer questions about unseen stories.

The key components of LLM2LAS include:

(i) open-source LLM-based few-shot semantic parser for generating ASP representations from natural language;

(ii) a learning module, based on ILASP, for learning common-sense knowledge needed to answer questions about given stories, and

(iii) a reasoning module for answering questions about a story using the learned common-sense knowledge.

We have evaluated our approach on the Q&A bAbI dataset [80], reporting 100% accuracy on 13 of the 20 bAbI tasks.

Published papers The contributions described in the thesis have been published in prestigious AI conferences. The first contribution was published at the International Joint Conference on Artificial Intelligence (IJCAI) 2024

held in Jeju South Korea. The second contribution was accepted at the International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR) 2024, Dallas United States of America.

- Manuel Borroto Santana, Irfan Kareem, and Francesco Ricca. "Towards Automatic Composition of ASP Programs from Natural Language Specifications." In IJCAI-24. Main Track. Pages 6198-6206. https://doi.org/10.24963/ijcai.2024/685

- Irfan Kareem, Katie Gallagher, Manuel Borroto, Francesco Ricca, and Alessandra Russo. "Using Learning from Answer Sets for Robust Question Answering with LLM." In Logic Programming and Nonmonotonic Reasoning. LPNMR 2024.LNCS, vol 15245. Springer, Cham. https://doi.org/10.1007/978-3-031-74209-5_9

Thesis structure The remaining part of the thesis is divided into two parts, Part I and Part II, with the preliminaries of both parts, are discussed in Chapter 2. The Part I NL2ASP, focuses on the composition of ASP programs from natural language specifications, divided into chapters: Chapter 3 discusses the problem, describes dataset construction workflow, and presents an architecture overview of the natural language to ASP tool, and Chapter 4 provides details of the experiments conducted to validate the proposed pipeline. The Part II presents a novel system named LLM2LAS that learns commonsense knowledge for machine comprehension, using large language models and inductive logic programming for robust Question & Answering. Chapter 5 discusses the system problem and the proposed methodology. Chapter 6 presents the empirical evaluation, including results and discussion. Chapter 7 reviews related work. Finally, Chapter 8 presents the conclusion and future work for both parts.

Chapter 2

Preliminaries

This chapter provides an overview of the main concepts needed to understand the work in this thesis. It starts with an overview of logic programming, leading to answer set programming, including basic syntax and semantics of ASP rules. Then, we discuss the concepts of controlled natural language used for ASP, with a brief overview of the CNL2ASP tool. Additionally, discusses the concepts related to Inductive Learning (IL) in ASP using Inductive Learning of Answer Set Programs (ILASP), and we conclude with a detailed overview of neural machine translation.

2.1 Logic Programming

Logic offers a formal and rigorous way to express specific scenarios and general knowledge to compute logical consequences. A logical system consists of three things, a formal language to express the concepts for example propositions $p, q, r, s, ..$, and logical connectives e.g., $\wedge, \vee, \neg, \iff, \rightarrow$ to represent the syntax. The semantics provide the meaning for formal language, the interpretation of the assignment of each proposition.

A logic program consists of a set of rules, it has two parts head and body shown in the following rule.

$$Rule: \quad \underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \ldots, b_m}_{\text{body}} \tag{2.1}$$

The informal semantics of the rule is 'if $b_1, \ldots, b_m$ is true, then a must be true'. Different systems implement traditional logic programming lan-

guages [56], such as Programming in logic (Prolog) [92]. However, these languages face limitations in terms of expressiveness, particularly when defining search strategies, and they are not entirely declarative [67]. Answer set Programming (ASP), on the other hand, offers a fully declarative approach to logic programming [37]. In the following section, we delve into ASP, discussing its language features and underlying semantics in detail.

2.1.1 Answer Set Programming (ASP)

ASP [8, 40] is a declarative approach to problem-solving [35]. Unlike traditional programming languages, where a programmer specifies how to solve a problem step-by-step, ASP focuses on describing 'what the problem is'. The system then generates a solution based on this description. ASP has roots in databases, non-monotonic reasoning, logic programming, satisfiability solving, knowledge representation and reasoning and stable model semantics. ASP is good for solving knowledge-intense combinatorial optimization problems i.e. problems of many decisions and constraints (Soduku, configuration, planning, timetabling etc.), and application in Artificial Intelligence, Knowledge Representation, Robotics, Bioinformatics and industry [26, 29].

ASP supports both qualitative (logical preferences) and quantitative (cost functions) optimization, allowing it to select the most suitable solution based on various factors. Qualitative criteria prioritize solutions that better satisfy certain conditions or constraints more effectively than others, while quantitative criteria involve optimizing numerical values like minimizing costs, maximizing benefits, or balancing trade-offs. The ASP includes several logic-based problem solvers [65], including Clingo, DLV, and Wasp to find solutions [67]. The semantics of ASP programs are in terms of stable models (or answer sets) [39].

The source problem is modeled into a formal representation and then solved by exploring advanced search algorithms that extract the implicit state space to find a solution. The key idea of ASP is to represent the source problem in a logical format and the solutions to the problem correspond to the models of this representation, which are referred to 'answer sets'. In the following, we overview the language of ASP with some rules description.

2.1.2 The Language of ASP

The main construct of the ASP language is the logic rule, which is a construct of the form:

$$a_1 \mid \ldots \mid a_n \ :- \ b_1, \ldots, b_k \ ; \ not \ b_{k+1}, \ldots, \ not \ b_m \tag{2.2}$$

where a_i $(0 \leq i \leq n)$ are *atoms*, b_i $(0 \leq i \leq k \leq m)$ and $not\ b_j$ $(k \leq j \leq m)$ are positive and negative *literals*, respectively. Informally, a rule reads as follows: "at least one of the a_i is true whenever all b_i $(0 \leq i \leq k \leq m)$ are true and all b_j $(k \leq j \leq m)$ are false". On the right-hand side of a rule is the *body* (or antecedent) and on the the left-hand side of a rule is the *head* (or consequent). The rule is called *Fact* if the body of a rule is empty. A rule with an empty head is called *Constraint*. An ASP program is a set of logic rules, which is interpreted according to common sense principles. An ASP program represents a problem to be solved that together with input (also expressed by a collection of rules) admits a collection of answers (possibly also no answer) corresponding to the solutions of the problem [67]. For example, the well-known 3-colorability problem is solved by:

$$col(X, red) \mid col(X, green) \mid col(X, yellow) \ :- \ node(X)$$
$$:- col(X, C), col(Y, C), edge(X, Y)$$

The first rule reads "if X is a node then it can be colored either in red or in green or in yellow", and the second rule (a constraint) reads "it is not possible that two adjacent nodes have the same color". Here, the input is modelled by a set of facts of the form $node(\cdot)$ and $edge(\cdot, \cdot)$. Suppose we provide as input the graph $node(1)$, $node(2)$, $node(3)$, $edge(1, 2)$ and $edge(1, 3)$; it can be verified that a possible solution (answer set) is $\{col(1, red), col(2, green), col(3, yellow)\}$. The 'color' is abbreviated as 'col'.

Weak Constraints

ASP features *weak* constraints to model optimization problems. *weak* constraints are rules of the form:

$$:\sim \ b_1, \ \ldots, \ b_k, \ not \ b_{k+1}, \ \ldots, \ not \ b_m. \ [w@l, \overline{t}] \tag{2.3}$$

Here $[w@l]$ means weight w at level l, where weight is the *"cost"* of violating the weak constraint, and *levels* define priority among preference

criteria. Intuitively, the above statement means: "each time we violate the condition in the body we pay a cost of w at preference level l. In this example of minimizing the edge conflicts,

$$:\sim \text{col}(X, C),\ \text{col}(Y, C),\ \text{edge}(X, Y).\ [1@1]$$

The weak constraint penalizes each case where two adjacent nodes X and Y share the same color C, assigning a cost of 1 at priority level 1 for each such conflict.

Choice Rules

In addition to standard rules, ASP supports *choice rules*. A choice rule allows one to specify that any subset of a set of atoms can be true. The head of a choice rule includes expression in braces { }. The general form of a choice rule is:

$$\{a_1, a_2,\ \ldots,\ a_n\}\Theta t\ :- b_1, \ldots, b_k \tag{2.4}$$

where a_i $(1 \leq i \leq n)$ are *atoms*, and b_i $(1 \leq i \leq k)$ are positive *literals*, $\Theta \in <, >, =$ is a comparison operator and t an integer term (Θt is optional). This rule allows for any combination of the literals in $\{a_1, a_2, \ldots, a_n\}$ (of cardinality Θt) to be included in a stable model, as long as the conditions specified in the body of the rule are satisfied. Additionally, not $b_{k+1}, \ldots,$ not b_m are negative literals and must not be included in the model. Here is an example of N-Coloring, where it is required to choose one color for each node.

$$\{color(X; C) : color(C)\} = 1\ :- node(X).$$

Here, the expression $\{color(X; C) : color(C)\}$ specifies the choice of colors for node X, and $= 1$ restricts the rule to selecting exactly one color for each node.

Aggregate Functions

ASP also supports *aggregates*, which allow for more complex conditions over collections of literals. An aggregate is an expression of the form:

$$L_g < f\{S\} < U_g \tag{2.5}$$

Here, L_g and U_g are the lower and upper bounds, respectively.

The $f(S)$ is an *aggregate function*, where S is a symbolic set, and $f \in \{\#count, \#sum\}$ is an *aggregate function symbol*.

A *symbolic set* is a set specified syntactically as $\{Vars : Conj\}$, where *Vars* is a non-empty list of variables, and *Conj* is a non-empty conjunction of standard literals.

$$L_g < f\{Vars : Conj\} < U_g \tag{2.6}$$

An *aggregate atom* is of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq, =\}$ is a comparison operator, and T is a term called guard. An aggregate atom $f(S) \prec T$ is ground if T is a constant and S is a ground set. A *ground set* is a set of pairs of the form $\langle consts : conj \rangle$, where *consts* is a list of constants and *conj* is a conjunction of ground standard literals. Consider the following example of aggregate expression:

$$5 < \#count\{EmpId : emp(EmpId, male, Skill, Salary)\} \leq 10$$

If the number of male employees is greater than 5 and does not exceed 10, then the atom is true. For a more detailed (and formal) account of ASP please refer to dedicated literature [11, 8, 40, 67, 35].

2.2 Controlled Natural Language for ASP

ASP is widely used for its simple syntax and semantics and the availability of efficient systems. However, the usage of ASP is still problematic and requires a good mathematical or logical background. In this regard, different systems have been proposed to ease this problem, such as introducing controlled natural language to compose ASP [12, 88, 27].

2.2.1 Controlled Natural Languages (CNLs)

Controlled natural languages are subsets of natural languages with restricted grammar and vocabulary [57]. Traditionally, CNLs have different names such as simplified, technical, structured, processable or controlled [58]. Some examples of controlled languages are Basic English, Caterpillar Fundamental English, Semantics of Business Vocabulary and Business Rules (SBVR) Structured English, and Attempto Controlled English.

The wide variety of such languages is designed to improve communication between humans, for example, they improve clarity in technical documentation, particularly for non-native speakers [89]. In machine translation, CNLs provide structured input which helps to enhance the quality of translation [89]. Additionally, CNLs are effective in Knowledge Representation for logical reasoning and problem-solving in semantic systems [33]. The CNLs are categorized by their design and application, both in natural and formal languages [58].

A number of studies (see [Related Work Section 7]) have exploited controlled natural language or mathematical definitions to derive ASP programs, thereby facilitating the conversion of natural language into ASP code.

2.2.2 CNL2ASP Tool

A comprehensive CNL tool for ASP that supports all the main language constructs is CNL2ASP [12]. In CNL2ASP, CNL specifications are made up of propositions that are structured by means of clauses, connected by connectives to express concepts and conditions. The CNL grammar by Dodaro et al. [12] includes several proposition templates, namely: definitions, whenever/then clauses, fact propositions, negative and positive constraints, weak constraints, and quantified choice. Definition propositions are useful to define

the domain of discourse, whereas the remaining statements are respectively translated into: rules, facts, constraints, weak constraints, and choice rules.

To provide flavour and a basic understanding of CNL2ASP, we resort to our running example, e.g., an encoding of the 3-colorability problem. First of all, we define node, edge and color by using the following definition statements:

```
A node is identified by an id.
A edge is identified by a firstnode, and by a secondnode.
A color is identified by an id.
```

These are used by CNL2ASP to initialize the internal data structures and know about the domain of discourse. Then, we specify the assignment of colors to nodes by using the following `whenever/then` clause:

```
Whenever there is a node with id X then we can have a col with node
X, and with color equal to blue, or a col with node X, and with color
equal to red, or a col with node X, and with color equal to green.
```

Finally, we specify valid colorings with the following *negative strong constraint* statement:

```
It is prohibited that C1 is equal to C2, whenever there is a col with
node X, and with color C1, whenever there is a col with node Y, and with
color C2, whenever there is an edge with firstnode X, and with secondnode
Y.
```

The ASP program resulting from a call to CNL2ASP is:

$$col(X, blue) \mid col(X, red) \mid col(X, green) \ :- \ node(X).$$

$$:- C1 = C2, \ col(X, C1), \ col(Y, C2), \ edge(X, Y).$$

It is worth observing that the CNL specification is amenable to a human and does not require to know the technicalities of the syntax of ASP. The interested reader can find more details CNL2ASP in the original work [12].

2.3 Inductive Learning of Answer Set Programs (ILASP)

Inductive logic programming is a logic-based machine learning method that uses logic programs or rules from training examples and background knowledge, enabling generalization and reasoning capabilities [19]. The output of ILP algorithms is logic programs under given semantics. The expressivity of logic programs, combined with the integration of background knowledge, enables ILP to enhance its reasoning and learning capabilities, particularly in scenarios where machine learning struggles. The strength of ILP lies in its ability to leverage human-like reasoning and generalize from limited examples. ILASP is a recently developed state-of-the-art system for learning ASP programs from a set of examples [61]. It has been used to learn commonsense knowledge such as choices, defaults, exceptions, and preferences. A brief recap of the ASP syntax is discussed in relevant in section 2.1.1, and also referring the reader to [39, 8, 11] for a formal account on ASP syntax and semantics.

2.3.1 Learning from Answer Sets

In this thesis, we use the *Learning from Answer Sets* (LAS) framework and its state-of-the-art system ILASP [60] for learning ASP programs. The LAS framework solves *learning tasks* which consist of a *background knowledge*, the *mode bias* and a set of *examples*. The background knowledge, denoted as B, is an ASP program which describes a set of concepts that are known before learning. The mode bias, denoted as M and often called *language bias*, is used to express the ASP programs that can be learned. A *mode bias* is defined as a pair of sets of mode declarations $M = \langle M_h, M_b \rangle$, where M_h (resp. M_b) are called the *head* (resp. *body*) *mode declarations*. Each mode declaration is a literal whose abstracted arguments are either `var(t)` or `const(t)`, for some constant `t` (called a *type*). For each type, a set of constants is provided along with the maximum number of variables that a rule can take, thus constraining the search space induced M. Informally, a literal is *compatible* with a mode declaration m if it can be constructed by replacing every instance of `var(t)` in m with a variable of type `t`, and every `const(t)` with a constant of type `t`.

Definition 1. Given a mode bias $M = \langle M_h, M_b \rangle$, a normal rule R is in the hypothesis space S_M if and only if (i) the head of R is compatible with a mode declaration in M_h; (ii) each body literal of R is compatible with a mode declaration in M_b; and (iii) no variable occurs with two different types.

The set of examples, denoted as E, describes a set of semantic properties that the learned ASP program should satisfy. They are defined in terms of *partial interpretations*. A partial interpretation is a pair of sets of ground atoms $\langle e^{inc}, e^{exc} \rangle$, called respectively inclusion and exclusion sets. An interpretation I *extends* e iff $e^{inc} \subseteq I$ and $e^{exc} \cap I = \emptyset$. A ILASP example $e \in E$ is a *context dependent partial interpretation* (CDPI). This is a tuple $e = \langle e_{id}, e_{pi}, e_{ctx} \rangle$, where e_{id} is an identifier for e, e_{pi} is a partial interpretation and e_{ctx} is an ASP program called a *context*. A CDPI e is *accepted* by a program P if and only if there is an answer set of $P \cup e_{ctx}$ that extends e_{pi}. The idea of a context-dependent example is that each context only applies to a particular example. This is suitable for question-answering tasks where answer to a question is normally contextualised with respect to the story, or text provided to the learner. Formally, an ILASP *context-dependent learning task* is defined as follows.

Definition 2. A *Context-dependent Learning task* ($ILP^{context}_{LAS}$) is a tuple $T = \langle B, S_M, E \rangle$ where B is an ASP program, called the background knowledge, S_M is the set of rules allowed in the hypotheses (the hypothesis space), E is a set of CDPIs. A hypothesis H is an inductive solution of T (written $H \in ILP^{context}_{LAS}(T)$) if and only if:

1. $H \subseteq S_M$;
2. $\forall \langle e_{id}, e_{pi}, e_{ctx} \rangle \in E$, $\exists A \in AS(B \cup e_{ctx} \cup H)$ such that A extends e_{pi}.

A learning task may have multiple inductive solutions. These are scored in terms of their length (i.e., number of literals they include), $score(H, T) = |H|$. An inductive solution $H \in ILP^{context}_{LAS}(T)$ is *optimal* if there is no other inductive solution $H' \in ILP^{context}_{LAS}(T)$ such that $score(H', T) < score(H, T)$.

2.3.2 Event Calculus

Some fundamental concepts of Event Calculus (EC) are now introduced to facilitate the understanding of the application discussed in the second part of this thesis. The EC [55] is a known formalism introduced to represent

and reasoning about events. It is a normal logic program that typically includes domain-independent rules describing general principles for inferring when properties (called *fluents*) are true at particular time-points, denoted as $\texttt{holdsAT}(\texttt{F},\texttt{T})$, based on which events have previously occurred (denoted as *happens(E,T)*). In addition, an EC representation includes a collection of *domain-dependent* rules, describing the effect of events (using the predicates $\texttt{initiatedAt}(\texttt{F},\texttt{T})$ and $\texttt{terminatedAt}(\texttt{F},\texttt{T})$) as well as the time point at which events occur (using the predicate $\texttt{happensAt}(\texttt{E},\texttt{T})$).

In this thesis, we will make use of the following ASP encoding of domain-independent rules for EC:

$$\begin{aligned}&\texttt{holdsAt}(\texttt{F},\texttt{T}+1)\texttt{:-}\ \texttt{initiatedAt}(\texttt{F},\texttt{T}),\texttt{time}(\texttt{T}).\\&\texttt{holdsAt}(\texttt{F},\texttt{T}+1)\texttt{:-}\ \texttt{holdsAt}(\texttt{F},\texttt{T}),\texttt{not terminatedAt}(\texttt{F},\texttt{T}),\texttt{time}(\texttt{T}).\end{aligned} \tag{2.7}$$

where $\texttt{initiatedAt}(\texttt{F},\texttt{T})$ (resp.$\texttt{terminatedAt}(\texttt{F},\texttt{T})$) indicates the time T when a fluent F is made true (resp.false) by an event occurring. In learning common-sense knowledge we may observe fluents at particular time points, encoded using `holdsAt` predicates with the objective of learning the definition of `initiatedAt` and `terminatedAt`, that is how events cause changes in the truth value of fluents.

2.4 Machine Translation (MT)

A MT is a subfield of natural language processing that automates the translation of written text or speech from one language to another. Historically, MT has three primary developments as Rule-based Mchine Translation (RBNMT) [32], Statistical Machine Translation (SMT) [54], and Neural Machine Translation (NMT) [16]. Here we have provided a brief overview of these approaches.

2.4.1 Rule-based Machine Translation

A rule-based machine translation uses linguistic rules and dictionaries to perform translations [32]. The predefined grammatical rules and lexicons are used for both source and target translations. The translation process is treated as word replacement in the source sentence. Different languages may represent the same meaning of a sentence in different word orders, so the word replacement should follow the syntax rules of both languages so that

every word in the source sentence takes the same position as in the target sentence.

The RBNMT can produce accurate translations when the rules are well-defined in a specific domain. Natural language's complexity and variability lead to less fluent and contextually appropriate translations. It has computational inefficiency in determining the adaptive rule of one sentence, and hard to organize the grammar rules since there are too many syntax rules in one language. It ignores the context of information in the translation process which affects the robustness of the approach. For example, two sentences given *"The pen is in the box" and "The box is in the pen"*. The second sentence is ambiguous, as the word "pen" is polysemant (dual meaning), which means "fence" in English. These sentences show the complexity of commonsense reasoning and spatial relationships.

2.4.2 Statistical Machine Translation

SMT translates tasks from a statistical perspective, using a bilingual corpus to find words and phrases with the same meaning. For example, SMT divides a given sentence into several sub-sentences and replaces each part with a target word or phrase. These systems examine word alignment patterns and their frequency within the corpora to create translations based on probabilistic models [107]. The Phrase-based SMT (PBSMT) includes preprocessing, sentence alignment, word alignment, phrase extraction, phrase feature preparation, and language model training [45]. The PBSMT pairs the phrases in the source language with target phrases, the lexicon is built from a training dataset or bilingual corpus.

The SMT method provides greater flexibility and improved handling of various linguistic features compared to Rule-Based Machine Translation (RBMT). Nevertheless, SMT faces challenges, particularly in addressing long-range dependencies and generating grammatically coherent translations, especially in languages with intricate syntax.

2.4.3 Neural Machine Translation

The NMT models is based on deep learning, a sub-field of machine learning and artificial intelligence. The model learns complex patterns using layers of interconnected neurons or neural networks [41]. The NMT approaches with neural networks recognize the patterns in the source sentence to determine

the target sentence. In recent years, these approaches [103, 102, 4] have made rapid progress and are currently state-of-the-art in the machine translation field. This thesis part I, approach relies on NMT some techniques that allow MT have discussed.

2.4.4 Neural Networks

Neural Networks (NN) are machine learning models inspired by biological systems designed to solve complex problems and recognize patterns through layers of interconnected neurons or nodes [18]. NN algorithms mimic the human nervous system, where nerve cells or neurons communicate with other cells by electric signals.

An Artificial Neural Network (ANN) or simply an NN, is a set of neurons that consist of layers, such as an input layer, one or more hidden layers and an output layer. This basic structure of interconnected layers of neurons is known as a Fully Connected Network (FNN) [18]. The data or information is passed in an iterative process through the interconnected layer of neurons, this whole process is called training. The data goes through several transformations to detect the particular patterns that influence the network's final output. The functionality of a layer to its input data is stored in the layer's weights, which essentially, are a set of values associated with an existing neuron link in the layer. The learning involves adjusting the weights or finding the set of numbers, to correctly map input to its corresponding output label. Figure 2.1 shows an example of a deep neural network with two hidden layers, the first layer is input data, and the final layer of output [90].

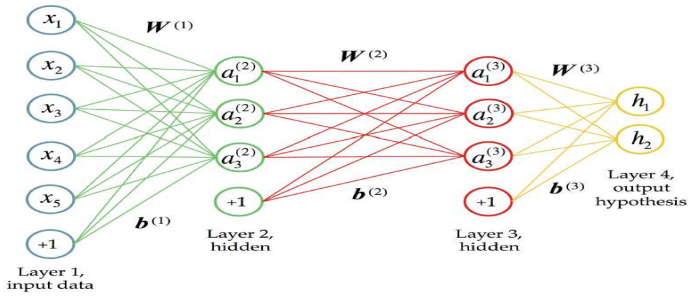

Figure 2.1: Neural Network overview [90].

Three key components guide the learning process: (i) the Neuron Activation Function (decides the activation of neurons), (ii) the Loss Function

(measures prediction error), and (iii) Backpropagation (updates the weights to minimize the loss).

The *Activation Function* of a neuron calculates the output of the neuron based on its inputs and their weights associated with the link of the respective neuron. The activation signal is sent to the neuron in the next layer and represents how a particular neuron responds to a specific input signal. This process of passing the signals and communication between neurons to the next layer is called Forward Pass. There are different activation functions, the common ones are (i) Sigmoid Function, (ii) Hyperbolic Tangent (Tanh) function, (iii) Rectified Linear Unit (RelU). These functions introduce non-linearity, enabling the network to learn complex patterns, and the choice depends on the problem under consideration.

The Loss Function, also known as *Error function*, helps in determining the learning process. It calculates the difference or loss between the model-predicted value and the actual ground truth value or target. It helps in capturing how the model is being trained and guides the model to improve the training accuracy. The choice of loss function depends on the problem. For example, some common loss functions are Cross-Entropy (CE) for classification tasks and Mean Squared Error (MSE) for regression tasks.

Backpropagation is an algorithm that helps to fine-tune the weights based on errors calculated through the loss function in each iteration. The proper tuning of weights reduces error in the iterative process, which helps to improve the training and allows the model to fit appropriately. It computes the error backwards using the Gradient Descent algorithm. An iterative optimization algorithm is used to minimize a loss function by updating model parameters in the opposite direction of the gradient. Gradient Descent helps to know the direction and magnitude of adjustments needed for each weight. The update rule for the weights can be expressed as:

$$w_{i+1} = w_i - \alpha \cdot \frac{\partial J(w_i)}{\partial w_i} \tag{2.8}$$

where w_i represents the current weight, w_{i+1} is the updated weight, and α is a hyperparameter known as the Learning Rate, which controls the step size for the update. The term $\frac{\partial J(w_i)}{\partial w_i}$ denotes the gradient of the Loss Function $J(w)$ with respect to w_i. The training process continues until the loss or a desired measure converges to a local or global minimum or maximum, indicating that the model has reached an optimal state, with no need for further training.

2.4.5 Recurrent Neural Network (RNN)

Recurrent neural networks [85] is the type of neural network, used for processing sequential or time series data. For example forecasting sales and stock market. Neural networks are suitable for handling independent data points, and processing fixed-length inputs, while sequential data can have variable lengths. RNNs have been designed to address these problems and have shown good performance in addressing complex natural language processing tasks. RNN can be used to solve problems such as language translation, natural language processing, sentiment analysis, speech and image recognition [18].

As neural networks, RNNs use training data to learn, but they are distinguished by memory, Figure 2.2, shows the RNN architecture. RNN iterates over the given sequence S, and keeps a hidden state h, that contains the already processed information. The result of processing an input at time t, is conditioned by state $t-1$. At each time step, a new hidden state h_t is computed by current input x_t with the hidden state from the previous time step h_{t-1}. The hidden state computation is expressed in equation 2.9

$$h_t = f(h_{t-1}, x_t) \tag{2.9}$$

Here, the function f is used to compute a non-linear activation function, which is typically a hyperbolic tangent (tanh) function, expressed in the following equation 2.10.

$$h_t = \tanh(Ux_t + Wh_{t-1} + b) \tag{2.10}$$

Here, U is the weight matrix associated with the inputs, W represents the weight matrix, which is linked to hidden units, b denotes a bias vector, and tanh is non-linear activation function applied element-wise to the vector.

This mechanism enables RNNs to process input sequences of variable length by recursively updating the hidden state across time steps.

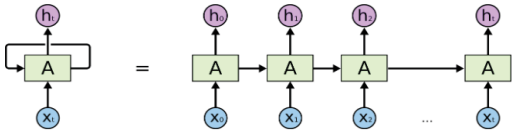

Figure 2.2: Recurrent Neural Network overview [90].

RNNs are categorized into four types based size of the input and output, and these types are one-to-one, one-to-many, many-to-one, and many-to-many. The many-to-many configuration also known as sequence-to-sequence, is particularly useful for tasks such as Text Summarization and Machine Translation.

2.4.6 Long Short-Term Memory (LSTM)

The basic version of RNN has shown good performance in many NLP tasks. Particularly, in processing sequential data in real-time where inputs are received one at a time and output predictions are based on the most recent input. They have problems with vanishing gradient descent and difficulty in learning long dependencies [47]. The vanishing gradient problem affects the deep networks using activation functions like sigmoid or tanh. Due to the backpropagation mechanism, the gradient becomes less and approaches near zero. This leads to minimal or even unchanged weights in the earlier layers. This problem is especially significant when dealing with long sequences due to more steps needed in backpropagation. This challenge makes it difficult to learn long sequences as the memory (or state) is frequently overwritten at each step, limiting the model's ability to capture only short-term dependencies.

Long short-term memory was proposed by Schmidhuber and Hochreiter [46] to overcome the problem of vanishing gradient. It can handle long-term dependencies, as LSTMs have 'memory cells' in hidden layers which have 3 gates, an input gate, an output gate, and forget gate [18]. These gates control the flow of information that is needed to predict the output. Figure 2.3 shows the information flow through the LSTM cell. LSTM saves information for later by preventing older signals from vanishing during processing [76].

LSTM have the mechanism of gates, to control the flow of information and decide about the important information what to retain and what to discard. The processing of data within the LSTM cell follows these steps:

$$i = \sigma(x_t U^i + h_{t-1} W^i)$$

$$f = \sigma(x_t U^f + h_{t-1} W^f)$$

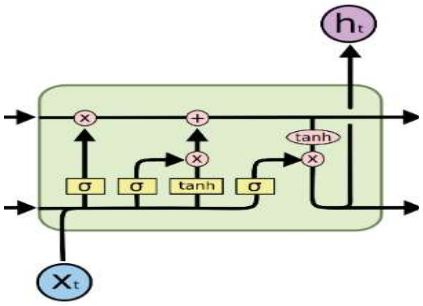

Figure 2.3: LSTM cell mechanism [76].

$$o = \sigma(x_t U^o + h_{t-1} W^o)$$

$$g = \tanh(x_t U^g + h_{t-1} W^g)$$

$$c_t = c_{t-1} \circ f + g \circ i$$

$$h_t = \tanh(c_t) \circ o$$

Here, i, f, and o represent the input, forget, and output gates, respectively, which are controlled by the Sigmoid function σ. The vectors are scaled to values between 0 and 1. The candidate memory state g is computed through the tanh function, on the current input and the previous hidden state. The new memory cell c_t at time t is a combination of the previous memory cell c_{t-1} and the current input depending on information from input gate i and candidate memory g. Finally, the new hidden state h_t is computed based on output gate o and new memory state c_t. In previous equations, the operator $\circ$ represents element-wise (Hadamard) multiplication.

This memory cell design allows the gradient to travel back without vanishing significantly, and overcome the problem of vanishing gradient descent. RNNs suffer in longer contexts, for example, "Sarah is vegetarian. She doesn't eat meat." The context of vegetarian, anticipates that meat can't be eaten. If this context was a few sentences prior, it is difficult for RNN to connect to the context. However, LSTM network can handle this problem efficiently. LSTM alleviates some of the limitations of recurrent neural networks, but they still struggle with large datasets and long sequences due to their sequential nature, which limits parallelism during training.

2.4.7 Bidirectional Recurrent Neural Networks (BRRNs)

The RNNs process the input in one direction only. The BRRNs process the sequence in both forward and backward directions [87]. There can be situations, where information from the future might be more important than the current time point. In that cases, BRRNs are important to see the future words usually, they consist of two Recurrent Neural Networks (RNNs), in form of simple RNN, LSTM or Gated Recurrent Units (GRUs) [17]. Each RNN processes the input sequence in one direction, and the output is combined by sum, average, multiplication or concatenation.

2.4.8 Sequence-to-sequence for MT

A key development in NMT is the sequence-to-sequence (seq2seq) framework, which generates the target using an encoder-decoder architecture [93]. The encoder reads the source sentence and converts it into a context vector representation, and the decoder processes this context vector to generate the target sentence word by word.

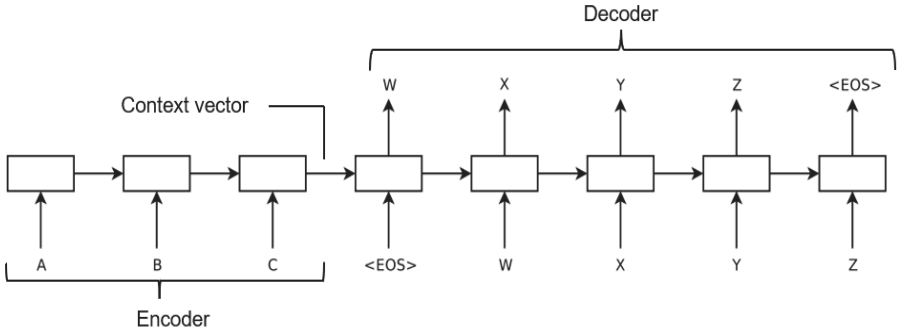

Figure 2.4: Sequence to sequence Network overview [93].

Given a source sentence $\mathbf{x} = (x_1, x_2, \ldots, x_T)$, where x_t denotes the word at position t, and a target sentence $\mathbf{y} = (y_1, y_2, \ldots, y_{T'})$, NMT seeks to maximize the conditional probability for the target sequence $\mathbf{y}$ given the source sequence $\mathbf{x}$ and the model parameters θ. This can be expressed as:

$$P(\mathbf{y}|\mathbf{x};\theta) = \prod_{t=1}^{T'} P(y_t|y_1,\ldots,y_{t-1},\mathbf{x};\theta) \tag{2.11}$$

In this equation, $P(y_t|y_1,\ldots,y_{t-1},\mathbf{x};\theta)$ represents the probability of generating the t-th word of the target sentence, conditioned on the previously generated words and the source sentence $\mathbf{x}$.

Figure 2.4 shows the encoder-decoder architecture, and processing of input sequence at encoder and generation of context vector, at decoder the iterative process of how the output y_t serves as input for the next time step $x+t$. It helps the decoder incorporate the previously generated token. The special symbols are introduced to indicate the end of the sentence $< EOS >$. It enables the model to define a distribution over sequences of all possible lengths. As highlighted in Figure 2.4, the model computes the representation of "A", "B", "C", "$< EOS >$", and after this computes the probability of "W", "X", "Y", "Z", "$< EOS >$".

The challenge is that traditional sequence-to-sequence models that employ recurrent networks struggle to retain enough information when dealing with long sentences as the fixed-length context vector makes it hard to retain enough information. The handling of long-range dependencies is improved by attention mechanisms and transformer models.

2.4.9 Attention Mechanisms

The problem of source fixed encoder representation is not optimal for both encoders and decoders, since it is hard to compress the sentence, while for the decoder different information may be relevant at different steps. For seq2seq models, the encoder compresses the whole source sentence into a single vector. It is hard at the decoder end to decode one source of representation from fixed representation. Several studies [4, 16, 108] have shown the performance of the RNN-based seq2seq model is affected as the length of the input sequence increases. That is mainly due to two problems, namely the decoder has limited information about input in a fixed length context vector, and with longer sentences, the backpropagation has more steps which leads to the vanishing gradient problem. LSTM has somehow solved this problem but not completely solved it.

To address this problem the 'Attention' mechanism was introduced by Bahdanau et al. [4] to let a model focus on different parts of the input.

Overall, the attention mechanism is part of the neural network. Here, an encoder does not compress the whole sentence into a single vector, rather, it gives representations for all source tokens, such as all recurrent neural network states instead of the last one. It allows the decoder to use the relevant information directly from the input sequence to predict the current token in the output sequence. This mechanism preserves the context from beginning to end. The core concept of the Attention mechanism is to preserve all hidden states generated by the encoder at each time step. Then a weighted combination of decoder output and encoder states is formed at each time step. It means, the decoder can access the entire input sequence and selectively focus on certain parts of that sequence particularly one with the highest weights to generate the output.

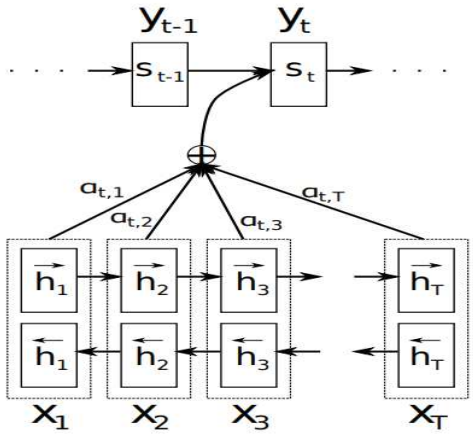

Figure 2.5: Graphical overview of Bahdanau Attention [4].

Attention has three basic steps, the first step is to calculate the attention score e_{ij} between hidden states of all encoder states $h_1, h_2, \ldots, h_m$, and the decoder's last output s_{t-1}. The resulting attention score vector shows, how the input is aligned with the output at the position t. The alignment model, is represented with an attention score represented as score$(S_t - 1, h_i)$. which scores how well the input around position j and the output at position t match, is represented as:

$$e_{ij} = \text{score}(s_{t-1}, h_j)$$

Here, the score function score(s_{t-1}, h_j) is defined as:

$$e_{ij} = v_a^T \tanh(W_a s_{t-1} + U_a h_j)$$

Once the attention scores e_{ij} are calculated, the next step is to compute the normalized alignment weights α_{ij}, which are obtained by applying a Softmax function:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

The final step is to compute the context vector c_i as a weighted sum of the encoder hidden states, where the weights are the alignment scores α_{ij}:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

This context vector c_i is then used by the decoder to generate the next word in the output sequence. The c_i is not a fixed length, and its size depends on the input sequence length. Due to the Softmax function, if the input score of an element is closer to one, its effect and influences on the decoder output are amplified. Figure 2.5 gives an overview of the explained mechanism, the model trying to generate $t-th$ target word y_i from a given input sentence $x_1, x_2, \ldots, x_m$ The context vector c_i is concatenated with the previous decoder output and then fed to the decoder cell to produce a new output.

Luong's General Attention [71] mechanism computes the alignment scores between the decoder's hidden state s_t and the encoder's hidden states h_i. The attention scores are calculated using three different methods: Concat, General, and Dot. The respective equations for each method are as follows:

$$\text{score}(s_t, h_i) = v_a^T \tanh(W_a[s_t; h_i])$$

$$\text{score}(s_t, h_i) = s_t^T W_a h_i$$

$$\text{score}(s_t, h_i) = s_t^T h_i$$

Once the alignment scores are computed, they are passed through a softmax layer to obtain the attention weights. The context vector c_t is then

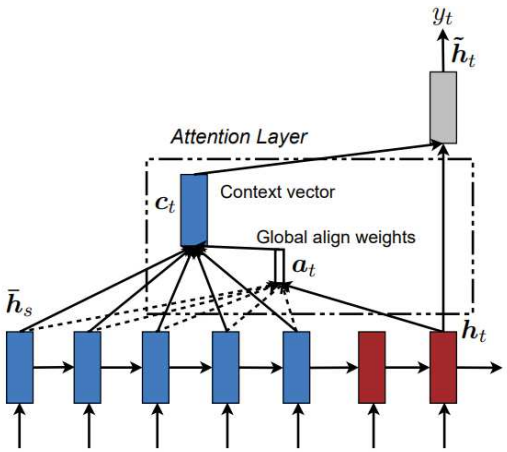

Figure 2.6: Luong Global Attention model [71].

computed as a weighted sum of the encoder's hidden states and concatenated with the decoder's current hidden state before generating the output.

Figure 2.6 illustrates the global attention model, a global context vector c_t is calculated as weight average according to a_t, over all source states [71].

Luong's Local Attention [71] mechanism reduces the computational cost by focusing on a subset of the source hidden states. The context vector c_t is calculated by taking a weighted sum over a window of source hidden states $[p_t - D, p_t + D]$, where D is chosen empirically. The position p_t is determined in two ways: monotonic alignment and predictive alignment.

In monotonic alignment,

$$p_t = t.$$

Here, the target and source sentences are assumed to be aligned monotonically. In Predictive alignment,

$$p_t = S \times \sigma(v_p^T \tanh(W_p s_t))$$

the S is the source sentence length, and v_p^T and W_p are trainable weights. A Gaussian distribution is applied around p_t to favour nearby positions.

Figure 2.7 illustrates the local attention model graphically. A window, centered on the source position p_t, is applied to calculate a context vector c_t. The attention weights a_t are computed based on the current target hidden state h_t and the source hidden states $\bar{h_s}$ within the defined window [71].

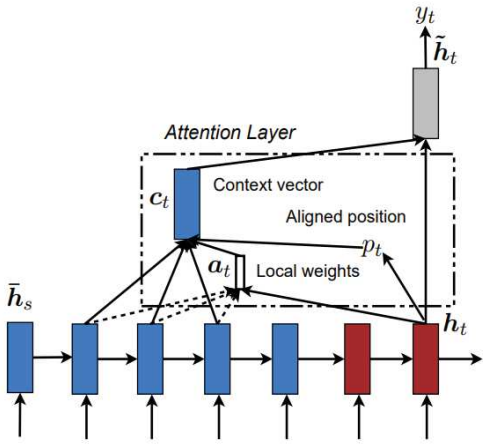

Figure 2.7: Luong Global Attention model [71].

2.4.10 Transformers

The transformer model is introduced by Vaswani et al. [95] based on an attention mechanism without recurrence or convolutions. The Transformers was the first NMT model to rely on the Self-Attention mechanism to compute input and output representation [95]. The most advanced models [63, 83] for NMT build on Transformer architecture, and Transformers are considered state-of-the-art in NMT and many other tasks in NLP.

What makes Transformers different from the attention mechanism is its unique architecture and key component of self-attention, the architecture of the model is shown in Figure 2.8. An overview of Transformer's main architectural parts is provided here.

Self-attention allows words or tokens in a sentence to interact with each other by gathering contextual information and updating the self representations. Unlike traditional attention, which typically operates between encoder and decoder states, self-attention works within the same layer, refining the relationships between tokens in the same sequence (e.g., among all encoder states in a layer). By taking in N-word embeddings without context as input, it returns N-word embeddings with contextual information. This mechanism follows the intuition of attention and each input token is assigned three representations: (i) Query Q, (ii) Key K, and (iii) Value V.

The alignment scores are computed by matching the given query q with all key vectors k_i through dot-product, following the similar approach discussed

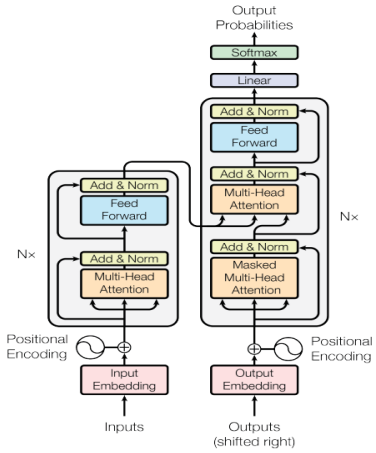

Figure 2.8: Architecture of Transformer Model [95].

by Luong et al. These scores are transformed using a Softmax function to produce weights. The weighted vector determines the level of attention a word represented by q should give to each other in the sentence. The final output, or vector representation is derived by calculating the weighted sum of the value vectors v_{k_i}, where each value vector corresponds to its respective key. The complete process is performed simultaneously for all words in the sequence, following the vectorisation and linear algebra operations of the queries, keys, and values into matrices. All calculations are done following this equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.12}$$

Here, Q, K and V are the matrixes for queries, keys and values respectively. The attention weights are the dot product of K and V normalized by the square root of the dimensionality of the key vectors d_k, applied with the softmax function. Finally multiplied by the value V to get the attention output.

In the decoder, the self-attention mechanism is referred to as *Masked Self-Attention*. While the encoder processes all tokens simultaneously, allowing them to attend to each other, the decoder generates tokens one at a time. To prevent the decoder from accessing future tokens, masked self-attention is applied. During training, reference translations (targets) are provided to the

model, and without masking, future tokens would be visible to the current token, which would compromise learning. Transformers, unlike recurrent models, process all tokens in parallel without recurrence.

The *Multi-Head Attention* mechanism in Transformers helps in focusing and understanding the role of the word related to different parts of the sentence. In this mechanism queries, keys and values are linearly projected h times using different learned projections. The Self-Attention is then applied independently to each of these h projections in parallel, to produce h outputs. These outputs are subsequently concatenated and projected again to form the final result. The core idea of Multi-Head Attention is to capture information from different subspaces, which is otherwise not feasible with single-head. The function is represented in the following equations:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_n)W^O \tag{2.13}$$

$$\text{head}_i(Q, K, V) = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.14}$$

Here, $QW_i^Q, KW_i^K, VW_i^V, W^O$ are weighted matrices, and the outputs from all heads are concatenated and multiplied by weighted matrix W^O, which enables the model to focus on different parts of the input sequence without increasing the model size.

The other components include *feed-forward blocks*, which consist of two linear layers with ReLU non-linearity between them. The purpose is to process and refine the information gathered using attention. The other components include *residual connections, layer normalization and positional encoding*. In the Add & Norm layer of the Transformer, residual connections are applied after the attention and feed-forward blocks. These connections are crucial for gradient flow and allow the stacking of many layers. The layer normalization normalizes the vector representation of each example in batch and helps to control flow to the next layer, improving convergence and quality. As Transformers does not know the order of input tokens, to let know model about the positions of the tokens the input representation has two embeddings of token and position. The main components and general idea of the Transformer model are described here, while further details are referred to source paper [95].

The Transformer has been widely adopted across various domains, including machine translation [96, 72, 83], computer vision [13, 70], speech applications [15, 110], and multimodal tasks like question answering [48, 64].

Transformer models can be categorized based on their architecture: encoder-only models, such as BERT [22], RoBERTa [69] are typically used for natural language understanding tasks, while decoder-only models, like the Generative Pre-trained Transformers (GPT) series, GPT [81], GPT-2 [82], and GPT-3 [9] have demonstrated impressive few-shot performance by leveraging large-scale pretraining, with task-specific prompts fed into the model. Encoder-decoder models, such as BART [63], extend BERT's denoising objective into a sequence-to-sequence framework, enabling both text generation and understanding. T5 [83] follows a similar architecture, being one of the first to introduce task-specific text prefixes for various downstream tasks. The advantage of an encoder-decoder setup is its ability to perform both natural language understanding and generation, making it highly versatile across different NLP tasks such as machine translation.

2.5 LLMs & Part-of-Speech (POS) Tagging

This section gives a basic overview of LLMs and POS tagging used in the second part of the thesis.

2.5.1 Large Language Models

The introduction of LLMs, such as GPT and BERT, has revolutionized NLP by enabling machines to process and generate human language with unprecedented accuracy [96]. These deep neural network models owe their effectiveness to the transformer-based architectures [96], which utilize self-attention mechanisms to process and contextualize vast amounts of text. Most currently available LLMs have billions of parameters and are trained in a self-supervised way to predict missing tokens or the next token in a given sequence. LLMs are usually instructed through text prompts to solve a specific task, such as translating or answering questions.

Prompting It is a technique to effectively communicate with LLM to get the desired response or output. Prompt engineering allows users to extract knowledge without extensive retraining. To interact with LLMs, different types of prompts are used, such as: (i) zero-shot learning, where a task is given without prior examples; (ii) one-shot learning, where one example is provided to give context; and (iii) few-shot learning, where a few examples are

given to help the model understand the required pattern of response. There are other prompting techniques such as Chain-of-Thought (CoT), where the instruction is given step-by-step. Prompt engineering is an emerging field as task-specific prompts are designed for different NLP [97].

In this thesis, we have used the few-shot prompting technique, as LLMs have also been used successfully for semantic parsing, e.g., converting text into a structured format for analysis [23, 75, 106].

2.5.2 POS Tagging

The POS tagging involves assigning labels to tokens within a text based on their grammatical function, i.e., whether the token is a noun, verb, adjective, adverb, or other [51]. Given a sequence $x_1, x_2, \ldots, x_n$ of words (tokens) and a set of tags, the goal is to generate a sequence $y_1, y_2, \ldots, y_n$ of tags, where y_i represents the assigned tag for the input x_i. POS tagging presents challenges due to word ambiguities because a word can have multiple meanings and functions depending on the context in which it is used. Current POS tagging techniques utilize contextual information to resolve these ambiguities.

Tag	**Description**
CC	Coordinating conjunction
IN	Preposition or subordinating conjunction
JJ	Adjective
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper Noun, singular
NNPS	Proper noun, plural
RB	Adverb
VB	Verb, base form
WDT	Wh-determiner
WP	Wh-pronoun

Table 2.1: Common POS tags and their descriptions.

In our approach, we use Universal POS tags by leveraging the spaCy library[1]. Table 2.1 list some common POS tags used in the implementation of

[1]https://spacy.io/

our system LLM2LAS. The following example demonstrates how sentences are processed: in the sentence *"Mary went to the store and she bought food"* the word *"she"* is replaced by *"Mary"*. All sentences are normalized by identifying coreferences and then enriched with POS tags. For example, the tagged sentence has POS tags: $[NNP, VBD, IN, DT, NN, CC, NNP, VBD, NN]$.

Part I

Towards Automatic Composition of ASP Programs from Natural Language Specifications

Chapter 3

The NL2ASP Tool

This chapter introduces the problem of transforming natural language specifications to answer set programming, with underlying assumptions, dataset construction, and the architecture of the NL2ASP tool.

3.1 Problem and Some Assumptions

The problem we tackle in this part of the thesis is to ease the process of composing formal ASP statements (ASP programs) from problem specifications given in natural language. However, this is quite an ambitious final objective. In order to take the first steps towards pursuing this goal we begin by making some pragmatic simplifying assumptions.

The first assumption we make is: problem formulations are given as a bag of statements, where contiguous subsets of statements can be encoded with a logical rule. This makes it possible to compose a program by iteratively transforming groups of statements in the corresponding rule.

The second assumption we make is that we are given a corpus of pairs that associates each bag of statements to a *"gold program"*. The gold program is an example of an expected encoding of the statements in ASP. A subset of that corpus can be used to *train* a neural network to perform the task of ASP program composition (training set); another (disjoint from the training set) subset of that corpus can be used for testing the correctness of the system (test set).

The problem we tackle is, thus, formalized as follows: Given a specification P of a problem (a bag of statements), compose a program P_{asp} that is

uniform equivalent to the gold program P_{gold} associated to P.

Two ASP programs P_1 and P_2 are uniform equivalent iff for any set of (non-disjunctive) facts F, $P_1 \cup F$ and $P_2 \cup F$ admit the same answer sets [25]. This captures that an ASP program models uniformly a problem over varying instances provided via facts [8].

The formulation of the composition problem makes it possible to iteratively process the input by identifying the statements corresponding to a rule, and transforming such statements into the corresponding rule. In our approach, this latter task is performed in two steps: (*i*) translation of the natural language in a corresponding CNL (NMT task); (*ii*) conversion of the CNL in ASP (the basic task of CNL2ASP).

To the best of our knowledge, no dataset supporting the above tasks is available; thus, we have developed one as described in the next chapter **??** . Pragmatically, we concentrated our efforts on graph-related problems. This is a domain of problems that is classically addressed with ASP [8, 26, 65, 11], and is very rich and variegate (in terms of number, complexity and importance of problems) to be considered a meaningful pragmatic choice to assess our approach.

Concerning the issue of testing our implementation, we observe that checking solutions is intractable (i.e., is undecidable in general and very hard for specific cases, e.g., uniform equivalence belongs to the second level of the polynomial hierarchy for already ground disjunctive ASP programs) [25]. Thus, in our experiment we will pragmatically consider, in analogy to what has been done in other similar contexts, looser measures of correctness, such as purely syntactic measures (BLEU, METEOR); and, in end-to-end tests, correctness was certified by a human expert.

3.2 Dataset Construction

One of the contributions of our work is a specialized dataset centered on graph-related problem specifications. This section details the creation process of this dataset, named NL2CNL. The workflow of the dataset creation is illustrated in Figure 3.1, which entails collecting ASP encodings of graph-related problems. NL2CNL includes problem statements in natural language and corresponding CNL propositions.

The ASP encodings were mainly collected from ASP competitions (2015[1],

[1]http://aspcomp2015.dibris.unige.it/benchmarks-suite-and-selection

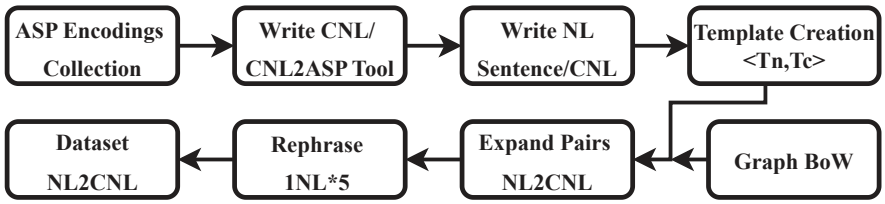

Figure 3.1: Dataset creation workflow

2017[2]), books [37], lecture notes, data shared by renowned professors, and other online sources[3][4][5]. For each collected ASP encoding, we have written the corresponding CNL and generated a new ASP encoding by using the CNL2ASP [12] tool. Then, we cross-verified the answer sets of both actual ASP encoding and tool-generated ASP encoding to identify trivial problems, and checked the encoding manually. This way, we ensure the CNL we have written is compliant with the actual encoding. We have tried to cover the five main grammar propositions of tool CNL2ASP: (i) negative strong constraint proposition, (ii) positive strong constraint proposition, (iii) weak constraint proposition, (iv) definition proposition, and (v) quantified choice proposition. We have ensured that each CNL is accompanied by a corresponding natural language statement that accurately reflects the original semantics, while also retaining important information about the *verb, noun*, and *variable names* used in the CNL. Despite successfully generating NL statements and CNL propositions for 20 programs, the resulting dataset (494 pairs) is too small for effective use in data-driven approaches. To address this limitation, augmentation techniques are applied to expand the size and diversity of the dataset.

Motivated by the existing template-based approaches [49, 24, 104, 94] in creating quality datasets, we decided to apply this type of approach in our case. Templates ensure content consistency by generating syntactically and semantically correct data, minimizing dataset errors. They are easily modifiable and scalable, ensuring well-structured, error-free datasets with specific patterns. Thus, we manually created CNL-NL template pairs based on the propositions of CNL2ASP, and ended up developing 369 unique templates

[2]http://aspcomp2017.dibris.unige.it/index.php/benchmarks-suite-and-selection
[3]https://asparagus.cs.uni-potsdam.de/
[4]http://pages.suddenlink.net/ykahl/
[5]https://teaching.potassco.org/

for each grammar proposition.

Sentence	**Template**
CNL: *Node 1* have an *edge node X*, where *X* is one of *2, 5.*	TCNL: *Noun_1 num_1* have an *verb_1 noun_1 var_1*, where *var_1* is one of *num_2, num_3.*
NL: There is *node 1* has an *edge* to *node X*, where *X* is one of the numbers *2 or 5.*	TNL: There is *noun_1 num_1* has an *verb_1* to *noun_1 var_1*, where *var_1* is one of the numbers *num_2 or num_3.*

Table 3.1: An overview of CNL & NL pair template

Templates were obtained as follows: we start from a valid CNL-NL pair, and we replace specific parts of speech (nouns, verbs, etc.) with placeholders in order to obtain a template CNL-NL pair. A placeholder is a symbolic representation of a specific replacement, and we used conventions for different categories: numbers are "num_X", verbs are "verb_X", nouns are "noun_X", variables are "var_X", colors are "color_X", and special predicate identifiers *PIDs* are "PID_X", and so on ("X" is a positive integer ensuring uniqueness of placeholders). Based on CNL patterns for specified facts, numeric placeholders are: (i) *num_range*, defined to signify a range of numbers; and (ii) *num_choice*, used for referencing specific numbers. For instance, the range from 1 to 4 is presented as *num_range (1 to 4)*, and *num_choice (1, 2, and 5)* or *num_choice (2 or 3)*. This process is exemplified in Table 3.1, where a template is obtained by introducing placeholders in sentences of a CNL-NL pair, e.g. *"noun"* placeholder, used to replace the word *"node"*, *"verb"* placeholder replaces the word *"edge"*, variable *"var"* placeholder replaces capital alphabet *"X"*, and number *"num"* placeholders replace the numbers *2 and 5.* Numeric placeholders are accompanied by other categories like verb *"verb_X"* and noun *"noun_X"* for word variations. Placeholder *"var_X"* represents capital alphabets, while *"col_X"* denotes a list of colors. Special placeholders are introduced for contexts with the clause "whenever,", where the defined list of placeholders follows the pattern of *PID_X*, e.g. *PID_1* is replaced with *"first vtx"*.

After creating the unique template CNL-NL pairs, we created suitable *Bag-of-Words* (BoW) to obtain possible replacements for placeholders. We considered three categories of words, *verbs, nouns*, and special predicate identifiers like *PIDs*. We manually identified: 21 *PIDs*, 77 *nouns*, and 408

Grammar Type	Source	Gener.	Rep.Ct.	Total
Def. Const/Comp.	154	21	875	1050
Def. 'When'	145	15	800	960
Def. 'Whenever'	110	41	755	906
Neg. Constraint	22	138	800	960
Pos. Constraint	39	121	800	960
Quant. Choice	13	156	845	1014
Weak Constraint	11	149	800	960
Grand Total	494	641	5675	6810

Table 3.2: Count of CNL2ASP propositions in NL2CNL. 'Gener.' stands for 'Generated', 'Rep.Ct.' stands for 'Rephrased Count'.

verbs. Words that might be used as predicate names were taken from problem specifications, and the others were added so to construct meaningful CNL and NL.

Having defined the templates and the BoWs, we replaced the placeholders at random by respecting the placeholder expected type (e.g., *verb_X* in *templates*, it is replaced by associating a verb from the BoW of *verbs*), *num_X* are replaced by integer values, *num_range* is replaced by numbers generated at random within the delineated limits, *num_choice*, we choose random integers between 1 and 10 and arranged them in order and comma separated.

The distinct counts for each grammar proposition is reported in Table 3.2 in column "Source". To mitigate potential class imbalances in training data-driven models, we instantiated the templates in such a way that the total number of NL2CNL pairs per proposition type in the expanded dataset is almost equal. In summary, we augmented the source dataset with 641 template-generated pairs (cfr. "Gener." column in Table 3.2), resulting in a total of 1135 NL2CNL pairs.

To enhance the quality and diversity of our dataset, we decided to perform an additional paraphrasing task on all the NL statements. This process increases the dataset size and introduces varied linguistic representations while maintaining the original meaning so to enable effective training of a robust Language Model (LM). Thus, we used *OpenAI API*[6] for this task, with *"text-davinci-003"* engine with specific parameters, such as: prompt set to *"Rephrase the following sentence: sentence"*, temperature value of 0.6,

[6]https://platform.openai.com/docs/api-reference

and maximum token limit of 1000. Each sentence was rephrased five times in order to strike a balance between diversity and semantic dilution. Generation of more than five may lead to unintended overlaps or even slight semantic shifts. After rephrasing natural sentences the count increased to 5675 (cfr. "Rep.Ct." column in Table 3.2). All available NL-CNL pairs constitute the dataset NL2CNL of 6810 pairs (see Table 3.2).

3.3 Architecture

The architecture of the NL2ASP tool for the automatic composition of ASP programs from natural language statements is depicted in Figure 3.2. In the first step, we transform NL statements into CNL statements, and in the second step, we obtain the ASP code by running the CNL2ASP tool. As a consequence, we inherit in the implementation some limitations of CNL2ASP, such as the fact that the user has to specify the schema (or the facts) before providing the rules, and the rule specifications must be provided in such a way that some body-to-head dependency is respected [12]. Note that, this limitation is not present in the translation from sentences to CNL. Indeed, in the proposed architecture, the first step relies on the recent advances in Neural Machine Translation [105]. Specifically, encoder-decoder-based Transformer architectures are incorporated to facilitate the translation of NL specifications into CNL propositions. Using these high-performance models, we capture language nuances and produce grammatically accurate translations. The use of NMT to address the first step is a decision that arises intuitively, as it is easy to realise that the task is like translating from English into another (more restricted) language, i.e., the language of CNL2ASP. Our architecture is simple, ensuring adaptability, can be implemented with different NMT tools, and can take profit from the dataset we have presented in the previous section.

In the implementation, we used state-of-the-art transformer-based architectures e.g., T5 [83] and BART [62], which have proven their efficiency for different language tasks like text summarization, question-answering, and machine translation. The T5 model introduces an encoder-decoder transformer architecture that undergoes pretraining in a combination of unsupervised and supervised tasks, with each task transformed into a text-to-text format [83]. The BART model follows a denoising autoencoder approach, where the input sequence is corrupted using a noise function [62]. Then,

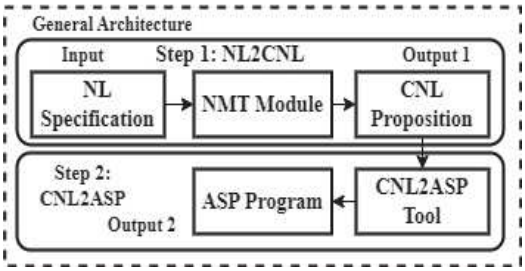

Figure 3.2: Architecture for automatic composition of ASP programs.

the sequence of corrupted entries passes through a bidirectional encoder, which captures contextual information from both directions. Then, the encoder representation is introduced into an autoregressive decoder from left to right, which generates the sequence by predicting the next token based on the previously generated tokens.

Chapter 4

Experiments

We have evaluated empirically our approach, and, in this chapter, describe two sub-analyses: (*i*) assessment of the NMT model's translation quality, and (*ii*) end-to-end assessment of the proposed architecture, i.e., assessment of the ability of our NL2ASP to accurately create ASP programs from natural language. To this end, we fine-tuned pre-trained versions of the T5 and BART models on the proposed dataset and measured several evaluation metrics. Datasets and tools are available at github[1].

4.1 Software and Hardware Details

We adopted the pre-trained T5 and BART models freely available on the Hugging Face platform[2]. Specifically, we used the T5-small[3] and Bart-base[4] models, which have a size of 60 and 140 million of parameters, respectively. The choice of versions was based on the capabilities of the hardware at our disposal.

The models were implemented using Keras 2.12.0 on top of Tensorflow and the Transformers 4.30.2 library. To run the experiments, we used an Ubuntu 20.04 server with 500GB of RAM, a 16 GB NVIDIA Tesla V100-PCIE GPU card, and CUDA 11.8. The code ran using Python 3.9.17 in a Jupyter Notebook environment.

[1] https://github.com/IrfanKareem/nl2asp
[2] https://huggingface.co/
[3] https://huggingface.co/t5-small
[4] https://huggingface.co/facebook/bart-base

4.2 Evaluation Measures

The translation quality of the model was evaluated using the BLEU [77], METEOR [5], and the translation *Syntactic Accuracy (SA)* measures. BLEU is founded on precision-based attributes and functions by comparing the n-grams (a sequence of n words) found in the candidate translation with those in one or more reference translations. BLEU falls within the range of 0 to 1, the larger the higher degree of similarity between the translation and the reference translation. The BLEU score is calculated by:

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{4.1}$$

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ \exp(1 - \frac{r}{c}) & \text{if } c \leq r \end{cases} \tag{4.2}$$

METEOR assesses machine translation hypotheses by aligning them with one or more reference translations. This alignment process considers exact, stem, synonym, and paraphrase matches between words and phrases. The metric relies on the harmonic mean of precision and recall for unigrams. METEOR is calculated as:

$$\text{METEOR} = \text{F}_{mean} \times (1 - Penalty) \tag{4.3}$$

$$\text{F}_{mean} = \frac{P \times R}{\alpha \times P + (1 - \alpha) \times R} \tag{4.4}$$

$$Penalty = \gamma \left(\frac{c_m}{m}\right)^{\beta} \tag{4.5}$$

where *Penalty* is responsible for the correct word order in the candidate. Equation 4.4 calculates the weighted F-Score, with α controlling the weight for precision and recall. In Equation 4.5, the value of γ determines the maximum penalty ($0 \leq \gamma \geq 1$), and β determines the functional relation

between fragmentation and the penalty. On the other hand, c_m is the number of matching chunks, and m is the number of matched unigrams between the reference and the candidate.

As for BLEU, the value falls within 0 and 1, with a value closer to 1 indicating a higher translation quality. METEOR is said to have a better correlation with human judgment than BLEU. The SA (see Equation 4.6) metric, measures the proportion of translated sentences that conform to the CNL2ASP grammar.

$$SA = \frac{\#\text{syntantically correct sentences}}{\#\text{total sentences}} \tag{4.6}$$

4.3 Assessment of the NMT Models

4.3.1 Experiment_A: Cross-Validation

In the first experiment, we performed K-fold cross-validation, models T5 and BART are trained k times, the dataset is divided in one fold as a test set, and k-1 folds are used as the training set. This approach provides a robust estimate of model generalization ability. The value of k was set to 5. We use all 6810 NL to CNL pairs from NL2CNL dataset, which are named as features of *"sentences"* and *"targets"* respectively.

To perform the text tokenization, we used the *AutoTokenizer* utility provided by the Transformer library, which allows us to use the pre-trained tokenizers of T5-small and Bart-base. To handle the collating and batching of input data for training, we used the *DataCollator* utility in the Transformer library. It plays a vital role in streamlining the data preparation process and

Split	BLEU-1	BLEU-2	BLEU-3	BLEU-4	MET
0	0.955	0.944	0.939	0.925	0.960
1	0.966	0.957	0.952	0.940	0.969
2	0.961	0.952	0.947	0.935	0.966
3	0.959	0.949	0.944	0.931	0.961
4	0.962	0.952	0.947	0.935	0.965
Avg.	0.961	0.951	0.946	0.933	0.964

Table 4.1: Scores across splits for T5-small model

ensuring seamless integration during training and inference. The batch size was set to 16.

The Adam Weight Decay (AdamW) optimizer is used in the training process. This optimizer is based on the adaptive estimation of first and second order moments with an added method to decay the weights. AdamW is a variant of the Adam optimizer that decouples weight decay from the adaptive learning rate, which leads to better performance. We set learning rate= 2×10^{-5}, and weight decay= 0.01.

To calculate the BLEU we have used *"corpus_bleu"*, a function from the NLTK [99] library in Python. It computes the BLEU score of the entire dataset, by taking multiple pairs of source and reference sentences. Moreover, we decided to calculate the cumulative BLEU up to 4-gram. We have tuned METEOR with standard parameters, i.e. alpha=0.9, beta=3, and gamma=0.5. These values make the trade-off between recall, precision, the penalty for sentence length mismatches, and word order differences respectively. We ran the training of the models for 200 epochs for each split and applied the Early Stopping technique to monitor the validation loss with minimum mode and patience equal to 20, thus reducing the overfitting and computational resources.

Table 4.1 illustrates the T5-small BLEU and METEOR scores by split and the final average. The BLEU scores for all n-grams are generally high, ranging from 0.90% to 0.97%, which shows strong performance in terms of n-gram overlap with the reference text. The METEOR score also indicates high performance, with values ranging from 0.94% to 0.98% across the splits. On the other hand, Table 4.2 shows the metrics scores for the fine-tuned Bart-base model. Bart-base model scores are low compared to model T5-small (cfr. Table 4.1).

Although training times might vary with hardware, there are significant differences in training and scoring times between T5-small and Bart-base models. T5-small takes about 5 hours on average to train across the split, with scoring times of 3 to 5 minutes, while Bart-base averages approximately two hours for training and six hours for scoring. Despite Bart-base requiring fewer epochs, it spends more time on calculating translation quality measures. T5-small surpasses Bart-base in all translation quality measures and excels in high-quality text generation. Despite a longer training time, T5-small delivers faster predictions compared to Bart-base. These performance advantages make T5-small the preferred choice, despite it requires longer training times.

Split	BLEU-1	BLEU-2	BLEU-3	BLEU-4	MET
0	0.841	0.783	0.751	0.686	0.875
1	0.845	0.786	0.753	0.686	0.877
2	0.853	0.797	0.767	0.704	0.890
3	0.818	0.758	0.726	0.660	0.864
4	0.842	0.787	0.757	0.696	0.879
Avg.	0.84	0.782	0.751	0.686	0.877

Table 4.2: Scores across splits for Bart-base model

4.3.2 Experiment_B: Syntax Correctness

This more demanding experiment aims at checking the syntactic correctness of the CNL propositions produced by the two models. Thus, we assessed each CNL proposition using the syntax checker from the CNL2ASP tool. In particular, the syntactic correctness checking was performed during K-fold cross-validation by running the trained models on their splits. Since the NL2CNL dataset has a length of 6810, for each split, a validation set of 1362 examples is used. Again model T5-small, with a SA of about 94%, performs better than model Bart-base with SA is around 55% (cfr. Table 4.3).

Looking at problematic generations we observe that T5-small exhibits issues, particularly with the clause "whenever" when used with other propositions. Also sentences exceeding 60 characters occasionally result in incomplete generation. Some minor issues are present in the transformation of "less than or equal to" to "at most or equal to,". Nonetheless, a human user can fix these issues with minor adjustments. On the other hand, Bart-base model exhibits some more issues in addition to those previously discussed for T5-small. These include occasional generation of additional spaces (e.g.,

Split	Total	Bart-base		T5-small	
		Errors	Acc.	Errors	Acc.
0	1362	650	52.28	93	93.17
1	1362	630	53.74	78	94.27
2	1362	572	58.00	69	94.93
3	1362	619	54.55	82	93.98
4	1362	600	54.95	80	94.13

Table 4.3: Syntax check for T5-small & Bart-base with K=5.

"C2" generated as "C 2"), inconsistent word connections (e.g., vice versa), and problems with capitalization (e.g., "whenever There is a"). Furthermore, the model occasionally uses dashes instead of underscores to connect words (e.g., "Dominating_Set" vs. "Dominating-Set"), which is not acceptable for the CNL2ASP parsing tool, but could be addressed with post-generation processing.

4.3.3 Experiment_C: Assessment on Unseen

In this experiment, we train both models with the complete dataset. During training, T5 converged at 155 epochs, with a training loss of 0.022 and validation loss equal to 0.0467; the best BART model was obtained at 37 epochs, with training and validation losses of 0.010 and 0.0492, respectively. These values are quite low, evidencing the ability of the models to learn from the dataset. As test set we prepared a new test dataset manually, which consisted of 209 new specifications. We did the inference and calculated BLEU, METEOR, accuracy, precision, recall, and F1-score for both T5-small and Bart-base. T5-small consistently outperformed Bart-base in all metrics, see Table 4.4. Both models generalize well and achieve robust performance on unseen instances.

4.3.4 End-to-End Evaluation

We now measure the ability of the NL2ASP tool to produce valid ASP programs. To this end, we targeted six well-known graph-related problems, i.e. Maximize Clique, Hamiltonian Cycle, Graph Coloring, Connected Dominating Set, Traveling Salesman Problem, and Hierarchical Clustering. We manually created a dataset with the specifications for the aforementioned problems, including two different encodings for the Graph Coloring problem (called GCv1 and GCv2). The inherent flexibility of natural language allows for a multitude of ways to express the same concept. So to make the assessment more realistic the dataset was expanded by paraphrasing the former specifications with the support of ChatGPT. In total, we obtained 21 problem specifications.

The NL2ASP tool, configured with the best performing T5-small fine-tuned in the Experiment_C, was fed with one problem specification at time to obtain the corresponding ASP programs. Since automatic checking of correctness is unfeasible, we manually checked the produced programs.

Metric	T5-small	Bart-base
BLEU-4	0.860	0.704
METEOR	0.935	0.876
Precision	0.929	0.860
Recall	0.927	0.921
F1-Score	0.928	0.88
Accuracy	0.368	0.253

Table 4.4: Evaluation measures for T5-small & Bart-base.

During the translation step, NL2ASP was able to produce 389 syntactically correct CNL statements out of 393, performing for a good 98.98%. In this case, we found minor errors, for example, our tool generated *"at most or equal to"* instead of *"less than or equal to"*. The mentioned issues were found in two variants of the Hierarchical Clustering problem, so NL2ASP was able to produce 19 *correct* ASP programs over 21 problem specifications. It is important to mention that only minor manual fixes would be required by a human to fix the aforementioned issues that affected Hierarchical Clustering instances.

4.4 Additional Experience With LLMs

As a final test, we wanted to check to what extent a general purpose LLM, such as GPT 3.5, is capable of carrying out ASP program composition. The goal of this experiment is not to establish a direct comparison with ChatGPT, but to ensure that specific-purpose tools like ours can perform better than general-purpose models that are not directly trained to tackle this kind of tasks. The prompt we used to request the program generation is: *"Please give the ASP program of the following natural language statements:"*, followed by the natural language statements. As expected, the "out-of-the-box" ChatGPT was able to generate ASP programs given the specification. However, these programs are neither syntactically nor semantically correct most of the time (17/21), and quite some effort would be required to fix them manually. For all wrong programs we asked ChatGPT to regenerate them 3 times, but this resulted in no gain. It is an open research question to explore how/if these models can be robustly used for this task.

Part II

Leveraging ILP for robust question answering with LLM and ASP

Chapter 5

The LLM2LAS Tool

In this chapter, a brief overview of the problem is presented, along with a detailed discussion of the proposed architecture of LLM2LAS.

5.1 Problem Overview

A challenging problem in Artificial Intelligence (AI) is to enhance the ability of the machine's commonsense reasoning and learning capabilities [21]. In recent AI approaches such as transformer-based Large Language Models (LLMs) have fallen short in providing truthful answers and performing on natural language reasoning benchmarks [109, 100, 59]. The lack of transparency and explainability in LLMs makes it challenging to determine whether these models truly acquire commonsense reasoning [86]. The proposed system LLM2LAS addresses the problem of manually creating ASP programs and automatically creating and solving the ASP programs and learning the commonsense knowledge from answer sets using Inductive Learning of Answer Set Programs (ILASP).

5.2 Architecture

In this section, we present our neurosymbolic system LLM2LAS, which combines LLMs with LAS (Learning from Answer Set) to learn commonsense knowledge for story-based Q&A expressed in natural language. As illustrated in Figure 5.1, the system consists of several modules. The *Story Processing* normalizes the story statements and enriches them with POS tagging data;

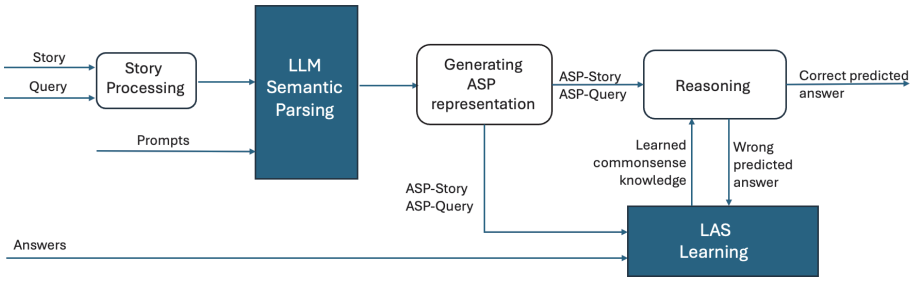

Figure 5.1: Architecture of LLM2LAS

the *LLM Semantic Parsing* generates from the given story relevant fluent representations. These are then used to generate ASP representation of the narrative described in the story. The reasoner module attempts to answer the question using the extracted narrative and the domain-independent rules given in (2.7). If the answer is incorrect, the learner module is invoked to learn relevant commonsense knowledge from the given narrative, question and ground truth answer. We detail each of these steps in what follows.

5.3 Story Processing

The module receives as input a story and a question from which the system is supposed to learn some knowledge. A story consists of an ordered set of statements describing a narrative or a scenario, while the question is designed to be answered by exploiting the information in the story. Each question is associated with the correct and incorrect answers. All sentences are *normalized* by identifying coreferences, both basic and compound, in the text and replacing them with the corresponding referents. The coreference resolution task determines whether two different referring expressions point to the same entity. For example, in basic coreference the sentence *"Mary went to the store, and she bought food."*, the word *"she"* is replaced by *"Mary"*. Moreover, sentences containing negations are identified and flagged to support the automated generation of mode bias in the learning phase. Finally, the module enriches each sentence in the story with a POS tagging sequence and a syntactic parse tree. To perform these pre-processing tasks, we rely on the NLP library spaCy.

Listing 5.1: Prompt for Fact Extraction

```
Please parse the following English sentences into the semantic parse.
The available keywords
are: go_to, and be_in:
Sentence: Mary moved to the bathroom.
Semantic parse: go_to(mary,bathroom)
Sentence: John went to the hallway.
Semantic parse: go_to(john,hallway)
...
Sentence: Where is Sandra?
Semantic parse: be_in(sandra,V1)

Your turn:
```

5.4 LLM Semantic Parsing

LLMs have proven to work well in many NLP tasks, including semantic parsing [23, 75, 106]. We exploit this strength and leverage an LLM to parse the processed sentences and the question into fluent representation, i.e., a predicate that encodes the semantic meaning of a sentence within a story and illustrates the facts. The fluent representation allows for creating the EC representations and the mode bias declarations in the next stage.

Most available LLMs are trained on extensive public data, allowing them to achieve reasonable zero-shot generalization on diverse tasks. However, these models are not expected to perform as well in domain-specific semantic parsing tasks, where the inductive bias from pre-training is less favorable. To address this limitation, we used the few-shot prompting technique, which involves giving the model a few task-specific examples within the prompt to help guide its responses [23]. Listing 5.1 shows an example of the prompt we have designed to ask the LLM to parse the bAbI dataset statements.

For example, if we ask an LLM model to parse the sentence *"Sam moved to the bathroom."*, using the previous prompt, the result would be: `go_to`(`sam`, `bathroom`). Our system parses each statement separately, using the same prompt multiple times to give a precise semantic representation in fluent terms. Table 5.1 (second column) shows a few examples of fluent representations.

5.5 Generating ASP Representation

Once statements of the story are parsed into their corresponding fluent representation, the next step is to create the EC representations (if needed) and the mode bias fluent. The EC representation depicts the actions and their effects in the story and comprises the four predicates introduced in Section **??**. On the other hand, the mode bias fluents aid the learner in automatically generating mode bias declarations for both formal representations.

The construction of the EC representation from the fluent representation involves choosing an EC predicate and a time point. Given a sentence and its fluent representation, we select the predicate according to the following schema: (i) if the sentence is a question, then the "holdsAt" predicate is used; (ii) if the base of the literal's predicate is "be" and the statement is negated, then the "terminatedAt" predicate is used; (iii) if the base of the literal's predicate is "be" and the statement is not negated, then the "initiatedAt" predicate is used; (iv) otherwise, the "happensAt" predicate is used. The time point for the EC predicate is determined by the sentence's placement within the story. The first sentence is given time point 1, and every subsequent sentence has a time point determined by the previous one plus 1. Questions are given a time point according to when they are asked. Table 5.1 shows some examples of statements and their representations.

Mode bias fluents consist of atoms from the sentence's fluent representation where all arguments have been replaced by their types wrapped in either *"var"* or *"const"*. The argument types are determined using the POS tagging data and the WH-determiners of the questions. If the sentence fluent contains an argument that is a variable (i.e., the sentence is a WH-question), then the variable is given a type in the following way: if the sentence is a

Table 5.1: Examples of statements with fluent, and EC representations (Rep.).

Statement	**Fluent Rep.**	**Event Calculus Rep.**
Mary went to the garden.	`go_to(mary, garden)`	`happensAt(go_to(mary, garden), T)`
John and Helen went to the store.	`go_to(john, store)`, `go_to(helen, store)`	`happensAt(go_to(john, store), T)`, `happensAt(go_to(helen, store), T)`
John is in the park or garden.	`{be_in(john, park)`, `be_in(john, garden)}`	`{initiatedAt(be_in(john, park), T)`, `initiatedAt(be_in(john, garden), T)}`
Yesterday Ana went to the park.	`go_to(ana, park, yesterday)`	`happensAt(go_to(ana, park, yesterday), T)`

"what", *"when"*, or *"where"* question, then the variable's type is *"nn"*; if the sentence is a *"who"* question, then the variable's type is *"nnp"*; if the sentence is a *"why"* question, then the variable's type is *"jj"*; if the sentence is a *"how many"* question, then the variable's type is *"number"*. In all other cases, the argument's type is given by its associated POS tag. We provide the types for all arguments that have a temporal aspect and the types of variables in *"why"* questions with *"const"* wrappings. The types of all other arguments are given *"var"* wrappings. Table 5.2 provides the mode bias fluents for the sentences in a story.

5.6 Reasoning

The reasoning module attempts to answer a question using the information extracted from the story and the learned hypothesis. It involves automatically generating and solving an ASP program that combines the ASP representations and learned hypothesis. Ideally, the correct solution to this program (i.e., the answer sets) will contain the correct answer to the question. To extract the answer from the reasoning output, we divide the questions into two types: *"yes/no/maybe"* and others. In the case of the former, we use a *representation search* that checks the question's representation against the answer sets based on the following criteria: (i) if there is at least one answer set, and the representation is in all answer sets, then return *"yes"*; (ii) if there is at least one answer set, and the representation is in some, but not all answer sets, then return *"maybe"*; (iii) otherwise the answer is *"no"*. For all other questions, we extract the answer using a *unification search*, i.e., by finding all ground atoms in the set of answer sets that unify with the question's formal representation. To identify these unifications, a regular expression is constructed from the question's formal representation, replacing variables with the wildcard expression *".*"*. Once unifications are detected, the ground terms corresponding to the *".*"* sections of the regular expres-

Table 5.2: Sentences, fluent representations, and mode bias fluents for a short story.

Sentence	**Fluent Representation**	**Mode bias fluents**
Mary is a mouse.	`be(mary, mouse)`	`be(var(nnp), var(nn))`
Mice are afraid of wolves.	`be_afraid_of(mouse, wolf)`	`be_afraid_of(var(nn), var(nn))`
What is Mary afraid of?	`be_afraid_of(mary, V1)`	`be_afraid_of(var(nnp), var(nn))`

sions are added to the answer list. For example, the question in Table 5.2 generates "$be\_afraid\_ofn(mary, .*n)$". In case of a wrong predicted answer, the learner is invoked with the question and correct answer to learn from.

5.7 LAS Learning

Learning commonsense knowledge from story-based Q&A is initiated through the LAS Learning module. It takes the EC representations of the story and the question, generated by the ASP representation module and the correct and incorrect answers for the question, as input. It creates the context dependent learning task for ILASP by automatically generating the mode bias declarations, using the mode bias fluent representations, and the set E of CDPI examples. To create the mode bias declarations, the system checks whether the sentence is a question, or whether the base of its fluent predicate is "be". This two-check scheme suffices to generate the language bias, given our basic sentences and limited bAbI dataset vocabulary. For questions, the system aims to learn the concept introduced in it. So, its mode bias fluent representation becomes the argument of a mode head declaration, denoted in ILASP as *"modeh"*. If the sentence is not a question and the base of its fluent representation is "be", then ILASP *"modeb"* declarations are generated with the sentence's fluent as argument of mode body declaration. For the story presented in Table 5.2 the mode bias are shown in following listing 5.2:

Listing 5.2: Mode Declaration for story presented in Table 5.2

```
#modeb(be(var(nnp),var(nn))).
#modeb(be_afraid_of(var(nn),var(nn))).
#modeh(be_afraid_of(var(nnp),var(nn))).
```

When LLM2LAS detects that the task requires reasoning about events, the language bias, referred to as EC mode bias, includes dedicated predicates. To learn the concept introduced by the question, LLM2LAS learns if it is *initiated* or *terminated*, considering also the initiation or termination of other fluents in the story. For the question's fluent, two mode head predicates are generated: *"initiatedAt"* and *"terminatedAt"* and declared as arguments of ILASP *"modeh"*. A body predicate is created by enclosing the question's fluent in a *"holdsAt"* predicate, which is then wrapped in *"modeb"*. For

sentences that are not questions, the system detects if they describe an initiated state or a state that holds. In the first case a mode body predicate is created by enclosing the sentence's fluent into a *"initiatedAt"* predicate, in the second case the sentence's fluent is enclosed into a *"holdsAt"* predicate. Both become arguments of ILAPS *"modeb"* declarations. The EC mode bias declarations for the example in Table 5.2 are as following listing 5.3:

Listing 5.3: EC Mode Bias Declarations for story presented in Table 5.2

```
#modeb(initiatedAt(be(var(nnp),var(nn)),var(time))).
#modeb(initiatedAt(be_afraid_of(var(nn),var(nn)),var(time))).
#modeb(holdsAt(be(var(nnp),var(nn)),var(time))).
#modeb(holdsAt(be_afraid_of(var(nn),var(nn)),var(time))).
#modeh(initiatedAt(be_afraid_of(var(nnp),var(nn)),var(time))).
#modeh(terminatedAt(be_afraid_of(var(nnp),var(nn)),var(time))).
#modeb(holdsAt(be_afraid_of(var(nnp),var(nn)),var(time))).
```

The formal representation of non-question sentences is used to prove or disprove other facts using the domain-independent EC rules in (2.7) and the learned hypothesis. So sentences that do not have a "be"-based verb, denote actions at the specific time point in the story. So their mode bias fluent representation becomes argument of the *"happensAt"* predicate. Consider the sentence *"Mary goes to the store."*, whose mode bias fluent is $\texttt{go\_to}(\texttt{var}(\texttt{nnp}), \texttt{var}(\texttt{nn}))$. The system generates the mode body declaration as illustrated in listing 5.4:

Listing 5.4: EC Mode Bias Declaration for 'no be-based & happensAt event

```
#modeb(happensAt(go_to(var(nnp),var(nn)),var(time))).
```

Another key step is the automatic generation of CDPI examples. Examples are created from questions, their stories, and their correct and incorrect answers. Each example corresponds to an incorrectly answered question by the reasoning module, as this would trigger the learning module. Intuitively, the representations of all sentences prior to the question would form the example's context, the formal representations of the correct answer would be part the example's inclusion set and the formal representations of some of the incorrect answers would be part of the example's exclusion set. However, the formal presentation of the story may include choice rules, hence lead to multiple answer sets. When no choice rule is present, if the example is generated

for a yes/no/maybe question and the correct answer is "yes" (resp. "no"), the representation of the question forms the CDPI example's inclusion (resp. exclusion) set and the exclusion (resp. inclusion) set is empty. Similarly for questions which are not yes/no/maybe questions. If choice rules are present in the formal representation of a story, the example generation has to take into account brave and cautious entailment. For a yes/no/maybe question, if the answer is "maybe", then the example will include the question representation in its inclusion set to guarantee that the question's concept occurs in at least one answer set. If the correct answer is a "yes", then a negative example is created where the inclusion set is empty and the exclusion set includes the question's correct answers. This is to guarantee that the question's formal representation is true in all answer sets. In all other cases, a negative example is created with an empty exclusion set and inclusion set given by the representation of the question's answers. This is to guarantee the wrong answer will be false in all answer sets.

To illustrate some of the cases explained above, consider the following story: *Daniel went to the kitchen. Daniel went to the bedroom.* The question: *Where is Daniel?* The correct answer is the *bedroom* and incorrect answer is *kitchen*. If the incorrect answer is predicted, then the following example is created:

Listing 5.5: A Learning program example of short story.

```
%Background Knowledge
holdsAt(F,T+1) :- initiatedAt(F,T),time(T).
holdsAt(F,T+1) :- holdsAt(F,T), not terminatedAt(F,T),time(T).

%Mode bias
#modeb(happensAt(go_to(var(nnp),var(nn)),var(time))).
#modeb(holdsAt(be(var(nnp),var(nn)),var(time))).
#modeh(initiatedAt(be(var(nnp),var(nn)),var(time))).
#modeh(terminatedAt(be(var(nnp),var(nn)),var(time))).

%Positive and Negative Examples
#pos({holdsAt(be(daniel,bedroom),3)},{holdsAt(be(daniel,kitchen),3)},
{time(1..3). happensAt(go_to(daniel,kitchen),1).
happensAt(go_to(daniel,bedroom),2).}).
```

The system requires minimal background knowledge. If Event Calculus (EC) is needed, the background knowledge includes only the rules in (2.7). When ILASP system is run to solve the generated learning task, if the task is satisfiable, the reasoner is updated with the learned hypothesis.

Chapter 6

Evaluation

This section presents an evaluation of the proposed approach using the bAbI dataset from Facebook Research [101]. The dataset is first described, followed by a description of the hardware and software configurations, as well as the baselines used for comparison. Finally, a discussion of the results is provided that confirms the efficacy of our system.

6.1 Dataset

The bAbI dataset is a dataset comprising 20 non trivial tasks of text understanding and reasoning that was proposed by Facebook Research. The bAbI dataset was conceived as a benchmark for assessing a range of natural language reasoning abilities, including deduction, path finding, spatial reasoning, and counting [101]. Each task of the dataset is constructed by simulating words that represent entities and actions. An entity, denoted as a noun, can be a location, an object, or a person, and possesses internal attributes such as size, color, or relative position to cardinal directions. Within this simulation, each entity can perform ten fundamental actions, with each action associated with a collection of replacement synonyms, pronouns, and temporal adverbs, ensuring lexical diversity within the tasks. More in detail, the dataset comprises 4 actors, 6 locations, and 3 objects per task, it features stories ranging from 3 to 229 sentences and 1 to 12 questions. Each sentence within a story is uniquely identified and accompanied by its answer for each task. The dataset includes both training and test data for each task, with a strong focus on learning from a few examples. The stories are also avail-

able in human-readable formats in various languages. In this experiment, we place our focus on the natural language, examining the 13 tasks for which our implementation can be directly applied, namely tasks identified as: 1, 6, 8, 9, 10 11, 12, 13, 14, 15, 16, 18, and 20.

6.2 Hardware and Software Setup

All experiments are conducted on a machine equipped with Intel(R) Core(TM) i7-1255U processors, 16.0 GB of RAM, and GPU NVIDIA RTX 6000 Ada Generation with 48 GB of memory. The experiment pipelines are implemented in the Python programming language version 3.9. Our architecture has been implemented using spaCy, a free open-source library for basic natural language processing tasks (e.g., POS tagging), the open-source LLM Falcon, Clingo 5.6.2 for reasoning on ASP programs, and ILASP 4.0 for learning from answer sets. Concerning the learning parameters, the maximum penalty for the size of the hypothesis was set to 50, and the maximum number of variables was set to 3 for the tasks solved with fluent representation, and to 4 for the tasks solved with EC representation. Each task has been evaluated on 1000 training examples. Our implementation has been compared against two baselines from the literature, also based on logic programming: the ILP-based system in Mirta and Baral [73], and the approach proposed by Yang, Ishay and Lee [106]. All the material needed to reproduce our experiments can be downloaded from `https://github.com/IrfanKareem/llm2las/tree/LPNMR24`.

6.3 Results and Discussion

Our experimental results are summarized in Table 6.1, that provides the accuracy of each task. The task 14, 16, 18, and 20 were solved with fluent representation. Task 14 tests the system's ability to understand the use of time expressions; Task 18 assesses the system's capability to reason about the relative sizes of objects; Task 20 reasons about agent motivations, encompassing the state of an agent (e.g., tired, hungry) and actions such as moving to a location or picking up an object. The remaining tasks (1, 6, 8, 9, 10, 11, 12, 13, 15) are solved with EC representation. These tasks assess the system's ability to reason about object and agent locations, discern true or false statements from mixed information, generate single-word answers based on object properties, identify negations, handle ambiguous queries, detect coreference and conjunction, and deduce basic agents (i.e. animal) properties.

The learning times for our system has an average (over all tasks) of 58s, with a peak of 210s for the hardest task 16, and a minimum of 2.8s for task 14. Concerning the accuracy of answers provided, we observe that our system achieves 100% accuracy on all tasks. Interestingly, the encoding learnt are substantially the same as the ones provided by human experts in Yang, Ishay and Lee [106]. In conclusion, our system successfully learns and reason and provide answers over 13 commonsense driven tasks in the bAbI dataset, matching the performance of human-expert manually engineered ASP programs.

Table 6.1: Performance of compared methods in terms of accuracy.

Task	**LLM2LAS**	**Yang et al.**[106]	**Mirta et al.**[73]
1 Single supporting fact	100	100	100
6 Yes/no questions	100	100	100
8 Lists/sets	100	100	100
9 Simple negation	100	100	100
10 Indefinite knowledge	100	100	100
11 Basic coreference	100	100	100
12 Conjunction	100	100	100
13 Compound coreference	100	100	100
14 Time reasoning	100	100	100
15 Basic deduction	100	100	100
16 Basic induction	100	100	93.6
18 Size reasoning	100	100	100
20 Agents motivations	100	100	100

Chapter 7

Related Approaches

This chapter (Chapter 7), discusses the existing approaches.

The benefits and drawbacks of automated composition of programs have been studied in the literature [28], and the enormous potential of these tools is nowadays recognized [53, 78, 20]. Almost all mainstream programming languages have been supported by academic or industrial tools [14]. The benefits of providing tools for easing the development of Answer Set Programming (ASP) programs reducing the impedance mismatch existing from natural language specifications and ASP source code has been also recognized in the literature [27, 30, 88, 12]. We are not currently aware of ASP-specific tools targeting automatic program composition from natural language based on NMT as NL2ASP. Nonetheless, there are some that support inputs from more structured or simpler languages.

Baral and Dzifcak [6] automated the solving of logic puzzles in simplified English by translating their descriptions into ASP by resorting to λ-calculus and probabilistic combinatorial categorical grammars. The Controlled Natural Languages (CNLs) are subsets of the full natural languages with restricted grammar and vocabulary [57]. Several efforts have been made to develop CNLs that target ASP programs. Erdem and Yeniterzi [27] described the specific grammatical structure of a CNL named BIOQUERYCNL, as well as the algorithm for converting queries into ASP. Fang and Tompits [30] introduced a CNL approach for representing answer sets based on Language for ANnotating Answer-set programs (LANA) annotations which was implemented in the SeaLion IDE. Guy and Schwitter [88, 42, 43] developed a grammar to specify and verbalize answer set programs using a CNL named PENG^{ASP}. Kain and Tompits [52] presented a tool *Uhura* for translating problem descriptions in a controlled natural language called (L^U) into ASP rules. The proposed CNL is based on PENG^{ASP} grammar. Lifschitz [68] highlighted the connection between mathematical definitions and the knowledge representa-

tion capability of ASP. More recently, Caruso et al. [12] presented CNL2ASP, a comprehensive publicly available tool that converts controlled natural language into ASP programs. NL2ASP resorts to the CNL and tool CNL2ASP devised by Caruso et al., CNLs ease the task of programming but still require human developers. The NL2ASP tool aims at overcoming this by making the encoding process automatic.

Our approach is also related to the development of datasets for knowledge-based question answering and to other applications of ASP to natural language. Datasets that are useful for question answering were proposed by Perevalov et al. [79] that extended the Knowledge Graph Question Answering (KGQA) benchmarks QALD-9 by adding question query pairs up to 8 languages. Dubey et al. [24] introduced a comprehensive dataset for complex question answering, called LC-QuAD 2.0. This dataset includes 30,000 questions, their corresponding paraphrases, and SPARQL queries. An NMT method based on long short-term memory units was presented by Sutskever, Vinyals and Quoc V. Le [93] on an English-to-French translation task. As part of the English-to-French translation task, Bahdanau, Cho and Bengio [4] proposed an NMT module based on Bidirectional Recurrent Neural Networks. An NMT method for KBQA that automatically translates natural language in SPARQL queries was proposed by Borroto and Ricca [7] that performed well in the Question Answering over Linked Data (QALD) competition.

More recently, Nye et al. [75] presented a GPT-3 based dual-system (neural System 1 and the logical System 2) model, which generates the semantic parses from natural language sentence and couples it with reasoning modules. In this line of research, Yang, Ishay and Lee [106] proposed LLMs like GPT-3 can act as few-shot semantic parsers, converting natural language into logical forms for answer set programming. This system can address diverse question-answering tasks without distinct retraining. Ishay, Yang and Lee [50] leveraged LLMs to get the ASP with prompt engineering for solving logic puzzles. They have used the logic puzzle dataset of [73]. Although LLMs excel in System-1 processes, their results can frequently be inconsistent and incoherent. Mitra and Baral [73] proposed a heterogeneous agent model for question-answering tasks in Facebook's bAbl dataset where ASP is used for knowledge representation and reasoning. Moldovan et al. demonstrated that a logic prover can be integrated into a Question Answering system [74], where logic representations are used to transform questions and answers.

In the second part of the thesis, ILP and the most recent neuro-symbolic for machine comprehension and story-based Q&A systems.

Mitra and Baral [73] developed a three-layer Q&A system that combines statistical methods with inductive rule learning and reasoning. The system includes a Statistical Inference layer, which uses an Abstract Meaning Representation (AMR)

parser, a Translation layer, which converts the AMR parser output into Event Calculus syntax using a naive deterministic algorithm, and the Reasoner layer, which uses a modified version of the ILP algorithm XHAIL [84] to learn the knowledge required for reasoning. The system achieves on the bAbI dataset an accuracy of 99.68%, but requiring users to manually specify mode declarations and task-dependent background knowledge.

Nye et al. [75] proposed a neuro-symbolic approach to improve coherence and consistency of text generation in a story completion task. The approach uses GPT-3 to generate candidate completion sentences and an LLM-based parser to derive logical representations of a given story and generated sentences. The latter are compared to symbolic candidates inferred using a minimal world model to check consistency. Only consistent candidates are considered for the final generation. The system performs well on different benchmarks e.g. bAbI, CLUTRR, and gSCAN. However, the main limitation is the manual engineering of the world model, which is task specific.

Ishay, Yang and Lee [50] combined LLM and ASP to solve logical puzzles in a step-by-step manner. The method uses GPT-3 with prompt engineering to extract relevant objects, their categories and typed predicates from text descriptions of the puzzles. It then generates an ASP program that captures the rules of the given puzzle, using a Generate-Define-Test approach. The outcomes are computed symbolically using the generated ASP program. The method is interpretable but requires human intervention to resolve errors in the generation process.

Yang, Ishay and Lee [106] demonstrated GPT-3 to be effective in few-shot semantic parsing of natural language into ASP representation. Their approach handles Q&A tasks but with task-specific manually handcrafted background knowledge, achieving promising results on different NLP benchmarks including bAbI, StepGame, CLUTRR and gSCAN. Our approach differs from this work in that we can learn the relevant knowledge needed to solve a task.

We adopt Learning from Answer Set (LAS) to learn the knowledge needed to solve a Q&A task, thus reducing human intervention, and exploit LLM based semantic parsing capability to automatically generate LAS learning tasks from the given natural language dataset. This combination of LLM and LAS is novel and offers promising performance.

Chapter 8

Conclusion

The goal of this thesis is to take a first step towards automating the logic programs specifically answer set programming. By addressing the lack of or limited support for generating the ASP programs and the challenges of manually engineering symbolic knowledge for neuro-symbolic approaches. This thesis targets enhancing usability in ASP through existing natural language processing approaches and leveraging large language models for machine comprehension and commonsense reasoning. To address these issues, this thesis presents two main contributions.

- **NL2ASP**: A tool for translating natural language specifications into ASP programs.
  - Construction of a dataset and an approach to building the corpus, following the structure of parsing language-controlled natural language.
  - A novel two-step architecture, implemented in the NL2ASP tool, utilizing state-of-the-art neural machine translation models such as T5-small and Bart-base for text-to-text translation.
  - Empirical evidence showing that the dataset and architecture represent promising initial steps toward automated ASP program generation.
- **LLM2LAS**: A system that learns commonsense reasoning knowledge automatically using large language models and inductive logic programming.
  - The development of an open-source LLM-based few-shot semantic parser. As an OpenAI-based model like GPT is not free to use with API, so we have used open source to avoid the usage costs.
  - The development of a reasoning module, which automatically generates the ASP program based on the extracted facts using LLMs and

background knowledge according to the task, and generates the answer set.

- A learning module based on ILASP, for automatically learning commonsense knowledge needed to answer questions about the given stories. Only a few examples are required in order to learn the knowledge and answer the question.
- The evaluation of the proposed approach on the Q&A the bAbI dataset and reported 100% accuracy on 13 out of 20 tasks.

NL2ASP Concerning the automatic composition of ASP Programs, the first step is taken to automate the ASP program generation based on the user's natural language specifications. The automatic tool can boost the productivity of experts and potentially reduce adaptation barriers even in industry. In this regard, there is a need for a dataset composed of natural language specifications and intermediate language to parse statements into answer-set programming rules. We have focused on well-known graph-related problem specifications that are used to develop and assess the NLP tool's automatic composition. For the construction of the dataset, we have collected ASP programs from well-known professors in the field and did reverse engineering to build the dataset to demonstrate the viability of the proposed pipeline.

The future work involves expanding the dataset to include non-graph problems, enhancing both the dataset and system to process more natural sentences with reduced reliance on CNL2ASP constraints and explicit mention of variables, and further exploring the usage of LLM for this task.

LLM2LAS Concerning the contribution of leveraging ILP for robust question answering with LLM and ASP, the second part of the thesis proposed a novel system, called LLM2LAS that learns common-sense knowledge for machine comprehension. This system uses the open-source large language model for extracting knowledge or facts from statements, as LLM have proven semantic parsers. These facts are subsequently processed into ASP rules for reasoning. The integration of a learning module that uses ILASP to learn commonsense knowledge, and achieved high accuracy on benchmark Q&A of the bAbI dataset.

Future work involves solving the remaining bAbI tasks and making the system more general-purpose by extending the tasks done by LLMs. In addition, we intend to scale up by utilizing FastLAS for non-recursive learning and explore the use of LLMs for data processing.

In summary, this thesis presents a foundational framework for leveraging NLP and large language models to address core challenges in ASP program generation and commonsense reasoning for machine comprehension, laying the groundwork for further advancements in neurosymbolic AI.

Bibliography

[1] Hadeel Al-Negheimish, Pranava Madhyastha, and Alessandra Russo. Numerical reasoning in machine reading comprehension tasks: are we there yet? In *EMNLP (1)*, pages 9643–9649. Association for Computational Linguistics, 2021.

[2] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV2. In *LPNMR*, volume 10377 of *LNCS*, pages 215–221. Springer, 2017.

[3] Mario Alviano, Davide Cirimele, and Luis Angel Rodriguez Reiners. Introducing ASP recipes and ASP chef. In *ICLP Workshops*, volume 3437 of *CEUR WP*. CEUR-WS.org, 2023.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[5] Satanjeev Banerjee and Alon Lavie. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In *IEEvaluation@ACL*, pages 65–72. ACL, 2005.

[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 *AAAI Fall Symposium: Advances in Cognitive Systems*, volume FS-11-01 of *AAAI Technical Report*. AAAI, 2011.

[7] Manuel A. Borroto and Francesco Ricca. Sparql-qa-v2 system for knowledge base question answering. *Expert Syst. Appl.*, 229(Part A):120383, 2023.

[8] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[9] B.; Ryder N.; Subbiah M.; Kaplan J. D.; Dhariwal P.; Neelakantan A.; Shyam P.; Sastry G.; Askell A.; et al. 2020. Brown, T.; Mann. Language models are few-shot learners. In *NeurIPS*, 2020.

[10] Paula-Andra Busoniu, Johannes Oetsch, Jörg Pührer, Peter Skocovsky, and Hans Tompits. Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support. *Theory Pract. Log. Program.*, 13(4-5):657–673, 2013.

[11] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory Pract. Log. Program.*, 20(2):294–309, 2020.

[12] Simone Caruso, Carmine Dodaro, Marco Maratea, Marco Mochi, and Francesco Riccio. CNL2ASP: converting controlled natural language sentences into ASP. *Theory Pract. Log. Program.*, 24(2):196–226, 2024.

[13] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 1691–1703. PMLR, 2020.

[14] Mark Chen, Jerry Tworek, and Heewoo Jun et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[15] Xie Chen, Yu Wu, Zhenghao Wang, Shujie Liu, and Jinyu Li. Developing real-time streaming transformer transducer for speech recognition on large-scale dataset. In *ICASSP*, pages 5904–5908. IEEE, 2021.

[16] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. In *SSST@EMNLP*, pages 103–111. Association for Computational Linguistics, 2014.

[17] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734. ACL, 2014.

[18] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[19] Andrew Cropper, Sebastijan Dumancic, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30. *Mach. Learn.*, 111(1):147–172, 2022.

[20] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. Github copilot AI pair programmer: Asset or liability? *J. Syst. Softw.*, 203:111734, 2023.

[21] Ernest Davis and Gary Marcus. Commonsense reasoning and commonsense knowledge in artificial intelligence. *Commun. ACM*, 58(9):92–103, 2015.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[23] Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. Compositional semantic parsing with large language models. In *ICLR*. OpenReview.net, 2023.

[24] Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In *ISWC (2)*, volume 11779 of *LNCS*, pages 69–78. Springer, 2019.

[25] Thomas Eiter, Michael Fink, and Stefan Woltran. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Trans. Comput. Log.*, 8(3):17, 2007.

[26] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Mag.*, 37(3):53–68, 2016.

[27] Esra Erdem and Reyyan Yeniterzi. Transforming controlled natural language biomedical queries into answer set programs. In *BioNLP@HLT-NAACL*, pages 117–124. ACL, 2009.

[28] Neil A. Ernst and Gabriele Bavota. Ai-driven development is here: Should you worry? *IEEE Softw.*, 39(2):106–110, 2022.

[29] Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. Industrial applications of answer set programming. *Künstliche Intell.*, 32(2-3):165–176, 2018.

[30] Min Fang and Hans Tompits. An approach for representing answer sets in natural language. In *DECLARE*, volume 10997 of *LNCS*, pages 115–131. Springer, 2017.

[31] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. ASPIDE: integrated development environment for answer set programming. In *LPNMR*, volume 6645 of *LNCS*, pages 317–330. Springer, 2011.

[32] Mikel L. Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis M. Tyers. Apertium: a free/open-source platform for rule-based machine translation. *Mach. Transl.*, 25(2):127–144, 2011.

[33] Tiantian Gao. Controlled natural languages for knowledge representation and reasoning. In *ICLP (Technical Communications)*, volume 52 of *OASIcs*, pages 19:1–19:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[34] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASIcs*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[35] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[36] Martin Gebser, Marco Maratea, and Francesco Ricca. The seventh answer set programming competition: Design and results. *Theory Pract. Log. Program.*, 20(2):176–204, 2020.

[37] Michael Gelfond and Yulia Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, 2014.

[38] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

[39] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[40] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.*, 9(3/4):365–386, 1991.

[41] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[42] Stephen Guy and Rolf Schwitter. Architecture of a web-based predictive editor for controlled natural language processing. In *CNL*, volume 8625 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 2014.

[43] Stephen Guy and Rolf Schwitter. The PENG ASP system: architecture, language and authoring tool. *Lang. Resour. Evaluation*, 51(1):67–92, 2017.

[44] Susana Hahn, Orkunt Sabuncu, Torsten Schaub, and Tobias Stolzmann. Clingraph: A system for asp-based visualization. *CoRR*, abs/2303.10118, 2023.

[45] Hieu Hoang, Nikolay Bogoychev, Lane Schwartz, and Marcin Junczys-Dowmunt. Fast, scalable phrase-based SMT decoding. In *AMTA (1)*, pages 40–52. The Association for Machine Translation in the Americas, 2016.

[46] S Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

[47] Sepp Hochreiter. Recurrent neural net learning and vanishing gradient. *International Journal Of Uncertainity, Fuzziness and Knowledge-Based Systems*, 6(2):107–116, 1998.

[48] Ronghang Hu, Amanpreet Singh, Trevor Darrell, and Marcus Rohrbach. Iterative answer prediction with pointer-augmented multimodal transformers for textvqa. In *CVPR*, pages 9989–9999. Computer Vision Foundation / IEEE, 2020.

[49] Xinyu Hua and Lu Wang. PAIR: planning and iterative refinement in pre-trained transformers for long text generation. *CoRR*, abs/2010.02301, 2020.

[50] Adam Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs. In *KR*, pages 374–383, 2023.

[51] Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.

[52] Tobias Kain and Hans Tompits. \mathsf uhura : An authoring tool for specifying answer-set programs using controlled natural language. In *JELIA*, volume 11468 of *Lecture Notes in Computer Science*, pages 559–575. Springer, 2019.

[53] Eirini Kalliamvakou. Research: quantifying github copilot's impact on developer productivity and happiness. *The GitHub Blog*, 2022.

[54] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran,

Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *ACL*. The Association for Computational Linguistics, 2007.

[55] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New generation computing*, 4:67–95, 1986.

[56] Robert A Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[57] Tobias Kuhn. A survey and classification of controlled natural languages. *Comput. Linguistics*, 40(1):121–170, 2014.

[58] Tobias Kuhn. A survey and classification of controlled natural languages. *CoRR*, abs/1507.01701, 2015.

[59] Brenden M. Lake and Gregory L. Murphy. Word meaning in minds and machines. *CoRR*, abs/2008.01766, 2020.

[60] Mark Law. *Inductive Learning of Answer Set Programs*. PhD thesis, London, UK, 2018.

[61] Mark Law. Conflict-driven inductive logic programming. *Theory Pract. Log. Program.*, 23(2):387–414, 2023.

[62] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461, 2019.

[63] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *ACL*, pages 7871–7880. Association for Computational Linguistics, 2020.

[64] Liunian Harold Li, Mark Yatskar, Da Yin, Cho-Jui Hsieh, and Kai-Wei Chang. Visualbert: A simple and performant baseline for vision and language. *CoRR*, abs/1908.03557, 2019.

[65] Yuliya Lierler, Marco Maratea, and Francesco Ricca. Systems, engineering environments, and competitions. *AI Mag.*, 37(3):45–52, 2016.

[66] Vladimir Lifschitz. What is answer set programming? In *AAAI*, pages 1594–1597. AAAI Press, 2008.

[67] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[68] Vladimir Lifschitz. Translating definitions into the language of logic programming: A case study. In *ICLP Workshops*, volume 3193 of *CEUR WP*. CEUR-WS.org, 2022.

[69] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[70] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *ICCV*, pages 9992–10002. IEEE, 2021.

[71] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *EMNLP*, pages 1412–1421. The Association for Computational Linguistics, 2015.

[72] Sachin Mehta, Marjan Ghazvininejad, Srinivasan Iyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. Delight: Very deep and light-weight transformer. *CoRR*, abs/2008.00623, 2020.

[73] Arindam Mitra and Chitta Baral. Addressing a question answering challenge by combining statistical methods with inductive rule learning and reasoning. In *AAAI*, pages 2779–2785. AAAI Press, 2016.

[74] Dan I. Moldovan, Christine Clark, Sanda M. Harabagiu, and Steven J. Maiorano. COGEX: A logic prover for question answering. In *HLT-NAACL*. The Association for Computational Linguistics, 2003.

[75] Maxwell I. Nye, Michael Henry Tessler, Joshua B. Tenenbaum, and Brenden M. Lake. Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. In *NeurIPS*, pages 25192–25204, 2021.

[76] Christopher Olah. Understanding lstms. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015. Accessed: 2024-10-24.

[77] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, pages 311–318. ACL, 2002.

[78] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from github copilot. *CoRR*, abs/2302.06590, 2023.

[79] Aleksandr Perevalov, Dennis Diefenbach, Ricardo Usbeck, and Andreas Both. Qald-9-plus: A multilingual dataset for question answering over dbpedia and wikidata translated by native speakers. In *ICSC*, pages 229–234. IEEE, 2022.

[80] Andrzej Perzanowski. Memory and reasoning in deep learning: Data efficiency of the sam-based two-memory (stm) model, 2022.

[81] Alec Radford. Improving language understanding by generative pre-training. 2018.

[82] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[83] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[84] Oliver Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3):329–340, 2009.

[85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[86] Maarten Sap, Vered Shwartz, Antoine Bosselut, Yejin Choi, and Dan Roth. Commonsense reasoning for natural language processing. In *ACL (tutorial)*, pages 27–33. ACL, 2020.

[87] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[88] Rolf Schwitter. Specifying and verbalising answer set programs in controlled natural language. *Theory Pract. Log. Program.*, 18(3-4):691–705, 2018.

[89] Rolf Schwitter. 2. controlled languages. 2023.

[90] Sara Sheehan and Yun S. Song. Deep learning for population genetic inference. *PLoS Comput. Biol.*, 12(3), 2016.

[91] Felix Stahlberg. Neural machine translation: A review. *J. Artif. Intell. Res.*, 69:343–418, 2020.

[92] Leon Sterling and Ehud Shapiro. *The Art of Prolog - Advanced Programming Techniques*. MIT Press, 1986.

[93] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

[94] Priyansh Trivedi, Gaurav Maheshwari, Mohnish Dubey, and Jens Lehmann. Lc-quad: A corpus for complex question answering over knowledge graphs. In *ISWC (2)*, volume 10588 of *LNCS*, pages 210–218. Springer, 2017.

[95] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[96] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[97] Shubham Vatsal and Harsh Dubey. A survey of prompt engineering methods in large language models for different NLP tasks. *CoRR*, abs/2407.12994, 2024.

[98] Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating answer-set programs in lana. *Theory Pract. Log. Program.*, 12(4-5):619–637, 2012.

[99] Wiebke Wagner. Steven bird, ewan klein and edward loper: Natural language processing with python, analyzing text with the natural language toolkit - o'reilly media, beijing, 2009, ISBN 978-0-596-51649-9. *Lang. Resour. Evaluation*, 44(4):421–424, 2010.

[100] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[101] Jason Weston, Antoine Bordes, Sumit Chopra, and Tomás Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. In *ICLR (Poster)*, 2016.

[102] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus

Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[103] Yingce Xia, Tianyu He, Xu Tan, Fei Tian, Di He, and Tao Qin. Tied transformers: Neural machine translation with shared encoder and decoder. In *AAAI*, pages 5466–5473. AAAI Press, 2019.

[104] Jingjing Xu, Xuancheng Ren, Yi Zhang, Qi Zeng, Xiaoyan Cai, and Xu Sun. A skeleton-based model for promoting coherence among sentences in narrative story generation. In *EMNLP*, pages 4306–4315. ACL, 2018.

[105] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. A survey of deep learning techniques for neural machine translation. *CoRR*, abs/2002.07526, 2020.

[106] Zhun Yang, Adam Ishay, and Joohyung Lee. Coupling large language models with logic programming for robust and general reasoning from text. In *ACL (Findings)*, pages 5186–5219. ACL, 2023.

[107] Changtong Zan, Keqin Peng, Liang Ding, Baopu Qiu, Boan Liu, Shwai He, Qingyu Lu, Zheng Zhang, Chuang Liu, Weifeng Liu, Yibing Zhan, and Dacheng Tao. Vega-mt: The JD explore academy machine translation system for WMT22. In *WMT*, pages 411–422. Association for Computational Linguistics, 2022.

[108] Biao Zhang, Deyi Xiong, Jinsong Su, Hong Duan, and Min Zhang. Variational neural machine translation. In *EMNLP*, pages 521–530. The Association for Computational Linguistics, 2016.

[109] Shen Zheng, Jie Huang, and Kevin Chen-Chuan Chang. Why does chatgpt fall short in answering questions faithfully? *arXiv preprint arXiv:2304.10513*, 2023.

[110] Yibin Zheng, Xinhui Li, Fenglong Xie, and Li Lu. Improving end-to-end speech synthesis with local recurrent neural network enhanced transformer. In *ICASSP*, pages 6734–6738. IEEE, 2020.