NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RAMIRO MIGUEL RIBEIRO HENRIQUES

BSc in Computer Science

EZASP – MAKING LEARNING ANSWER SET PROGRAMMING EASIER

THESIS REPORT

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
March, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

EZASP – MAKING LEARNING ANSWER SET PROGRAMMING EASIER

THESIS REPORT

RAMIRO MIGUEL RIBEIRO HENRIQUES

BSc in Computer Science

Adviser: Matthias Knorr
Associate Professor, NOVA University Lisbon

Co-adviser: Ricardo Gonçalves
Assistant Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
March, 2024

EzASP – Making Learning Answer Set Programming Easier
Thesis Report

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.5) [21].

Acknowledgements

Firstly I would like to thank my advisers Matthias Knorr and Ricardo Gonçalves for the continuous support throughout the makings of this dissertation.

I would also like to thank the department of Computer Science and Engineering of the NOVA School of Science and Technology, and more specifically to the professors of the Artificial Intelligence and Knowledge Representation and Reasoning Systems subjects, for letting us use their students as test cases for our work.

Next, I would like to thank the professor Stefano Germano, who despite in the end us deciding not to use LoIDE, relentlessly answered every question about the the tool we had during the research phase, and even made available to us versions of code that weren't yet public.

Lastly, I want like to thank my family, girlfriend and friends who helped me in a variety of different ways throughout all of my academic journey. I am profoundly thankful to every single one of you.

Abstract

In recent years, Answer Set Programming (ASP) has gained significant traction as a logic-based method for addressing complex combinatorial problems across various domains such as management, healthcare, and engineering. Despite its effectiveness in finding potential solutions, ASP presents a steep learning curve for beginners due to its unconventional syntax, rooted in declarative programming principles unfamiliar to users accustomed to more traditional programming approaches.

To mitigate this challenge, Fandinno et al. argue that the lack of structure in ASP encodings, although highly appreciated by experienced answer set programmers, can lead to confusion for newcomers. They then propose Easy Answer Set Programming, a methodology that emphasizes the importance of structure in ASP and aims to simplify the learning process of Answer Set Programming. However, no existing technology has implemented this methodology in practice.

This dissertation fills this void by creating a Visual Studio Code extension, which builds upon two existing extensions aimed at enhancing the ASP user experience. Our extension incorporates the Easy ASP methodology, offering a practical platform for its implementation, even across multiple files.

The extension also provides a range of extra features, such as syntax checking, on-hover messages and warnings for missing comments. All the features underwent an evaluation process involving students from relevant courses, eliciting both qualitative and quantitative feedback to assess its efficacy in supporting ASP learners. Ultimately, the results of this evaluation affirmed that the Easy ASP methodology does indeed facilitate students in understanding ASP concepts.

Keywords: Answer Set Programming, Methodology, Structure, Learnability

Resumo

Nos últimos anos, Answer Set Programing (ASP) ganhou uma força significativa como um método baseado na lógica para resolver problemas combinatórios complexos em vários domínios, como a gestão, os cuidados de saúde e a engenharia. Apesar da sua eficácia na procura de potenciais soluções, a ASP apresenta uma curva de aprendizagem acentuada para os principiantes devido à sua sintaxe pouco convencional, baseada em princípios de programação declarativa pouco familiares aos utilizadores habituados a abordagens de programação mais tradicionais.

Para atenuar este desafio, Fandinno et al. argumentam que a falta de estrutura nas codificações ASP, embora seja muito apreciada por programadores de conjuntos de respostas experientes, pode gerar confusão para os recém-chegados. Propõem então a Programação Fácil de Conjuntos de Respostas, uma metodologia que enfatiza a importância da estrutura em ASP e tem como objetivo simplificar o processo de aprendizagem da Programação de Conjuntos de Respostas. No entanto, nenhuma tecnologia existente implementou esta metodologia na prática.

Esta dissertação preenche este vazio ao criar uma extensão do Visual Studio Code, que se baseia em duas extensões existentes destinadas a melhorar a experiência do utilizador ASP. A nossa extensão incorpora a metodologia Easy ASP, oferecendo uma plataforma prática para a sua implementação, mesmo em vários ficheiros.

A extensão também fornece uma série de recursos extras, como verificação de sintaxe, mensagens on-hover e avisos para regras sem comentários. Todas as funcionalidades foram submetidas a um processo de avaliação que envolveu alunos de cursos relevantes, obtendo feedback qualitativo e quantitativo para avaliar a sua eficácia no apoio aos alunos de ASP. Em última análise, os resultados desta avaliação confirmaram que a metodologia Easy ASP facilita efetivamente a compreensão dos conceitos de ASP pelos alunos.

Palavras-chave: Answer Set Programming, Metodologia, Estrutura, Aprendizagem

Contents

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Contribution . . . 3
- 1.4 Document Structure . . . 4

2 State of the Art 5
- 2.1 Answer Set Programming . . . 5
  - 2.1.1 Basic Syntax . . . 6
  - 2.1.2 Stable Models . . . 7
  - 2.1.3 Advanced Syntax . . . 8
  - 2.1.4 Sample Scenario . . . 12
- 2.2 Methodologies for ASP beginners . . . 13
  - 2.2.1 Easy Answer Set Programming . . . 13
  - 2.2.2 Achievements . . . 16
- 2.3 Tools . . . 17
  - 2.3.1 Systems . . . 17
  - 2.3.2 Text Editors . . . 18
- 2.4 Discussion . . . 19

3 Proposed Solution 22
- 3.1 Application Design . . . 22
  - 3.1.1 Syntax Checking . . . 23
  - 3.1.2 Order Errors . . . 23
  - 3.1.3 Predicate Validation . . . 24
  - 3.1.4 Missing Comment Warnings . . . 25

3.1.5 On-Hover Predicate Information . . . 25
3.1.6 Feature Disablement . . . 26
3.2 Discussion . . . 26

4 Implementation **28**
4.1 Structure . . . 28
4.1.1 Text Formatting . . . 28
4.1.2 Rule Determination . . . 29
4.1.3 Predicate Extraction . . . 31
4.2 Features . . . 31
4.2.1 Syntax Checking . . . 32
4.2.2 Order Errors . . . 32
4.2.3 Predicate Validation . . . 35
4.2.4 Missing Comment Warning . . . 36
4.2.5 On-Hover Predicate Information . . . 36
4.2.6 Feature Disablement . . . 37

5 Evaluation **39**
5.1 Methodology . . . 39
5.1.1 Test Cases . . . 39
5.1.2 Evaluation Methodologies . . . 39
5.1.3 Questionnaire Design . . . 40
5.2 Results and Analysis . . . 40
5.2.1 Syntax Checking . . . 41
5.2.2 Warnings about Order of Rules Errors . . . 43
5.2.3 Predicate Validation . . . 43
5.2.4 Missing Comment Warning . . . 43
5.2.5 On-Hover Predicate Information . . . 47
5.2.6 Feature Disablement . . . 47
5.3 Discussion . . . 50

6 Conclusion **51**
6.1 Future Work . . . 51

Bibliography **53**

List of Figures

2.1 ASP rules example. . . . 6
2.2 ASP program example. . . . 7
2.3 Choice rule example. . . . 9
2.4 Cardinality constraint example 1. . . . 10
2.5 Cardinality constraint example 2. . . . 10
2.6 Cardinality constraint example 3. . . . 10
2.7 Optimization constraints example. . . . 11
2.8 Constant declaration example. . . . 11
2.9 Show statement example. . . . 12
2.10 Illustration of the game Sudoku implemented in clingo. . . . 12
2.11 Illustration of the program presented in Figure 2.10, with choice rules transformed with the methodology illustrated in 2.2.1.3. . . . 15
2.12 Illustration of the program presented in Figure 2.11, transformed to be an Easy Answer Set Program. . . . 16
2.13 Illustration of the Achievements methodology on the N-Queens problem. . . . 17
2.14 The opening page of the LoIDE [14] editor. . . . 20
2.15 The opening page of the Visual Studio Code [30] editor. . . . 21

3.1 Example of a syntax error detected by the EZASP Visual Studio Code extension. . . . 23
3.2 Example of an order error detected by the EZASP Visual Studio Code extension. . . . 24
3.3 Example of an predicate validation error detected by the EZASP Visual Studio Code extension. . . . 25
3.4 Example of a warning message from the EZASP Visual Studio Code extension, in contrast to Figure 3.5. . . . 25
3.5 Example of an on hover message from the EZASP Visual Studio Code extension. . . . 26

4.1 A schematization of the text formatting that the EZASP extension does to every Answer Set Program. . . . 29
4.2 A schematization of the information storage of the lines that the EZASP extension does to every Answer Set Program. . . . 30

4.3 Example of an error message on an out of order constant. 33
4.4 Example of an error message on an out of order fact. 33
4.5 Example of a program without a definition and no error message. 33
4.6 Example of a choice in between a block of definitions. 34
4.7 Example of an error message where a constraint is preceding a choice rule. 34
4.8 Pseudo-code for Calculating Order Errors. 35
4.9 Example of a config.json file created from the ASP Language Support extension. 36
4.10 Example of On-Hover Predicate Information feature, showing predicates that are being defined in different a file. 37
4.11 Example of a config.json file created from the EZASP extension. 38

5.1 Example of a quantitative question. 41
5.2 Example of a qualitative question. 41
5.3 A visual representation of the results of the questionnaire for the syntax checking feature. 42
5.4 A visual representation of the results of the questionnaire for the warnings about order of rule errors feature. 44
5.5 A visual representation of the results of the questionnaire for the predicate validation feature. 45
5.6 A visual representation of the results of the questionnaire for the missing comment warning feature. 46
5.7 A visual representation of the results of the on-hover predicate information feature. . . 48
5.8 A visual representation of the results of the evaluation on the disablement of features. . 49

List of Tables

2.1 Truth Table for program P 7

1

Introduction

In this chapter, we present an overview of this dissertation by providing context, discussing the motivation behind our work, and outlining its development process.

1.1 Context

Declarative programming is an approach to computer programming which emphasizes the declaration of what the program should accomplish, leaving the details of how to accomplish it to the underlying system. It contrasts with imperative programming, which focuses on prescribing the steps the program should take to achieve its goal, with the programmer providing a detailed specification of the solution.

Answer Set Programming (ASP) [8] is a declarative problem solving approach, where the problems are represented logically, and then answer set solvers, programs made to generate the solutions of the problem, calculate the desired answer sets. The formal roots of ASP lie in Logic Programming [20], a programming paradigm based on formal logic. ASP has a clear and logical format, characterized by the absence of any imposed order in the code. This, in combination with the declarative nature of ASP results in highly compact and simple encodings. On top of this, over the years developers have created very efficient answer set solvers, also referred to as ASP systems, using highly optimized algorithms and efficient data structures designed to handle problems with high complexity. Clingo [12] and DLV [3] are recognized as the current state-of-the-art systems, known for their robustness, efficiency and versatility.

ASP's benefits are demonstrated in numerous examples, such as the work of Ricca et al. in [27], where they present a case study highlighting how ASP can improve scheduling and management with the personnel for serving incoming ships in an efficient manner, in the context of a seaport. Similarly, in [1] Alviano et al. show how ASP can be applied to various healthcare domains, such as clinical decision making, resource allocation, and patient care management. The article highlights the benefits of using ASP in healthcare, including increased accuracy, efficiency, transparency, and accountability in decision making. Other examples can be seen in [9] where Falkner et al. present various case studies and real-world examples of how ASP has been applied to solve complex problems in industries such as engineering, logistics, and planning.

1.2 Motivation

Even though Answer Set Programming is a very useful tool, its learning curve is not ideal. The freedom of order in its encodings, regardless of it being highly appreciated by ASP experts, can cause confusion in newcomers given its declarative nature.

To address this issue, Fandinno et al. in [10] defend that for beginners it is advantageous to have structure in their encodings and present a methodology, Easy Answer Set Programming. This methodology promotes the separation of programs into distinct and manageable units, and by improving its structure and documentation, the code becomes easier for the programmer to understand and maintain. This modular approach to ASP enhances the overall structure and organization of the program, making it more intuitive and efficient for the programmer to navigate and make changes. Fandinno et al. [10] argue that adopting this structured approach can greatly benefit individuals who are learning to work with ASP.

Another technique that can significantly increase productivity in any programming approach, and particularly for those utilizing Answer Set Programming, is the proper use of comments in their encodings. On account of this, Vladimir Lifschitz in [19] introduces a possible commenting methodology where the programmer explains in each comment what was achieved with that line of code, which can serve as a tool newcomers use to better understand their answer set programs.

Similarly to other programming languages, the initial step to develop an answer set program involves writing the code in a text file, followed by processing it through a solver system, which can be run either locally or online, depending on the editor. To facilitate this process, throughout the years various text editing tools of different types have been created to assist ASP users. Let's explore a few examples:

- **Online code editors:** This option is very easily accessible, without any need for installation and can run on any device that has internet connection. However it has the limitations of running the solver online, as running encodings could take a long time depending of the server's resources, and if the browser is refreshed, the code will be lost. It can also provide visual support to the user by employing syntax highlighting, assigning distinct colors to each word based on its type, facilitating a clearer understanding of the code structure and syntax. Examples of online code editors are the website [24] owned by the clingo creators, at the University of Postdam, or the project LoIDE [14], a web-based IDE (Integrated Development Environment) for Logic Programming created by the Department of Mathematics and Computer Science of the University of Calabria, that can run both Clingo and DLV.

- **Integrated Development Environments (IDEs)**: The users can install the clingo [26] or DLV [28] releases, locally on their devices and utilize a text editor of choice while executing the encodings through the terminal. This option does not have the problems of it being run on a website, but depending on the chosen IDE it most likely would not have visual help to the user. In addition, it can be hard for inexperienced programmers to understand how to work with these installations. There are IDEs designed explicitly for ASP, such as the SeaLion tool [6], or even plugins to more general IDEs providing support for ASP systems. For example, the IDE Visual Studio Code [30] has two extensions for clingo [11]. One has the syntax highlighting similar to the online editors

described above, does not need an installation of the clingo release, and the other extension incorporates a solver directly into the IDE, and is running the encodings locally which creates no efficiency problems regarding the compilations. This IDE also has an extension for DLV2 [7] which also provides most of the functionalities described for its clingo counterpart.

Despite their differences, all these tools share one common characteristic: they fail to provide new users with support for structuring their programs or incentivize regular commenting of code. This absence of support calls for the implementation of a tool with these characteristics.

1.3 Contribution

In this dissertation we investigated how the methodologies presented above can be incorporated into existing text editing tools, and implemented them, with a specific focus on ASP beginners. The dissertation had these goals:

- Enhance a text editor to support the Easy Answer Set Programming methodology. This entails reinforcing the concept of structure in Answer Set Programming (ASP) and guiding users towards good practices.
- Assist users in discerning the appropriate moments for integrating comments into their encodings, incentivising effective commenting practices.
- Explore and implement functionalities within the text editor that contribute to the pedagogical aspects of learning Answer Set Programming, ensuring that the tool serves as an educational resource for users.
- Evaluate the effectiveness of the enhanced text editor in fostering a better understanding of ASP concepts among users.

To accomplish these goals, we developed a Visual Studio Code [11] extension. To this end, we determined the best approach for implementing these methodologies and identifying the specific features to incorporate into the extension, which materialized in various localized warnings and error messages, as they offer users an accessible means of understanding issues within each specific line of code in their projects, being crafted to address the objectives of our project.

This extension is available in the Visual Studio Code marketplace in [17], and its source code is accessible on its respective GitHub page in [16].

Lastly, we tested the effectiveness of the developed extension in three different sub-scales, efficiency, helpfulness and learnability. This evaluation was conducted with students from the courses of Artificial Intelligence and Knowledge Representation and Reasoning Systems, where after using the extension, the students provided quantitative an qualitative feedback by participating in a structured questionnaire. We examined the collected data and made a qualitative evaluation of each of our individual features, which resulted on a positive outcome on the Easy ASP methodology, but also showed a resistance by the students on using good commenting practices.

1.4 Document Structure

We now proceed to list the chapters that outline the structure of this document:

- **Chapter 2 - State of the Art:** This chapter dives into what is Answer Set Programming, the methodologies used in this dissertation, and what text editors exist for ASP;
- **Chapter 3 - Proposed Solution:** In this chapter we present the choices we had to make across this dissertation, and explain them;
- **Chapter 4 - Implementation:** We explain the process of implementing each feature and what difficulties we had to overcome;
- **Chapter 5 - Evaluation:** Here we present how the work was tested, and explain the results, providing context on which features were more and less sucessful;
- **Chapter 6 - Conclusion:** Lastly, we present a conclusion to this dissertation, and present a possible future work to extend upon this thesis.

2

State of the Art

In this chapter, we present an in-depth analysis of the most relevant concepts to this thesis.

We start by introducing what is Answer Set Programming in section 2.1, explaining its syntax in 2.1.1, what are stable models and how are they computed in 2.1.2, and illustrate the system's language expressions necessary for its full functionality in 2.1.3. Subsequently, in 2.1.4 we show a practical example, by demonstrating and dissecting an implementation [5] of the popular game Sudoku [29].

Next, in section 2.2, we discuss some methodologies created to facilitate the usage of answer set programming to newcomers. We start by analyzing what is the core methodology for this proposed dissertation, Easy Answer Set Programming in 2.2.1, then we will more briefly analyse what are Achievements in Answer Set Programming in 2.2.2.

We then show existing tools for creating Answer Set Programs, displaying their functionalities and their inherent limitations, creating an opportunity for the development of a new tool, in 2.3.2. The chapter concludes with a discussion in section 2.4 on the selection of the tool we opted to employ for developing our product.

2.1 Answer Set Programming

> *Answer set programming (ASP) is a declarative problem solving approach, initially tailored to modeling problems in the area of knowledge representation and reasoning (KRR). More recently, its attractive combination of a rich yet simple modeling language with high-performance solving capacities has sparked interest in many other areas even beyond KRR.*
>
> ***Answer Set Solving in Practice [13]***

As briefly discussed in 1.1, ASP provides a way of expressing and solving problems using declarative logic conditions. These programs can always be separated into two groups: facts, and rules. The facts provide information about the problem, being no more than atomic propositions that can be positive or negative. The rules consist of a set of statements used to express the relationships between the facts in a problem domain, and to define constraints on the problem. It is common practice in ASP to split

the logic program into separate files for facts and rules. This allows for flexible and efficient encoding of various problem instances.

2.1.1 Basic Syntax

Firstly, let us define the elements on an ASP rule. These elements can be terms and predicates. Terms can be constants or variables, and are strings that represent specific objects in the case of constants, and various possible objects in the case of variables. Predicates are used to represent relationships between objects and properties in a problem domain. With predicates and terms, we can create atoms, from a predicate and a set of terms, with the number of terms being equal to the arity of the atom.

Predicates and constants are represented by lowercase letters, like a, or strings starting with a lowercase character, like one. In contrast, variables are represented by uppercase letters like B, or strings starting with an uppercase character, like Two.

Given this, an Answer Set Program is a set of rules of the form[1]:

$$a \text{ :- } b_1, ..., b_m, not\ c_{m+1}, ..., not\ c_n.$$

Where a and all b_i, for $i \in \{1, ..., m\}$, and c_j, for $j \in \{m+1, ..., n\}$ be atoms. Here, a is considered the head of the rule, and all b_i and all $not c_j$ are the body of the rule. The symbols in the rule have logical translations:

- The symbol ':-' [2] can be read as *if*;
- The symbol ',' can be read as *and*;
- The symbol 'not' denotes *negation*.

In other words, the rule above can also be read as 'a if b_1 and ... and b_m, and not c_1 and ... and not c_n'. Every rule must also end with a period, signifying its conclusion.

Additionally, we define a negation-free normal rule as definite, as seen in [10]. This rule will take the following form, a variation of the rule shown above:

$$a \text{ :- } b_1, ..., b_m$$

Let us now look at an example of a program, in Figure 2.1:

```
cappuccino :- beverage(a), has(a,coffee), has(a,milk).
icedtea :- beverage(a), has(a,tea), has(a,ice).
```

Figure 2.1: ASP rules example.

Here, the predicate cappuccino is only true in the system if the atoms beverage(a), has(a,coffee) and has(a,,milk) are also true, and the predicate icedtea is only true if the atoms beverage(a), has(a,tea) and has(a,ice) are also true.

[1] The contents and definitions on these following sections are derived from the presentation in [13].

[2] The symbol ← is a more commonly used notation in ASP systems, but for the purpose of simplicity and consistency, the : - notation will be used throughout this document.

Additionally, rules without body are facts in the program. This means that the head in these rules is always considered true, and the : - symbol can be omitted from them. For example, in the program showin in Figure 2.2, both predicates cappuccino and icedtea are true, because in this scenario flavouring(sugar) and flavouring(honey) are facts, and both cappuccino and icedtea are true with any possible flavouring, and the predicates elements(coffee,milk) and elements(tea,ice) are also facts.

```
1. flavouring(sugar).
2. flavouring(honey).
3. elements(coffee,milk).
4. elements(tea,ice).
5. cappuccino :- elements(coffee,milk), flavouring(X).
6. icedtea :-  elements(tea,ice), flavouring(X).
```

Figure 2.2: ASP program example.

We are now well-equipped to understand most basic syntax of Answer Set Programming, as there are only small remaining details to cover. The underscore symbol (_) is used to denote anonymous variables, variables which the corresponding values are not important in the rule. ASP also supports integer operations such as addition (+), negation (−), multiplication (∗), and integer division (/). It also allows for comparison between integers, including equality (==), inequality (! =), greater than (>), greater or equal than (>=), less than (<), and greater or equal than (<=). Additionally, a compact representation of a range of integers can be written as $q..n$, where q is less than n. Another detail to note is that facts of the same predicate can be represented in the same instance with a semicolon between them to save space. For example, the lines 1 and 2 of the program above could also be represented in one line with flavouring(sugar;honey).

2.1.2 Stable Models

In ASP, the program's outcomes are established by its stable models, which, in turn, originate from the classical models of the program. These classical models, also referred here as candidate models, are rooted in classical logic. To illustrate, let's consider the following program:

$$P = \{a \text{ : - } a., b \text{ : - } not\ a.\}$$

To calculate the candidate models of this example we convert the program to classical logic, and put together its truth table, that we can see in Table 2.1.

a	**b**	$(a \rightarrow a) \wedge (\neg a \rightarrow b)$
F	F	F
F	T	T
T	F	T
T	T	T

Table 2.1: Truth Table for program P

After a quick examination of 2.1, we can see that the candidate models of P are $\{a\}$, $\{b\}$ and $\{a, b\}$.

For us to define how to calculate the stable models of a program, we need to introduce a few concepts.

Firstly, we define the minimal model of a program. This models is such that no subset of its atoms is also a model of the same program. It means that the model is the smallest possible solution that satisfies all rules in the program and is the model that fully characterizes the problem and its solution.

Additionally, we must also introduce the concept of a body of a rule being positive and negative, represented by $body(r)^+$ and $body(r)^-$. $body(r)^+$ is the set of positive elements and $body(r)^-$ is the set of negative elements of the body of a rule. Using P as an example:

$$body(a \text{ :- } a)^+ = \{a\},\ body(a \text{ :- } a)^- = \emptyset$$

$$body(b \text{ :- } not\ a)^+ = \emptyset,\ body(b \text{ :- } not\ a)^- = \{a\}$$

Then, the reduced program of a program P, relative to a set of predicates X, is defined by:

$$P^X = \{head(r) \text{ :- } body(r)^+ \mid r \in P, body(r)^- \cap X = \emptyset\}$$

P^X is always a positive program (a program with only positive atoms in its rules), and consequently has always only one minimal model.

Given this, if this minimal model is equal to X, X is a stable model of P. Let us now calculate program P's stable models.

$$P^{\{a\}} = \{a \text{ :- } a.\}$$

$$P^{\{b\}} = \{a \text{ :- } a.\ , b.\}$$

$$P^{\{a,b\}} = \{a \text{ :- } a.\}$$

The minimal model of $P^{\{a\}}$ is {}, the minimal model of $P^{\{b\}}$ is $\{b\}$, and the minimal model of $P^{\{a,b\}}$ is {}. Since the minimal model of P^X is equal to X only if $X = \{b\}$, $\{b\}$ is the only stable model of P.

2.1.3 Advanced Syntax

Having established a foundational understanding of the basic syntax and fundamental components of an Answer Set Program, the focus now shifts to more advanced components of Answer Set Programming. Here we explain the all the different categories of rules, and their respective applications.

2.1.3.1 Integrity Constraints

Integrity constraints are rules of the form:

$$\text{:- } a_1, ..., a_m, not\ a_{m+1}, ..., not\ a_n.$$

These rules are meant to eliminate undesirable solution candidates. For example, the rule :- a removes all stable models of the program that the atom a belongs to. These constraints can also be mapped into a normal rule, if the symbol in this rule's head is a new symbol in the system, taking the following form, given that x is an atom not previously defined.

$$x \text{ :- } b_1, ..., b_m, not\ c_{m+1}, ..., not\ c_n, not\ x.$$

If we take the example used above, the rule :- a, would be equivalent to the rule x: - $a, not\ x$.

2.1.3.2 Choice Rules

Choice rules are rules of the form:

$$\{a_1, ..., a_m\} :- a_{m+1}, ..., a_n, not\ a_{n+1}, ..., not\ a_o.$$

These rules allow for indicating choices and every possible variation of atoms on the head of the rule will be a stable model, if the body of the rule is true. For instance, the program:

$$P = \{a., \{b; c\} :- a.\}$$

will have $\{a\}$,$\{a, b\}$,$\{a, c\}$, and $\{a, b, c\}$ as stable models. It is notable that the choice rule only generates these models because a is given as a fact.

2.1.3.3 Conditional Literals

A conditional literal is given by the form: $l : l_1, ..., l_n$, for $0 \leq i \leq n$.

This is used to iterate through the list of possible elements in these atoms. For example, given the facts color(red), color(green) and color(blue), paint(C):color(C) is equivalent to paint(red), paint(green), paint(blue).

This can be useful for choice rules, as we can slightly change the example above into the one in Figure 2.3.

```
1. beverage(tea;coffee).
2. flavouring(sugar;milk).
3. {drink_with(X,Y):  beverage(X), flavouring(Y)}.
```

Figure 2.3: Choice rule example.

The program here will generate all sixteen possible stable models by utilizing the predicate drink_with/2, which will have every possible pair of beverage and flavouring.

2.1.3.4 Cardinality Constraints

Cardinality constraints are rules of the form:

$$k\{a_1, ..., a_m : a_{m+1}, ..., a_n\}l :- a_{n+1}, ..., a_o, not\ a_{o+1}, ..., not\ a_p.$$

Or, they can also appear in the body, taking the subsequent form:

$$a_o :- k\{a_1, ..., a_m : a_{m+1}, ..., a_n\}l$$

On a first glance, they are very similar to choice rules, since the only difference between the form with a cardinality constraint on the head and the one seen in 2.1.3.2 is the k and l integers on each side of the braces. This indicates that each stable model can only have a number of positive head predicates that is equal or greater than k, and equal or less than l. For example, lets look into the Figures 2.4 and 2.5.

The two equivalent programs will restrict the sixteen stable models seen above, to only the five possible models. One where the predicate drink_with/2 does not occur, and four where all possible

```
1. beverage(tea;coffee).
2. flavouring(sugar;milk).
3. 0{drink_with(X,Y): beverage(X), flavouring(Y)}1.
```

Figure 2.4: Cardinality constraint example 1.

```
1. beverage(tea;coffee).
2. flavouring(sugar;milk).
3.{drink_with(X,Y): beverage(X), flavouring(Y)}.
4.:- not 0{drink_with(X,Y): beverage(X), flavouring(Y)}1.
```

Figure 2.5: Cardinality constraint example 2.

values of its variables are met, these being the pairs (coffee,sugar), (coffee,honey), (tea,sugar), and (tea,honey).

It is also notable that cardinality constraints, in each side of the rule, can also take the form of:

$$\{a_1; \ldots; a_m : a_{m+1}, \ldots, a_n\} = j$$

This form is used only when we want the number of head predicates in the stable models to be equal in all stable models. This form is equivalent to the form above, and if $k = l$. For example, let us consider the program in figure 2.6:

```
1. beverage(tea;coffee).
2. flavouring(sugar;milk).
3. {drink_with(X,Y):  beverage(X), flavouring(Y)} = 1.
```

Figure 2.6: Cardinality constraint example 3.

Compared with the program in 2.3, this program only has 4 stable models, as all pairs of flavouring stay the same, but the model without the head predicate drink_with/2 is removed.

2.1.3.5 Optimization Statements

Answer Set Programming also allows for the creation of optimization of results, either for minimization or maximization. For the number of answer sets available, the statement will either minimize or maximize the value in the variable in hand. These statements have the following format:

$$\#minimize\{w_1@p_1, t_1 : L_1, \ldots, w_n@p_n, t_n : L_n\}.$$

$$\#maximize\{w_1@p_1, t_1 : L_1, \ldots, w_n@p_n, t_n : L_n\}.$$

Here, in figure 2.7, w represents a given weight, p represents the priority of the value. When there are different priorities attached to tuples, a value is obtained for each priority, and then the tuples are compared based on the order of priorities. If omitted, priority is considered 0. Considering all of this, a set of predicates is considered the optimal answer set if the weight of the values is minimal or maximal depending on the statement, among all answer sets of the given program. It is also notable that the maximize function works in a similar way as the minimize function, but with the weights negated, so when computing a maximize optimization, the value optimized will be negated. For example, Figure 2.7:

```
1. beverage_preference(coffee,3;tea,2;juice,1).
2. {choice(B): beverage_preference(B,_)} = 1.
3. #maximize {P@1,B: choice(B), beverage_preference(B,P)}.
```

Figure 2.7: Optimization constraints example.

Here, as we can see in line 1, we attribute a number to all the beverages, representing preference where a greater number represents more preference. Then, in line 2, we generate all answer sets of possible beverages to choose, with the head predicate, choice/1. Finally, in line 3, we calculate between all answer sets with the weight of the preferences, which beverage is preferred the most, which results in the answer set with the predicate choice(coffee), with an optimization value of -3.

2.1.3.6 Constant Declarations

Answer Set Programming also allows the declaration of constants using the $\#const$ statement, as illustrated with an example here:

$$\#const\ n = 1.$$

Here, the keyword #const is followed by the constant's name (in this case, n) and its assigned value. This statement signifies that n is a constant with a fixed value of 1 throughout the program. This type of rule is great for having a program where a number can quickly be changed if necessary. For example, in Figure 2.8 we can see a program that uses this rule statement, where all beverage preferences are being calculated. However, if we only wanted to generate two possibilities, we just needed to change the value of n to 2.

```
1.#const n = 1.
2.beverage_preference(coffee,3;tea,2;juice,1).
3.{choice(B): beverage_preference(B,_)} = n.
```

Figure 2.8: Constant declaration example.

2.1.3.7 Show Statement

Answer Set Programming also allows the answer sets to be filtered, so that one predicate is shown. For this, a rule has to be written as follows:

$$\#show\ x/y.$$

Here, the keyword #show is followed by x and y, where x represents a predicate name, and y the number of arguments of x. For example, if we look at the program illustrating constant declarations, shown in Figure 2.8, this program has three stable models, each with an atom for the predicate choice/1, and the facts of the program, all three instances of beverage_preference/2. However, in the program showed in Figure 2.9, the program will still have the same three models, but these only contain the predicate choice/1, specifically the models choice(coffee), choice(tea) and choice(juice).

```
1.#const n = 1.
2.beverage_preference(coffee,3;tea,2;juice,1).
3.{choice(B): beverage_preference(B,_)} = n.
4.#show choice/1.
```

Figure 2.9: Show statement example.

2.1.4 Sample Scenario

Let us now apply the information discussed in this chapter so far. To do this, we will analyse the example in Figure 2.10, an implementation [5] of the well-known game Sudoku [29]. Sudoku is a game with a 9×9 grid, with some numbers already filled in. The objective is to fill the rest of the grid, so that each column, row, and each of the nine 3×3 subgrids contain all digits from 1 to 9, without repetitions.

```
1.  % positions
2.  n(1..9).
3.
4.  % each position has exactly 1 number
5.  1{in(X,Y,N):n(N)}1 :- n(Y), n(X).
6.
7.  % for each column, a number only occurs in one row
8.  1{in(X,Y,N):n(Y)}1 :- n(X), n(N).
9.
10. % for each row, a number only occurs in one column
11. 1{in(X,Y,N):n(X)}1 :- n(Y), n(N).
12.
13. % defining a subgrid
14. subgrid(X,Y,Z,W) :- n(X), n(Y), n(Z), n(W), (X-1)/3=(Z-1)/3,(Y-1)/3=(W-1)/3.
15.
16. % for each subgrid, a number only occurs once
17. :- subgrid(X,Y,Z,W), in(X,Y,N), in(Z,W,N), X!=Z, Y!=W.
18.
19. #show in/3.
```

Figure 2.10: Illustration of the game Sudoku implemented in clingo.

Firstly, it should be understood that this code alone is not the full program. The facts representing the filled cells, are hidden, and with this, the majority of the facts of the program are omitted with the exception of line 2, as it also represents a fact. This is due to it being common practice in ASP to separate the files with the facts that represent different instances of the problem, and the rules that solve the problem. These omitted facts, in this example, represent the spaces in the grid which are already filled with numbers in the beginning of a Sudoku game. These numbers are represented by the predicate $in/3$. On the predicate $in(x, y, n)$, n corresponds to the numeric value in the cell with the coordinates (x, y). Given any instance of the predicate as a fact, it locks the number into the coordinate in the grid.

Upon examining the first line of code, we quickly discover an unmentioned detail. The % symbol signifies that the line it appears in is a comment. As evident in lines 1, 4, 7, 10, 13, and 16, these comments provide explanations for the code below each one.

Moving on to line 2, in we observe some facts. This line creates nine instances of the predicate $n/1$, and decides that the board in this encoding will be a 9×9 grid. Because of the subgrid, this implementation was created only to support grids with those dimensions, but another encoding could simply take facts with any grid (as long as the subgrids are still squares) and respective instance of numbers, and solve it.

Now we generate every possible board, without constraints, with the code in line 5. The head of the rule consists of a choice rule, that will generate one answer set for each possibility of number n. Then, the body of the rule states that one instance of the predicate $in/3$ will be generated for every possible pair of $n(Y)$ and $n(X)$.

The rule in line 8 is similar to the one in line 5, but N is switched with the Y. Since every possible board has already been generated, this rule will restrict only the possible boards, stating that for every pair of X and N only one value of Y can exist, meaning that in every column only one number can occur. In line 11, the same concept is applied but to columns instead.

Now, most of the game is defined. The only game restrictions left to implement are the subgrids, and that's what lines 14 and 17 do. In line 14, the predicate $subgrid/4$ is created, where it stores two coordinates. The equations $(X-1)/3 = (Z-1)/3$ and $(Y-1)/3 = (W-1)/3$ ensure that those two stored coordinates are only coordinates of the same grid, and therefore the predicate subgrid/4 will contain one instance for every pair of coordinates in a subgrid. Then, in line 17, an integrity constraint is used, where it guarantees that for every subgrid, the number in the coordinates cannot be the same.

Lastly, line 19 has a show statement with the lone purpose of removing all predicates from the stable models, except the predicate in/3, showing the solution of the problem.

2.2 Methodologies for ASP beginners

Answer Set Programming can be challenging to learn, especially for those without prior experience in logic programming or computer science. To address this issue, over the years many ways to help these users have been created, from programming methodologies to interactive text editors. In this section, we explore some of these methodologies.

2.2.1 Easy Answer Set Programming

In ASP, the programmer is free to write rules in any order, which can be an advantage for experts of this approach, but for beginners it can lead to disorientation. With this in consideration we discuss Easy Answer Set Programming [10], a methodology to tackle this issue. This methodology uses the "Here-and-There" logic [4], a monotonic logic for ASP, to show that any logical formula can be broken down into a combination of sub-formulas, serving as the basis for the methodology, as any answer set program can subsequently be converted into a program that follows a certain order, an easy answer set program.

2.2.1.1 Austere ASP

Fandinno et al. in [10] also shows that every logic program can be expressed in the form of an austere logic program, a quadruple (F,C,D,I) consisting of a set F of facts, a set C of choices, a set D of definite

rules, and a set I of integrity constraints.

The semantics of austere logic programs follow the methodology of generate-define-test, a strategy where the solutions are generated, and then tested to see if the proposed solution meets conditions of the problem. The choices C generate a set of facts F_C. Let C' be one set of guessed choices over F_C, the logic program $F \cup C' \cup D$ has an unique stable model. It should be noted that the I is omitted because since the program $F \cup C' \cup D$ has a unique model, adding constraints would either have no effect or potentially remove its unique stable model.

2.2.1.2 Easy ASP

The principles provided by austere logic programs are used to create a methodology for ASP that aims at people who are new to this technology. For this, we first need to understand the definition of a dependency in ASP.

In a program P, a predicate p depends on another predicate q if there is a rule in P with with p in the head of the rule and q in the body. Also, if p depends on q, and q depends on r, p also depends on r. As such, the definition of a predicate p is the subset of P consisting of all rules with p on its head.

Let us consider the definition of a predicate symbol p in P be defined by $def(p)$. As such, a partition $(P_1, ..., P_n)$ of a program of P is defined as a stratification of P if:

- For each predicate p, $def(p) \subseteq P_i$, with $i \in \{1, ..., n\}$;
- Let two predicates p and q so that p depends on q, and $def(p) \subseteq P_i$, $def(q) \subseteq P_j$ with $i, j \in \{1, ..., n\}$, if:
  - q does not depend on p, then $i > j$;
  - q depends on p, then $i = j$.

In essence, a stratification represents a partition within an organized program. Here, each rule is systematically grouped in the correct sequence, and a predicate is employed only if it has been previously defined in the program.

Having a stratification of a program defined, we can also define an easy logic program as a logic program having stratification $(F, C, D_1, ..., D_n, I)$, such that F is a set of facts, C is a set of choice rules, D_i is a set of normal rules for $i = 1, ..., n$, and I is a set of integrity constraints.

2.2.1.3 Methodology

Given what was discussed in 2.2.1.1, we can conclude that given an easy logic program $(F, C, D_1, ..., D_n, I)$, and a set of choices C' contained in F_C, the logic program $F \cup C' \cup D_1 \cup ... \cup D_n$ has only one stable model.

With this, Fandinno et al. in [10] define a methodology for ASP programmers to follow:

- Separate the logic program into facts F, choice rules C, normal rules $D_1, ..., D_n$, and integrity constraints I, such as the logic program has the stratification $(F, C, D_1, ..., D_n, I)$;
- Define one predicate for each D_i ensuring that there are no dependency cycles (p depends on q then q cannot depend on p);

- Guarantee that every $F \cup C' \cup D_1 \cup ... \cup D_n$ has a unique stable model for all C' contained in F_C.

While the first two steps of this methodology can be checked syntactically, the third is not as direct. In this case, we can restrict all $R_1 \cup ... \cup R_n$ to definite rules, or use stratified negation. Stratified negation is used to remove recursion through negation, which can harm programs by creating incoherence in the stable models. To ensure that this is not a problem, the algorithm presented in 2.2.1.2 needs to be extended:

- Let r be a rule in P_i, if $p \in body(r)^-$, then $def(p) \subseteq P_j$, for some $j < i$.

This step ensures that if a rule contains any negated predicates, then these predicates definitions will be done in a lower partition than this rule. This ensures the elimination of recursion thought negation. It is important to note that by definition, austere logic programs, and by consequence easy answer set programs, do not permit cardinality constraints in the form studied. Because of this, the program example given in Figure 2.10 cannot be considered an illustration of this methodology.

Despite this, it is important for Fandinno et al. to cover the full expressiveness of ASP. So, they show that it is possible to convert cardinality constraints in choice rules into rules compatible with this methodology. They state that a set of choice rules, of the form $a\{l_1, ..., l_m\}b :- l_{m+1}, ..., l_n$, can be represented as the two following rules:

$$\{l_i\} :- l_{m+1}, ..., l_n.$$
$$:- not\ a\{l_1, ..., l_m\}b, l_{m+1}, ..., l_n.$$

Fandinno et al. also add that this decomposition is done by ASP solvers anyway, and therefore this separation will bring no performance loss.

```
1.  % positions
2.  n(1..9).
3.
4.  % each position has exactly 1 number
5.  {in(X,Y,N)} :- n(N), n(Y), n(X).
6.  :- not 1{in(X,Y,N):n(N)}1, n(Y), n(X).
7.
8.  % for each column, a number only occurs in one row
9.  {in(X,Y,N)} :- n(Y), n(X), n(N).
10. :- not 1{in(X,Y,N):n(Y)}1, n(X), n(N).
11.
12. % for each row, a number only occurs in one column
13. {in(X,Y,N)} :-n(X), n(Y), n(N).
14. :- not 1{in(X,Y,N):n(X)}1, n(Y), n(N).
15.
16. % defining a subgrid
17. subgrid(X,Y,Z,W) :- n(X), n(Y), n(Z), n(W), (X-1)/3=(Z-1)/3,(Y-1)/3=(W-1)/3.
18.
19. % for each subgrid, a number only occurs once
20. :- subgrid(X,Y,Z,W), in(X,Y,N), in(Z,W,N), X!=Z, Y!=W.
```

Figure 2.11: Illustration of the program presented in Figure 2.10, with choice rules transformed with the methodology illustrated in 2.2.1.3.

We can now apply this methodology onto the program in Figure 2.10, replacing the lines 5, 8, and 11, with the result being the program in Figure 2.11.

```
1.  % positions
2.  n(1..9).
3.
4.  %every possible atom for in/3 is generated
5.  {in(X,Y,N)} :- n(N), n(Y), n(X).
6.
7.  %defining a subgrid
8.  subgrid(X,Y,Z,W) :- n(X), n(Y), n(Z), n(W), (X-1)/3=(Z-1)/3,(Y-1)/3=(W-1)/3.
9.
10.  % each position has exactly 1 number
11.  :- not 1{in(X,Y,N):n(N)}1, n(Y), n(X).
12.
13.  % for each column, a number only occurs in one row
14. :- not 1{in(X,Y,N):n(Y)}1, n(X), n(N).
15.
16. % for each row, a number only occurs in one column
17. :- not 1{in(X,Y,N):n(X)}1, n(Y), n(N).
18.
19. % for each subgrid, a number only occurs once
20. :- subgrid(X,Y,Z,W), in(X,Y,N), in(Z,W,N), X!=Z, Y!=W.
```

Figure 2.12: Illustration of the program presented in Figure 2.11, transformed to be an Easy Answer Set Program.

In fact, by doing this, the lines in 5, 9 and 13 are now equal, and therefore the program in Figure 2.11 can be reduced, as seen in Figure 2.12. It must also be noted that to follow the Easy ASP methodology, the order of the rules was changed so that this program can accurately describe an Easy Answer Set Program.

Now, this example is an Easy Answer Set Program where F is representing the facts omitted in the encoding, as explained in 2.1.4, $F \cup \{2\}$ is the set of facts, $\{5\}$ the set of choices, $\{8\}$ the set of definite rules, and $\{11, 14, 17, 20\}$ is the set of integrity constraints, where the numbers represent the correspondent line of code. As such, the program in Figure 2.11 has the stratification $(F \cup \{2\}, \{5\}, \{8\}, \{11, 14, 17, 20\})$.

2.2.2 Achievements

Vladimir Lifschitz in [19] defines achievements in the process of writing an ASP program as the fact that the programmer has achieved what he intended with one single executable piece of code, and will commit to maintain it.

This definition opens up the of idea of a methodology that, for every achievement, the user should record it, and explain what the intended code is responsible for. Lifschitz also puts forward the idea of explaining what the code should do before even implementing it, and after a successful implementation declare the feature as achieved. To illustrate this let us take a look into Figure 2.13 [19], where a solution of the N-Queens problem[3] is presented, with the methodology described.

Just by reading the comments in each line, we can understand the encodings in a much more compreehensive way, and therefore this methodology represents a complete, detailed record of the achievements, and can be a solution for the better understanding and development of Answer Set Programs.

[3]N-Queens problem can be further studied in [22].

```
1.  % Program NQueens , with a record of achievement
2.
3.  % input : positive integer n ( the size of the board ).
4.
5.  % A square on the board is represented as a pair , column
6.  % number and row number , both from the set {1 ,.. , n }.
7.
8.  row (1.. n ).
9.  % achieved : row /1 = {1 ,... ,n }.
10.
11. col (1.. n ).
12. % achieved : col /1 = {1 ,... ,n }.
13.
14. n { queen (I ,J ) : col (I) , row (J) } n .
15. % achieved : Set queen /2 consists of n squares .
16.
17. : - queen (I ,J) , queen (I , JJ ) , J != JJ .
18. % achieved : Each column includes at most one square from queen /2.
19.
20. : - queen (I ,J) , queen (II , J) , I != II .
21. % achieved : Each row includes at most one square from queen /2.
22.
23. : - queen (I ,J) , queen (II , JJ ) , (I ,J )!=( II , JJ ) , |I - II |=| J -JJ |.
24. % achieved : Each diagonal includes at most one square from % queen /2.
```

Figure 2.13: Illustration of the Achievements methodology on the N-Queens problem.

2.3 Tools

In the realm of Answer Set Programming (ASP), various tools cater to different user needs and preferences. ASP systems, such as clingo and DLV2, provide a declarative problem-solving approach, while text editors play a crucial role in facilitating the creation and editing of ASP programs. This section explores ASP systems and text editors, briefly explaining their capabilities, limitations, and the considerations that guide users in choosing the most suitable option.

2.3.1 Systems

Over the years, many Answer Set Programming systems have been created. These systems have varied syntax or optimization differences, but they all share ASP's declarative expressiveness, and have progressively grown to became more efficient and intuitive.

As of today, two state of the art systems are clingo [12] and DLV2 [3]. In fact, as mentioned briefly on 2.1.1 the syntax we have been using, and will keep using throughout this document is not the classical formulas of ASP, but in fact clingo syntax, given that it is the technology we explore in this thesis.

In general, both these systems are user-friendly ASP systems, as their syntax is intuitive, they have fast runtimes, and documentation available online, in [12] for clingo, and in [3] for DLV, if further study of these systems is desired.

2.3.2 Text Editors

As briefly discussed in 1.2, effective text editors can greatly facilitate the creation and editing of ASP programs. In this section, we will examine the features and capabilities of some of the most popular and widely used ASP text editors, as well as any limitations they may have.

2.3.2.1 Online Editors

Online code editors are getting increasingly popular, since the lack of necessity for any specialized software or installations is welcomed by users. ASP is no exception, as online code editors for this approach have been created.

One of the main advantages of online code editors are their cross-platform compatibility, as users can access these editors from any device with an internet connection. Another benefit is their accessibility, as many of these tools are open-source, and developers from the community can improve on their functionalities.

Despite these benefits, there are also some limitations. Online tools may lack a manager of files and versions, and the performance and speed of these tools can vary depending on the user's internet connection and device.

One example of an ASP online editor is the Potassco [25] run website [24] of its system, clingo. This website is a great example of the pros and cons of online tools described above, as it can be accessed from any device with access to the web, with its syntax highlighting, though it is not open-source and can't be improved by the community. Despite this, it has no version control, since it is not possible to save progress directly in the website and if the page is refreshed the progress is lost. Additionally to store encodings it would have to be copied into a file on the user's machine. It also has performance issues, since the answer sets are computed on a server, it takes longer to run, and the results are only shown after the whole sets are calculated. This can create serious performance problems, particularly when opting to run with all answer sets, given the potential for network failures, slow connectivity issues or server overload due to a large number of users.

Another option for an online editor is the tool LoIDE [14], created by the University of Calabria. This editor can also be accessed in any web device, has syntax highlighting and additionally permits the usage of various files at the same time, and has an incorporated feature to save this files locally, directly. Moroever, this project is open source, supports both DLV and clingo systems, and records the project for the next session. In addition, this tool is also open-source permitting developers to improve upon this technology. However, it also has similar issues to the example seen above, as to store files it would have to be locally. To solve the performance problem, this tool restricts the amount of answer sets printed in the console.

2.3.2.2 Integrated Development Environments in ASP

Installing software systems locally has been a common practice for many years. This method provides the advantage of having direct control over the system and its configurations, as well as easy access to the data and applications stored on it. This option is highly efficient in running the code, depending on the machine or local network that it is installed into, but it is very likely to be faster then online

applications. It also provides a greater control as the full functionality of the system are available to the user. However, this option is the least user-friendly, because it also requires a better understanding of the systems functionalities.

However, locally installing systems also comes with some limitations and drawbacks, such as the requirement of technical knowledge and skills to maintain and troubleshoot the system. To tackle this issue, Integrated Development Environments (IDE) were created.

IDEs are software applications that provide tools for software development. They typically consist of a code editor, debugger, and other tools. Over the years, IDEs have become increasingly popular among software developers, as they provide an efficient and convenient way to create, test, and debug code. In general, IDEs have a very user-friendly interface, since their main focus is to improve the programmers experience and to make coding as intuitive as possible. It is also common for IDEs to have auto completion of code and automatic error prompts. They usually have good debugging tools for users to quickly understand and identify bugs. Despite that, since IDEs have many features, depending on the machine they are being run on, it can be difficult to run the application smoothly.

In ASP, most systems have available releases to install directly on the machine. For example, clingo's and DLV's releases can be found in [26] and in [28], respectively. There are also some plugins for existing IDEs. For example, Visual Studio Code [30] has two extensions, one for clingo [11] and one for DLV [7]. Both these extensions are good examples for the advantages for editing with IDEs, as managing files, the code highlighting and intuitive interface are a great environment to create ASP programs. Unfortunately, these extensions do not provide features for correcting automatically syntax problems in the code, nor debugging tools to better analyse the encodings.

2.3.2.3 Comparison

Each of these editing tools have distinct capabilities and limitations, prompting the question: which one is the most suitable? It depends on the user, and on the circumstances in which he or she wants to create code. Online editors can be better for laypersons who are only starting to learn programming, IDEs might better for ASP beginners that have a background of computer science, while ASP installations are better for experts who fully understand all capabilities of this technology. The choice must also be weighted based on which platform the user might want to program, since online editors can be run on any device with internet connection, while IDEs must be installed, and can require a powerful machine. ASP systems installations do not need such power, but not all systems can be run in all operating systems easily (simulated environments can always be used but those only create more accessibility and efficiency problems). Choosing one option must take into consideration all this factors, and must be done according to the context of the discussion at hand.

2.4 Discussion

We have now studied many approaches to help ASP encodings. Easy Answer Set Programming is a good methodology to help structure code, and commenting can help users to better understand them, and many text editors have been developed over the years. However, these three methodologies are given in a separate form, and a technology uniting all of them does not yet exist. Such technology,

targeting ASP newcomers, would be a great help in building the community of ASP developers by guiding their first steps utilizing this methods. To accomplish this we needed to explore these possibilities and come to a choice as to what would be the most effective approach.

In the initial evaluation, the prospect of directly employing locally installed ASP systems emerged as a less viable option. This was primarily due to its dependence on extensive terminal usage, a potentially daunting task for those unfamiliar with programming or in the early stages of learning ASP. Additionaly, the necessity of an additional text editor for coding, coupled with the lack of integrated console functionality, solidified that this option lacked intuitive appeal.

Subsequently, we were presented with a pivotal decision between online editors and integrated development environments (IDEs), as both options offer disctinct advantages and drawbacks. Online editors, while capable of running on any device without necessitating installation, present a challenge in efficiently computing all answer sets in a program due to their reliance on an internet connection. In contrast, IDEs require installation and may not be universally supported across all devices and operating systems. Faced with this dual scenario, we explored potential online editors and IDEs, subjecting each to a thorough evaluation.

LoIDE 2.3.2.1 stands out among online code editors and is recognized as a promising web-based option. This editor encapsulates the advantages inherent in online platforms and notable incorporates features typically associated with IDEs, as a comprehensive file management system, the ability to segregate answer set programs into distinct files, and support for running multiple ASP systems.

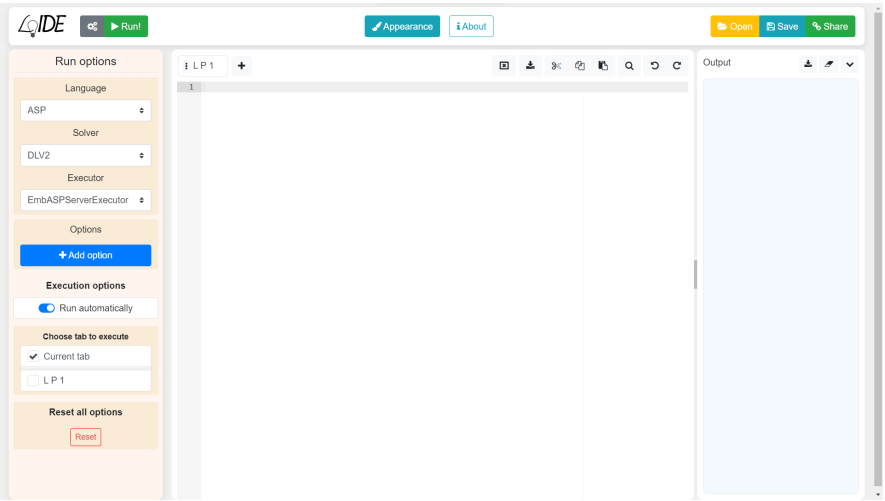


Figure 2.14: The opening page of the LoIDE [14] editor.

Despite these commendable attributes, LoIDE remains a tool in active development, introducing challenges due to its limited documentation and complexities in modifying project contents at scale without appropriate refactoring. Consequently, an alternative approach was considered in light of these considerations.

Our ultimate selection materialized in the development of a Visual Studio Code [30] extension. Firstly, Visual Studio code is widely accepted as one of the preeminent Integrated Development Environments, highly regarded by its flexibility, extensibility, and user-friendliness.

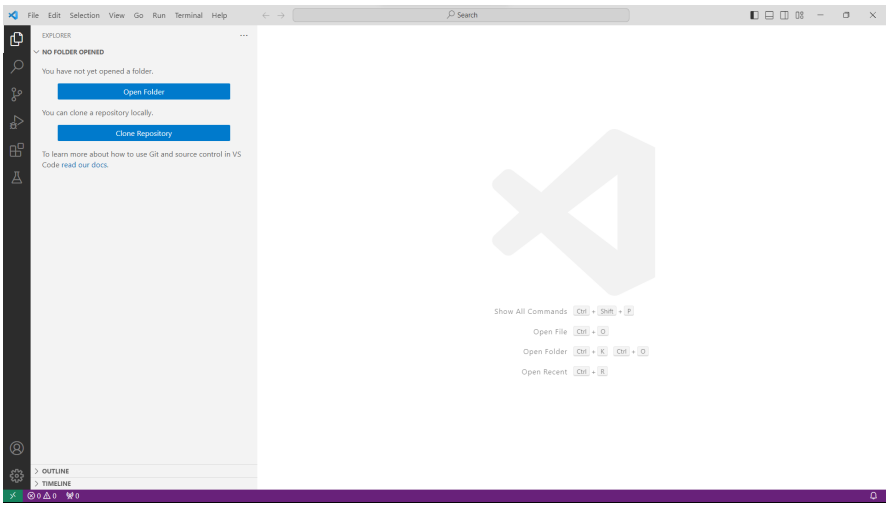


Figure 2.15: The opening page of the Visual Studio Code [30] editor.

Additionally, the presence of ASP extensions in the Visual Studio Code marketplace, particularly Answer Set Programming Language Support [11] and Answer Set Programming Syntax Highlight [2], served as a foundational reference point for our work.

Creating a Visual Studio Code extension also offered the advantage of building upon a clean slate while concurrently leveraging the functionalities of the aforementioned extensions. This avoided the need to modify their code, alleviating the limitations encountered in the LoIDE environment.

3

Proposed Solution

This chapter delves into the intricacies of our designed Visual Studio Code extension. The discussion commences with a comprehensive overview of the features detailed in Section 3, each explored in its dedicated subsection: Syntax Checking (3.1.1), Order Errors (3.1.2), Predicate Validation (3.1.3), On-Hover Predicate Information (3.1.5), Missing Comment Warning (3.1.4), and Feature Disablement (3.1.6). These features collectively contribute to the overall functionality and user experience of our extension.

We finish this chapter in Section 3.2, where we engage in a reflective discussion on the choices made during the design process, and reflect on some considerations that will shape the implementation of the extension.

3.1 Application Design

In 2.4, we introduce the development of a Visual Studio Code [30] extension, that depends on two already existing extensions, namely Answer Set Programming Language Support [11] and Asp Syntax Highlight [2].

The primary goal of developing this extension was to implement the Easy Answer Set Programming methodology, and emphasizing the importance of thorough commenting in the creation of answer set programs, specifically designed for individuals new to ASP.

To achieve this goal, we designed a range of features that analyze the supplied code, providing diverse error messages and code underlines. From this point forward, we will collectively refer to these errors, highlighted with red underlines and accompanied by on-hover messages, as error messages. Leveraging Visual Studio Code's support for such features, which aligned seamlessly with our project goals, each of these functionalities was tailored to address specific objectives integral to our project.

As highlighted in 2.1.4, embracing best practices in ASP encoding includes structuring the program into separate files, one where the solution to the problem is stored, and all different instances of said problem are present in the other files. Our designed features are crafted offering users a straightforward way of achieving this program separation.

With this foundation in place, we will provide a detailed introduction to each individual feature of our proposed work.

3.1.1 Syntax Checking

The first feature designed for our system is a syntax checker. The fundamental concept behind this functionality is to enable the system to identify invalid rules and notify the user of such occurrences, via error message. By doing so, this feature serves as a valuable aid for users in detecting errors within their code, elevating the overall program quality and mitigating mistakes. In Figure 3.1 we can see an example of an error message generated by an invalid rule.

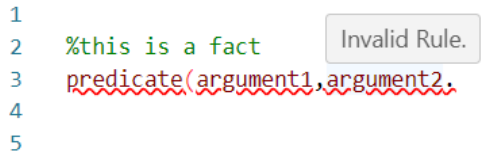


Figure 3.1: Example of a syntax error detected by the EZASP Visual Studio Code extension.

It is important to note that since this feature is not the primary focus of this dissertation, we do not provide an exhaustive explanation of all potential syntax errors. However, given that identifying invalid rules is essential for the features outlined later in this document, we have included this functionality because it aligns with our objectives.

It is also important to highlight that the Easy ASP methodology, as outlined in 2.2.1.3, does not accept cardinality constraints. However we find that the exclusion of these rules in the methodology is too restrictive, and as such, we have allowed for cardinality constraints in our application, deviating slightly from the specified methodology.

3.1.2 Order Errors

In Easy Answer Set Programming [10], as studied in Section 2.2.1, Fandinno et. all defend that the imposition of order in Answer Set Programs can be a great help in aiding ASP newcomers in their encodings.

Following this methodology, we designed a feature where we enforce the order of rules. Here, we detect if the code is in the following order:

- Constant Declarations;
- Facts;
- Choices;
- Definitions;
- Constraints;
- Optimization Statements;
- Show Statements.

If a rule is out of order, the program shows an error message in this specific rule, explaining where it should be located. We can see in Figure 3.2 an example of an error message where it points that the highlighted fact must be before the choice rule.

It is worth noting that while definitions are one of the rule types present in the Easy ASP methodology, we recognized that it may not be essential for all programs, as there are programs that might not need to define an extra predicate. As a result, we chose not to enforce its inclusion, and not show an error message if a program does not have definition rules.

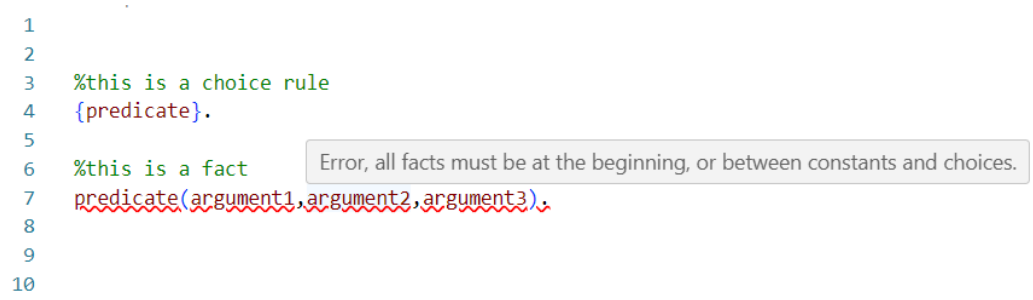


Figure 3.2: Example of an order error detected by the EZASP Visual Studio Code extension.

It is important to note that, contrary to the methodology's conventions, constant declarations, optimization statements, and show statements are not mentioned in the Easy ASP methodology. Nevertheless, we understood that they are important to Answer Set Programs, and therefore we extended this feature, in the order that we understood was most comprehensible for ASP newcomers. We also designed the extension so that it will only show error messages if constant declarations, optimization statements and show statements are in the wrong order, and if they do not appear, the extension will not show any errors. With this, the only rules that are mandatory are facts, choices, and constraints.

3.1.3 Predicate Validation

As also mentioned in Section 2.2.1, another intricacy of Easy Answer Set Programming is that rules must also be in an order so that every rule utilizes only predicates defined in the preceding rules. To accomplish this, we implemented an error message that detects if a predicate has not been defined, and if so it notifies the user.

In contrast to Order Errors feature discussed in Section 3.1.2, this specific feature does not only show if rules are out of order, but also detects if the predicate has not been defined at all, which will greatly help users detecting potential issues within their code. An example can be seen, in Figure 3.3, where the definition is using a predicate with two arguments, which has not been previously defined.

We also recognize the significance of maintaining continuity across files for this feature. Consequently, in the scenario where a program spans multiple files and a predicate has been defined in a preceding file but not in the current one, we ensure that the error message will not be triggered, by considering the defined predicates in all intended files.

Another noteworthy aspect to mention is our belief in the benefit of incorporating support for the Clingraph [15] predicates, as this offers a straightforward method for visualizing graphs in Clingo. These graphs are defined through fixed predicates derived from a predetermined list of facts, and

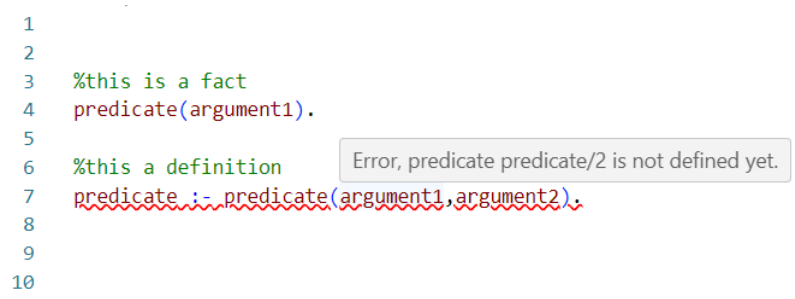


Figure 3.3: Example of an predicate validation error detected by the EZASP Visual Studio Code extension.

consequently, the program should not display an error message, given that these predicates are consistently present in the defined list. Because of this the program should not show an error if the reserved words are mentioned, more specifically the predicates node/1, edge/1, attr/4.

3.1.4 Missing Comment Warnings

As detailed previously in Section 3.1.5, proper documentation is essential where the predicates are defined, and in light of this we opted to introduce an additional of visual notification, ensuring users are aware of when a comment is absent.

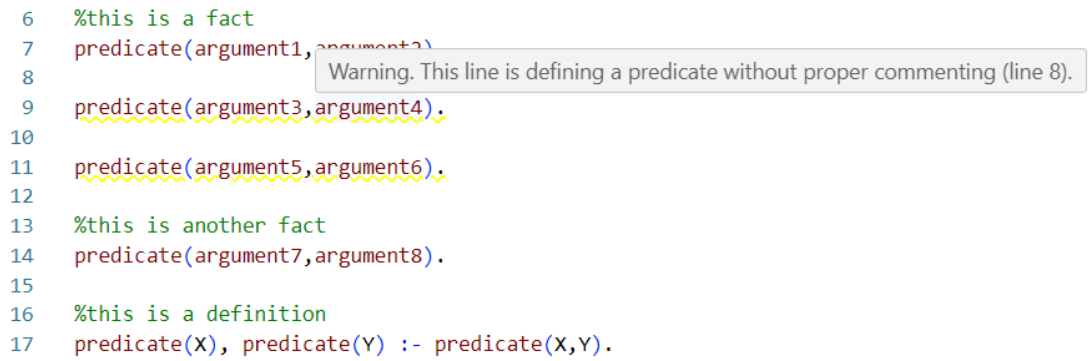


Figure 3.4: Example of a warning message from the EZASP Visual Studio Code extension, in contrast to Figure 3.5.

After careful deliberation, we settled on displaying a warning message, similar to the error messages distinguished by a yellow underline. This message will explain that the predicate has no comment, and points to the line where it should be present.

3.1.5 On-Hover Predicate Information

This feature is designed as follows: if a rule includes a comment, the tool captures and associates it with the respective predicate. Consequently, whenever a user hovers their mouse over a predicate in the program, a popup displays not only the associated comment but also the number of the line of code where the predicate is defined. An illustrative example can be seen in Figure 3.5.

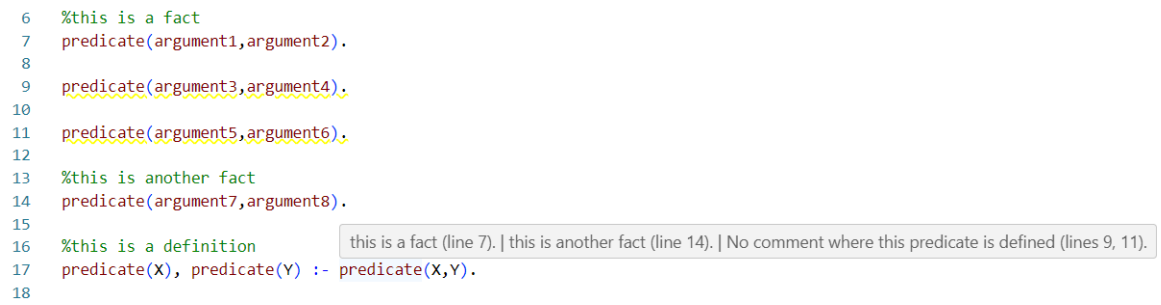


Figure 3.5: Example of an on hover message from the EZASP Visual Studio Code extension.

This feature goes hand in hand with one of the major objectives of this dissertation, which is the promotion of proper commenting of Answer Set Programs, and also serves good quality of life improvement for the users, increasing the efficiency in which they can understand their programs.

However, for this feature to achieve its full functionality, it is mandatory that the users create comments, and consequently, in the predicates that are not defined without a comment we incorporate a specific general message, explaining the case to the user.

Similarly to the Predicate Validation feature, we acknowledge the importance of extending the scope of the On-Hover Predicate Information across multiple files. Consequently, if a predicate is defined in a different file, it should still be accessible in the current file, and in scenarios where a program spans multiple files, the popup message must include the name of the file to provide comprehensive context, as the line number alone is no longer be sufficient.

It is also important to highlight a contrast from this feature to Lifschitz's methodology: our approach suggests placing comments immediately before the corresponding rule. Unlike the Achievements methodology, where comments primarily serve to document what has already been accomplished and should remain unchanged, our commenting method is designed to help users understand each line of code. We believe that placing comments immediately before the associated rule offers more benefits in terms of clarity and comprehension.

3.1.6 Feature Disablement

An in-depth evaluation of our design will be conducted in the following sections of this document. However, one potential challenge we acknowledge upfront is the possibility of information overload on the screen when dealing with larger-scale programs.

To tackle this issue, we incorporated a feature that enables users to promptly selectively disable each of the aforementioned features. This provides users with increased flexibility to customize the tool according to their specific needs and preferences, mitigating the potential for information overload.

3.2 Discussion

Having comprehensively outlined the various features and design considerations in our tool, we now engage into a discussion of the proposed solution.

We believe that this solution can be of great help for newcomers to ASP, as while we do not delve into many other intricacies, we are great help for many of the basis of what is Answer Set Programming, while still giving them the freedom to explore and code as they so desire.

Firstly, the ASP Language Support [11] extension works with the file extensions ".lp", ".ilp", ".cl", ".clp", ".Clp", ".iClp", ".blp" and ".iblp". As our extension is depending on the ASP Language Support extension, we cannot alter these extensions. However, we recognize that having numerous extensions, all serving the same purpose, might be unnecessary and potentially confusing for newcomers. Initially considering using only ".lp" (an acronym for "logic program"), we later decided to maintain compatibility with all extensions but recommend newcomers to stick to ".lp" for simplicity.

Another consideration we made was that, upon examining the Achievements methodology example in 2.2.2, we recognize the significance of commenting in structuring and understanding Answer Set Programs. While this methodology offers advantages, we deemed it overly extreme to force comments for every line of code, and decided on showing a warning to each line without a comment instead.

Lastly, there are a few considerations that must be taken into account in the implementation of this program, namely that it can be easy to visually fill the screen with information that can limit the space of the screen, or can be visually distracting on the user. We have already tackled this issue by adding Feature Disablement, it remains an important aspect to consider as we proceed to the implementation of each feature.

4

Implementation

In this chapter we explain the implementation of the proposed work detailed in the previous chapter 3, and provide information on the details of the extension. We start by analyzing how we structured our code in Section 4.1, and the work conducted before the specific implementation of each feature. Then, in Section 4.2 we delve into the specifications of each planned feature and how we implemented them.

4.1 Structure

Our extension, named EZASP, is accessible through references [16] and [17]. This extension was built using Visual Studio Code API JavaScript library, a purpose-built toolkit designed explicitly for the creation of Visual Studio Code extensions. The choice of this library facilitated seamless integration with the VS Code environment, providing us with a robust foundation for extending the functionality of the code editor.

As detailed in Section 3.1, our work builds upon two existing VS Code extensions: Answer Set Programming Language Support [11] and Asp Syntax Highlight [2]. To streamline the user experience, our extension automatically installs these dependencies when being installed in the VS Code environment.

We have also tested and validated the extension on Windows, Linux, and macOS environments, enhancing the accessibility and usability of our tool, accommodating users with different operating system preferences.

We organized the basis of our code into two distinct specifications: parsing and error message calculation, where this error calculation was done hand in hand with the features described in Chapter 3. The parsing phase is responsible for analyzing the input ASP (Answer Set Programming) code, with three key functions, each serving a unique purpose: Text Formatting, Rule Determination and Predicate Extraction.

4.1.1 Text Formatting

The first challenge we had to face, was the formatting intricacies that ASP have, regarding its line breaks. In ASP, as briefly mentioned in 2.1.1, the end of each rule is represented by a period. However, this means that a rule can be broken into different lines, and to the solver, it will end up the same.

To tackle this issue, we formatted each program supplied to our system. In this process, we systematically stored every individual rule in an array, where each array position aligned with a specific rule, eliminating the spaces between rules, as it can be seen in Figure 4.1.

```
3   %facts
4   fact
5   (1;2;3).
6
7   %choice
8   {choice(X)}
9   :- fact(X).
```

(a) An Answer Set Program with line breaks

```
0: ''
1: '%facts'
2: 'fact(1;2;3).'
3: '%choice'
4: '{choice(X)}:- fact(X).'
```

(b) The rule array that the program generates

Figure 4.1: A schematization of the text formatting that the EZASP extension does to every Answer Set Program.

Furthermore, to preserve the original locations of text requiring error messages, we introduced another array. For each rule in the formatted text array, this additional array stored four essential variables that encapsulate the information necessary for underlining specific lines of code using the Visual Studio Code API. We from now on will denominate these variables as the range of the rule, since they represent a range of characters in the text editor. These variables are:

- Starting line;
- Finishing line;
- Starting position within the line;
- Finishing position within the line.

An illustrative example can be seen in Figure 4.2, where first in 4.2(a) we see a program with an error in a line that has a line break, and then in 4.2(b) we see the corresponding array of the variables of each line.

By adopting this strategy, we ensured implementation of the features outlined in Chapter 3, even in scenarios where programs contained line breaks.

4.1.2 Rule Determination

To implement our envisioned features, we were faced with another challenge: determining the type of each rule in the program.

To address this, we devised a function so that given a rule, it identifies its type. If the rule is not in alignment with supported types, the function determines it as an invalid rule.

```
1
2   %facts
3   fact
4   (1;2;3).
5
6   %choice
7       {choice(X)}
8   :- fact(X).
9
```

(a) A syntax error on an program with line breaks

```
0: {lineStart: 1, lineEnd: 1, indexStart: 0, indexEnd: 6}
1: {lineStart: 2, lineEnd: 3, indexStart: 0, indexEnd: 9}
2: {lineStart: 5, lineEnd: 5, indexStart: 0, indexEnd: 7}
3: {lineStart: 6, lineEnd: 7, indexStart: 4, indexEnd: 11}
```

(b) The line information array

Figure 4.2: A schematization of the information storage of the lines that the EZASP extension does to every Answer Set Program.

This determination is done by checking each supported type, being comment, fact, choice, definition, constraint, optimization statement, show statement, constant and empty. Let us now explain the verification process [1]:

- Initially, we remove any empty spaces from the rule and check if the rule is an empty string. If it is, we return the "empty" rule type.
- If a rule starts with the character "%", we return the "comment" rule type. If the rule contains a "%" symbol in the middle, we shorten the rule at that point, removing everything after the "%" symbol, and continue the examination.
- We then verify if the rule contains only certain valid symbols, such as lowercase and uppercase letters, numbers, '_', '#', '+', '-', '*', '/', '\', '"', ',', ':', '{', '}', '(', ')', '.', '|', '<', '>', '=', '!', ';', '.', '@' and ' '. If any other character is present, we return the "invalid rule" rule type.
- Next, we check whether each opening parenthesis or bracket has a corresponding closing parenthesis or bracket. If not, we return the "invalid rule" rule type.
- If a rule starts with the string "#show" and is followed by a fact or contains a "/" symbol followed by a number, we return the "show statement" rule type.

[1] We could have added a specific error message to each possible syntax error in each rule, but we decided against it given that syntax validation is not the main goal of this dissertation.

- If a rule starts with the string "#const" and contains the "=" symbol, we return the "constant declaration" rule type.
- If a rule starts with the strings "#maximize" or "#minimize," followed by the "{" symbol and ending with the "}" symbol, we return the "optimization statement" rule type.
- If a rule does not include the symbols ":-", any brackets, the first character is a lowercase letter, has an equal number of open and closed parentheses, and the last character is an uppercase or lowercase letter or number, we return the "fact" rule type.
- If a rule contains brackets before the ":-" symbols in the head, and the characters before the "{" and after the "}" are numbers, or if there are no characters between the brackets, we return the "choice rule" rule type.
- If a rule contains the symbols ":-", has no head, and the body includes uppercase and lowercase letters, we return the "definition" rule type.
- If a rule contains the symbols ":-", lacks a head, and the body includes uppercase and lowercase letters, we return the "constraint" rule type.
- If the rule does not satisfy any of the previous logical checks, it returns the "invalid rule" rule type.

4.1.3 Predicate Extraction

Before delving into the specifics of each feature, we need to do a last calculation, where we extract and convert every predicate in the program into a readily accessible representation.

To achieve this, we opted for a JavaScript object to represent each predicate, where we stored the predicate's name and the number of arguments it possessed. Additionally, we made a deliberate distinction between predicates used in the head of each rule and those in the body. As explained later in this chapter, this separation proved beneficial for certain features we implemented, notably Predicate Validation and On-Hover Predicate Information.

The predicate extraction was implemented using regex patterns, specifically one generated with the assistance of prompts to ChatGPT [23]. This regex identifies patterns resembling functions, being a word followed by parenthesis, capturing the predicate name (starting with a lowercase letter) and, if present, the content within parenthesis (arguments).

In this process, we distinguish the head from the body of each rule, apply the regex parsing, and store the matches of predicates in the array corresponding to that rule's head or body.

4.2 Features

Our extension works in a two-step process. Initially, it reads and formats the files, extracts the predicates and determines the type of each rule. Then it utilizes this parsed information to compute the features delineated in Chapter 3.

With this in mind, we can understand the intricacies of each individual feature, which is what we explore in this section.

4.2.1 Syntax Checking

The Syntax Checking feature involves examining the type of each rule in the program. If any rule is identified as invalid, its corresponding range is added to an array designated for error messages, tagged with the message "Invalid Rule.".

The importance of this feature lies in the Rule Determination function, detailed in Section 4.1.2. While this feature does not support every intricacy of Answer Set Programming (ASP), like for example the Python script writing capabilities of clingo [12], it does provide robust coverage. Despite not addressing every nuance, we ensured the Rule Determination function's reliability by subjecting it to unit tests that cover all the intricacies of the rule determination function.

4.2.2 Order Errors

As mentioned in Section 3.1.4, programs must be organized in the following manner: constants, facts, choices, definitions, constraints, optimization statements, and show statements.

Depending on the rule out of order, incorporated these possible error messages:

- "Error, all constants must be at the beginning."
- "Error, all facts must be at the beginning or between constants and choices."
- "Error, all choices must be at the beginning or between facts and definitions."
- "Error, all definitions must be between choices and constraints."
- "Error, all constraints must be between definitions and either optimization or show statements."
- "Error, all optimization statements must be between definitions and show statements."
- "Error, all show statements must be after constraints or optimization statements."

Regarding order, we made adjustments for each error message to correspond with the type of rule. For example, if a block of constants is not in the beginning of the program, the error message is readily shown, as we can see in Figure 4.3. However, for a fact, it can be also at the beginning, but if a constant exists, which are not obligatory, it must necessarily be after this constant rule, as shown in Figure 4.4. In contrast, for choices, their respective error message shows that they can be in the beginning as well, due to the fact that there is a possibility for all the facts to be concealed in another file, as we will explain further down in this document.

Then, other adjustments can be seen due to the fact that neither optimization statements or show statements are obligatory, and in spite of this, the error messages show each possibility of rule that can be its neighbour.

It is also important to understand that the code iterates the rules from first to last, and therefore will always prioritize the first rules. For example, if we look closely at Figure 4.4, either the choice in line 7 or the fact in line 10 can be considered wrong, but since the choice is before the fact, we consider the fact in line 10 to contain the error. Despite the fact that we could have done this differently, we believe that this method is the most intuitive to users, as order is the main concept we are imposing.

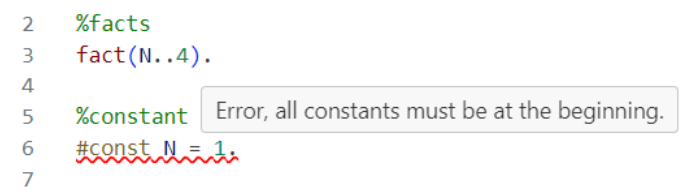


Figure 4.3: Example of an error message on an out of order constant.

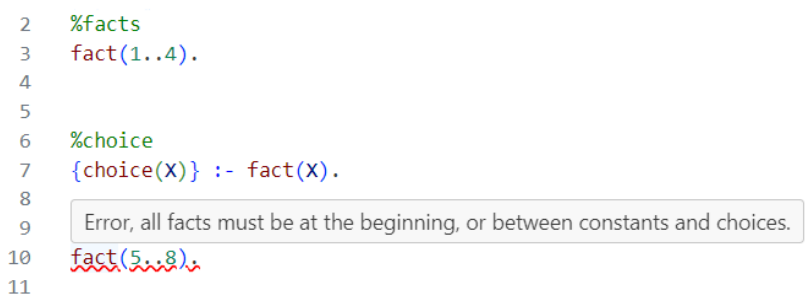


Figure 4.4: Example of an error message on an out of order fact.

Additionally, as mentioned shortly in 3.1.2 we deemed that definition error messages are not necessary in all programs, and therefore we did not add an error message covering it, as it is possible to see in Figure 4.5

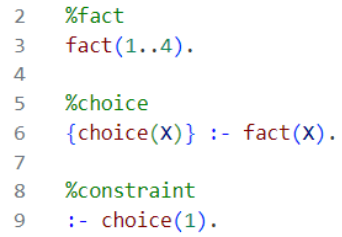


Figure 4.5: Example of a program without a definition and no error message.

However, we did add two more error messages, to cover two specific cases:

- "Error, this block of *rule type* is in between a block of other rules."
- "Error, constraints must be preceded by choice rules."

Firstly, the error message "Error, this block of *rule type* is in between a block of other rules.", arises when a rule is wrongly positioned, and the rule immediately after it is also out of order. But, specifically, the first rule out of order is the one before the second rule in the stratification. This would mean that stating that the second rule to need to be before the first one would be confusing, and therefore we changed this specific case to this message.

Let us consider the example in Figure 4.6. Both the facts in line 8 and the choice in line 11 are out of order. Because of this, we send two error messages, where for the fact we say that it must be in the beginning. However, for the choice rule, it would be confusing to say that it should be between a

fact and a definition, because it already is in between these two rules, and therefore we said that it is between a block of other rules, which it is in between a block of definitions.

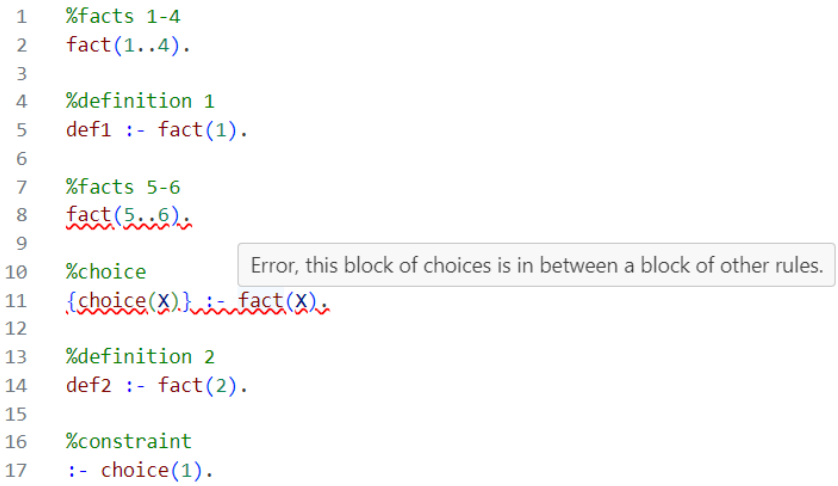


Figure 4.6: Example of a choice in between a block of definitions.

We also established a crucial condition for an answer set program's validity: choice rules must precede constraints. Consequently, a dedicated error message notifies of any constraint not preceded by a choice rule, stating, "Error, constraints must be preceded by choice rules.", as illustrated by Figure 4.7.

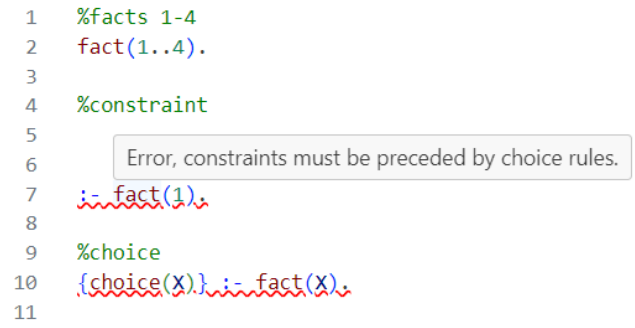


Figure 4.7: Example of an error message where a constraint is preceding a choice rule.

Lastly, we now explain the process of computing the specific error messages, and for this we can look at the pseudo code in Figure 4.8.

In this process, we systematically iterate through each rule type, following the defined rule order. For each rule type, we will also iterate the formatted text, and ignore any rules positioned before the current rule type in the rule's order. If a rule is detected after the expected order for the current rule type, a boolean variable, *ruleAfterRuleTypeFound*, is set to true. Subsequently, if a rule with the same type as the current rule type is encountered while *ruleAfterRuleTypeFound* is true, it signals the detection of an error.

```
FOR ruleType IN types {

    ruleAfterRuleTypeFound = FALSE

    FOR rule IN formatText(text) {

        IF getRuleType(rule) < ruleType {}

        ELSE IF getRuleType(rule) > ruleType {
            ruleAfterRuleTypeFound = TRUE
        }

        ELSE IF getRuleType(rule) == ruleType && !ruleAfterRuleTypeFound {
            push to errorMessages getMessage(ruleType)
        }
    }
}
```

Figure 4.8: Pseudo-code for Calculating Order Errors.

4.2.3 Predicate Validation

In this feature, we show an error message if a predicate is being used in a rule without it having been defined previously, even if it is defined in a subsequent rule, either by a fact, a choice or a definition rule.

To achieve this, we read each rule sequentially, from first to last, and for each predicate in the head of the rule, we add it to an array of defined predicates. In contrast, we check for each predicate in the body if they are in the array of defined predicates. If not, then the error is detected. As mentioned in Section 3.1.3, we add the exception for the predicate "attr/4", for the sake of the usage of this permanent predicate regarding the Clingraph [15] visualizer.

A detail that must be explained here is that, as mentioned in 2.1.4, adopting a best practice in ASP encoding involves organizing the program into distinct files. This approach ensures that a single file serves as a solution to various potential issues and different instances. The ASP Language Support extension for Visual Studio Code [11] already facilitates this by supporting the separation of programs.

To utilize this separation, the extension Answer Set Programming Language Support [30] has a command "> ASPLanguage: Initialize clingo config file in the current working directory.". This command generates a file named config.json in the same directory as the currently open file. Among other configurations, this file includes a parameter called "additionalFiles," which points to an array. Users can easily add files to this array by inserting the file names, and then run the program by executing the command: "> Compute Answer Sets (config.json).". In Figure 4.9 we can see an example of a config.json file generated by the Answer Set Programming Language Support extension.

Recognizing the importance of this feature, we have expanded the Predicate Validation feature, so that the extension now checks for the existence of a config.json file with additional files. If found, the extension will firstly read these extra files and their defined predicates, and store them making it so if these predicates are used in the current file the extension will not show an error message.

```
{
    "name": "Sample Config",
    "version": "0.1.0",
    "description": "Documentation: https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf",
    "author": "John Doe",
    "additionalFiles": [
        "filename.lp"
    ],
    "args": {
        "verboseMode": 1,
        "outputFormat": 0,
        "parallelMode": {
            "useParallelMode": false,
            "mode": "compete",
            "threads": 2
        },
        "timeLimit": 0,
        "solveLimit": {
            "conflicts": 0,
            "restarts": 0
        },
        "preProcessor": false,
        "models": 0,
        "customArgs": ""
    }
}
```

Figure 4.9: Example of a config.json file created from the ASP Language Support extension.

This is achieved by reading and validating predicates in each file independently. For each supplementary file specified in the config.json, the extension identifies and incorporates the defined predicates, integrating them into the system of the current file.

4.2.4 Missing Comment Warning

This feature shows a warning, similar to the error messages but with a yellow underline instead of red, to predicates that are being defined without a comment in the line immediately above. This feature complements the On-Hover Predicate Information, by providing visual help to the user, pointing to the predicates that will not have on-hover information.

The warning also points to the line in which the rule must have a comment, saying "Warning. This line is defining a predicate without proper commenting. (line 1)".

The implementation of this feature was made hand in hand with the On-Hover Predicate Information feature, described in 4.2.5. In the same context where we incorporated the message indicating the absence of a comment in the map, we also added its range to an array. This facilitated the application of a yellow underline, similar to how we handle other error messages.

4.2.5 On-Hover Predicate Information

Every time a predicate is defined, we store the comment, and subsequently, every time that predicate is used, if the user hovers the mouse on top of the predicate, the previously mentioned comment will appear as a pop-up message, also containing the line number of the rule where it was defined. Additionally, if a comment is not defined, for example, in line 1, the error message "No comment where this predicate is defined (line 1).".

If a predicate is defined more than once, then all the hover messages appear in order of definition, with the symbol "|" separating them. In the case of multiple definitions not being commented, the

lines will appear in the error message together, as seen in Figure 3.5.

In similarity to the Predicate Validation feature, we also added to On-Hover Predicate Information multiple file support, where if a config.json file exists, we include comments from the additional files as hover messages.

Additionally, to prevent confusion arising from multiple files, we have also incorporated information about the file of origin. This improvement addresses potential complexities introduced by the use of multiple files.

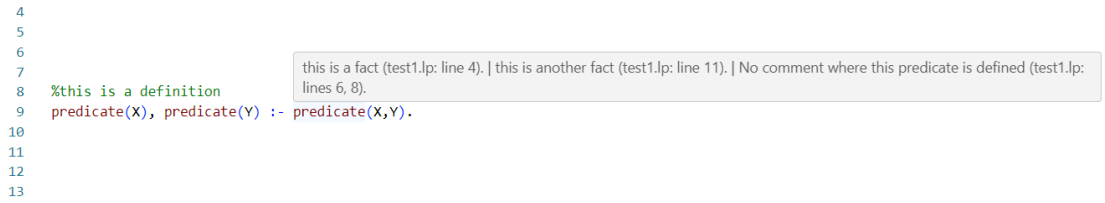


Figure 4.10: Example of On-Hover Predicate Information feature, showing predicates that are being defined in different a file.

To calculate these messages, we iterate through each predicate within the formatted text. We create a map where the predicates located in the head of each rule serve as keys, and the corresponding values are arrays containing the messages found in the comments associated with these predicate definitions. The last element of this array consistently holds a message indicating the presence of any defined predicates lacking comments. In cases where there are defined predicates without comments, the program will update this array position with information about the lines and filenames where these predicates are found.

4.2.6 Feature Disablement

Lastly, to enhance user customization, we introduced the option for users to selectively display features based on their preferences.

To facilitate this, a new command, "> EZASP - create config.json," has been implemented to generate the config.json file. Within this file, an additional parameter, disableFeatures, has been incorporated. This parameter encompasses five sub-parameters: syntaxChecking, orderErrors, predicateErrors, hoverPredicates, and commentWarnings, each corresponding to a boolean value (true or false). By default, these boolean values are initialized as false, indicating that the respective feature is enabled. When toggled to true, the specific feature is deactivated.

It is crucial to acknowledge that since the boolean values are represented as strings in JSON, and any non-"true" or non-"false" string may be entered, we have opted to maintain the feature as enabled if any non-standard word is provided.

It is important to note that this revised config.json file is compatible with all other features of the Answer Set Programming Language Support extension [11]. Consequently, we recommend utilizing the new command and config file for these configurations as well.

In this feature we designed separate arrays for each type of error message corresponding to each individual feature. If a rule type to disable a particular feature is set to true, we simply do not run the code that calculates and adds information to these arrays.

```
{
    "name": "Sample Config",
    "version": "0.1.0",
    "description": "Documentation: https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf",
    "author": "John Doe",
    "additionalFiles": ["test1.lp"],
    "disableFeatures":{
        "syntaxChecking":"false",
        "orderErrors":"false",
        "predicateErrors":"false",
        "hoverPredicates":"false",
        "commentWarnings":"false"
    },
    "args": {
        "verboseMode": 1,
        "outputFormat": 0,
        "parallelMode": {
            "useParallelMode": false,
            "mode": "compete",
            "threads": 2
        },
        "timeLimit": 0,
        "solveLimit": {
            "conflicts": 0,
            "restarts": 0
        },
        "preProcessor": false,
        "models": 0,
        "customArgs": ""
    }
}
```

Figure 4.11: Example of a config.json file created from the EZASP extension.

5

Evaluation

In this chapter we delineate the evaluation conducted to our work, and the subsequent results. In Section 5.1 we start by presenting the methodologies used. Then, in Section 5.2 we demonstrate the results obtained by our evaluation phase, and analyse and discuss the obtained results. Finally, in Section 5.3 we explore the conclusions drawn from this evaluation.

5.1 Methodology

5.1.1 Test Cases

Our study focused on two separate test cases within the NOVA School of Science and Technology's Bachelor and Masters in Computer Science and Engineering, where Answer Set Programming is studied at two different levels. One in an introductory context, the Bachelor's subject of Artificial Intelligence (AI), and one in a more advanced setting, the Master's subject of Knowledge Representation and Reasoning Systems (KRRS). Since we had these test cases, we opted in doing an evaluation based on a questionnaire to easily accommodate the feedback of the two student groups.

5.1.2 Evaluation Methodologies

In software development, many methodologies for evaluating applications have been created over the years. Holyer, in [18], enumerates some examples of classic methodologies to evaluate interfaces, including some via questionnaires, like 'GOMS' (Goals Operators Methods and Selection Rules), 'SUMI' (Software Usability Measurement Inventory), etc.

In GOMS each separated task, referred to as operators, is evaluated individually and then all components are evaluated together to understand if the main goal was achieved.

On the other hand, SUMI, provides an overall score, along five sub-scales:

- "Affect" (a measure of how much the users found working with the system to be "pleasurable");
- "Efficiency" (to what degree that considered their use of the system to be "productive");
- "Helpfulness" (how much help they perceived that the system gave them),
- "Control"(their subjective feeling of "being in control");

- "Learnability" (how easy they felt the system was to learn).

These methodologies were possibilities to explore. GOMS suited our work well, considering that our work is easily divided into different components, if we consider each feature to be a component and evaluate them individually. On the other hand, SUMI provides some sub-scales of what we determined that was of value to measure in our extension's objectives. Consequently, we decided on utilizing both these methodologies in part, by creating a questionnaire where we individually evaluate each of the features described in Chapter 3, based on the sub-scales given by SUMI. However, to reduce the response time of the questionnaire, which increases with each question, we reduced the sub-scales to the ones we considered to be in line with the main objectives of this dissertation, and focused on testing efficiency, helpfulness, and learnability. We omitted affect due to its subjective nature, which would return less reliable and consistent conclusions. Additionally, we excluded control since our extension, designed to enforce order that is not mandatory, could introduce an extra layer of complexity for the results making it also challenging to take clear conclusions.

5.1.3 Questionnaire Design

As previously mentioned in Section 5.1.2, our evaluation methodology involved administering a questionnaire to students enrolled in the Artificial Intelligence and Knowledge Representation and Reasoning Systems courses. This questionnaire aimed to measure how efficient, helpful, and learnable each feature is. To capture these aspects, the questionnaire included the following questions:

- "How easy is it to understand the *name* feature?";
- "How practical is it to use the *name* feature?";
- "How much did the *name* feature make it easier for you to understand your code and find mistakes?".

These questions were presented in both English and Portuguese to ensure accessibility for all respondents. This approach accommodated students who might not have a strong understanding of English, as well as Erasmus students who might not be proficient in Portuguese.

For each question, the students were asked to rate each question on a scale from 1 to 5, with 5 indicating the highest rating and 1 indicating the lowest. Additionally, an open-ended question was included for each feature, prompting the students to provide any additional feedback, report encountered issues, or suggest improvements: "With this feature, did you have any problems understanding it, find any bugs, or have any other type of feedback?".

With this approach we received qualitative and quantitative ratings, which prompted our evaluation process. In Figure 5.1 we can see an example of a quantitative question, and 5.2 we can see an example of a qualitative question.

5.2 Results and Analysis

During the two separate subjects, there are 245 registered in the IA subject, and 80 in the KRRS subject. Our extension was installed a total of 298 times, where the difference in number of students registered

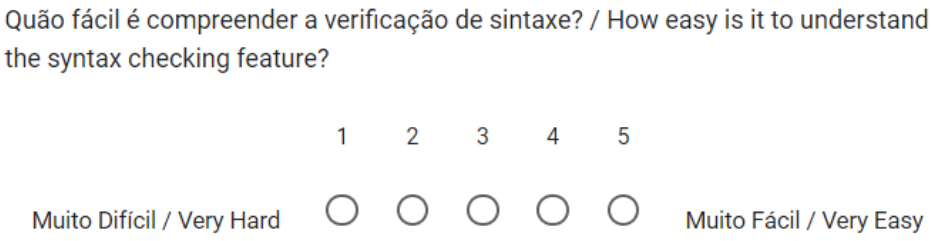


Figure 5.1: Example of a quantitative question.

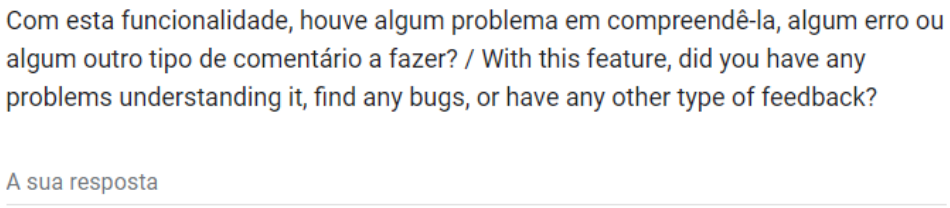


Figure 5.2: Example of a qualitative question.

in the classes compared to the number of installations consists on the students who were repeating the classes and did not need to redo the projects. Out of these students, 38 answered to the questionnaires, with 16 from the IA subject and 22 from the KRRS subject.

Despite the relatively low response rate, we provided our contact information to the professors of each subject for bug reporting purposes, and only one bug was reported out of the 298 installations, which was promptly addressed, suggesting that there are no major installation issues with the extension.

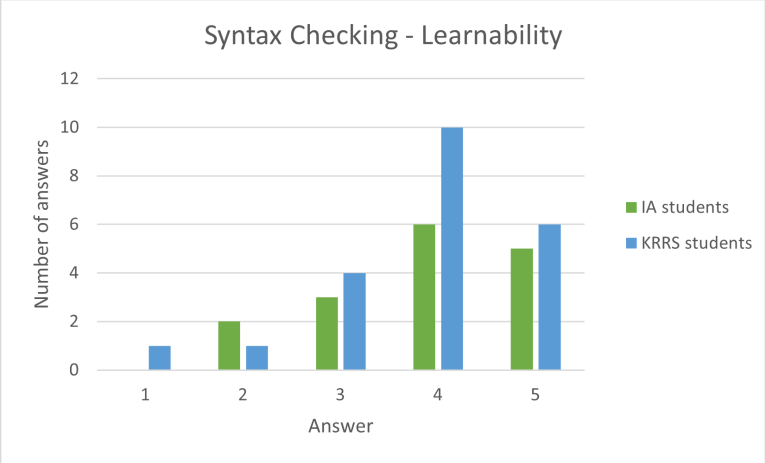
Now, we will delve into the results for each individual feature.

5.2.1 Syntax Checking

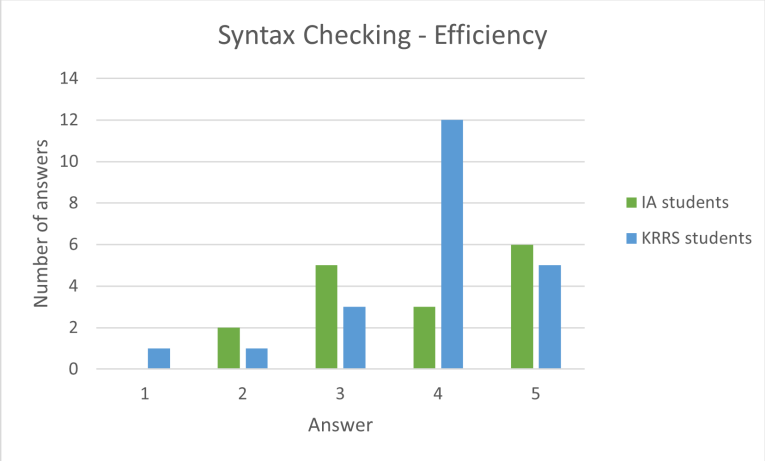
In Figure 5.3, we present the results of the questionnaires, measuring learnability (5.3(a)), efficiency (5.3(b)), and helpfulness (5.3(c)) for both classes of students.

The average scores for these metrics were approximately 3.9 for learnability, 3.8 for efficiency, and 3.4 for helpfulness. From these results, we can infer that error messages were well-received by the students, indicating ease of learning and practical use. However, further investigation is required to understand the relatively lower score for helpfulness.

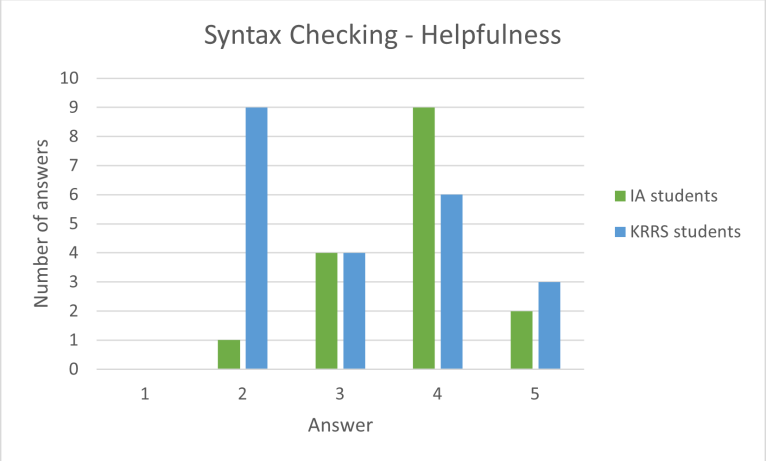
Firstly, we understood that the score could have been higher if we had a specific error message for each specific syntax error. However, upon segregating the data by class, we observed a notable difference in helpfulness. Students from the introductory subject rated it 3.8 on average, while those from the advanced subject rated it 3.4. This suggests that while syntax checking was beneficial for students learning basic ASP syntax, it was perceived as less helpful by those already familiar with the syntax. We can further see these results by the fact that the majority of the IA students gave the helpfulness score of 4, while the most common answer for KRRS students was 2.



(a) The results on how easy to understand the syntax checking feature is



(b) The results on how practical to use the syntax checking feature is



(c) The results on how helpful to use the syntax checking feature is

Figure 5.3: A visual representation of the results of the questionnaire for the syntax checking feature.

5.2.2 Warnings about Order of Rules Errors

Figure 5.4 illustrates the results of the questionnaires for the warnings about order of rule errors, measuring learnability (5.4(a)), efficiency (5.4(b)), and helpfulness (5.4(c)).

The average scores for these metrics were 3.6 for learnability, 3.5 for efficiency, and 3.3 for helpfulness. While generally well-received, students expressed confusion about the error messages showing on the IDE and not while running the programs. Because of that it makes sense that the scorers are relatively low. Additionally, in this feature all scores are higher in the students of the IA class than the ones from the KRRS class, which points to order errors being more appreciated by students who only started learning Answer Set Programming, and less by those who already have some understanding of it. This observation aligns with our expectations, as users with prior knowledge of ASP do not benefit as much from the enforcement of strict order.

5.2.3 Predicate Validation

Figure 5.5 depicts the results of the questionnaires for predicate validation, measuring learnability (5.5(a)), efficiency (5.5(b)), and helpfulness (5.5(c)).

The average scores for these metrics were 3.8 for learnability, 3.9 for efficiency, and 3.7 for helpfulness. This feature was well-received overall, with consistently high scores across both classes. However, KRRS students rated efficiency slightly lower, possibly due to their preference for the flexibility that the lack of order gives in predicate usage.

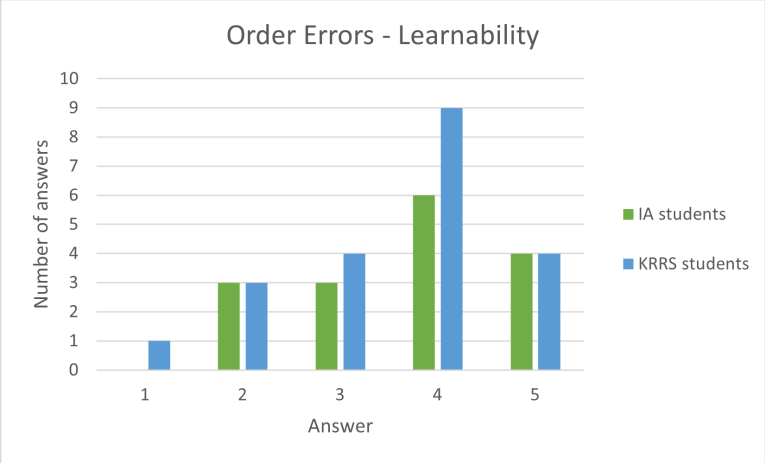
It is also worth noting that for IA students, the most frequent ratings in terms of learnability were 3 and 5, with fewer responses falling in the 4 range. This observation can be attributed to the extension generating the same error message when a predicate does not exist, resulting in an error during program execution. In contrast, if a predicate exists but is not defined before its usage, no errors occur during execution. This inconsistency could lead to confusion, explaining why students who understood this difference rated the feature higher, while those who encountered difficulties tended to give it a lower score.

5.2.4 Missing Comment Warning

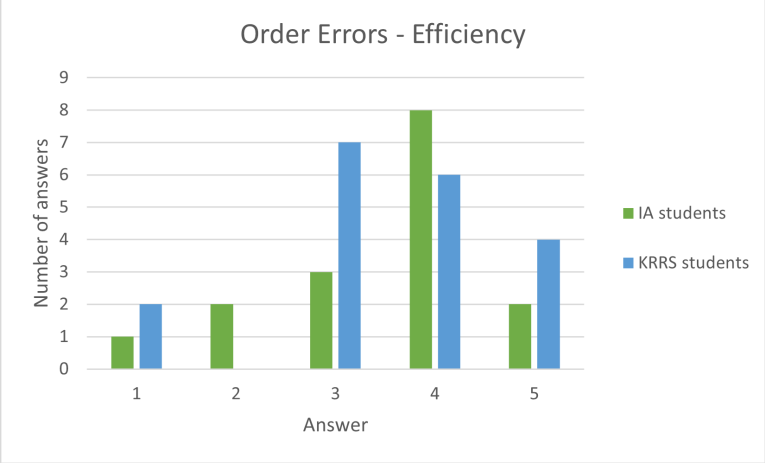
Figure 5.6 presents the results of the questionnaires for the missing comment warning feature, measuring learnability (5.6(a)), efficiency (5.6(b)), and helpfulness (5.6(c)).

The average scores for these metrics were 3.8 for learnability, 3.0 for efficiency, and 2.9 for helpfulness, with the most common answers for efficiency being 1 and 3, and for helpfulness 3 for the IA students, and a mix between 1 and 4 for the KRRS students. With this relatively low scores and the answers we had in the open question, we found that this feature was perceived as somewhat intrusive and less helpful by some students, with suggestions to disable it by default.

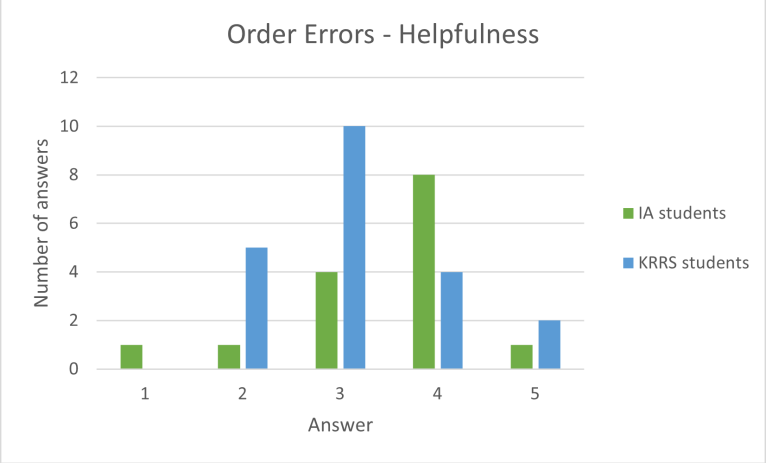
Initially, we recognized the potential intrusiveness of this feature. However, given the common absence of comments in many students' code, we deemed it necessary to alert users about missing comments. However, after receiving feedback we acknowledged the intrusiveness of the feature and made adjustments, since the On-Hover Predicate Information also incentivizes proper commenting, we chose to follow the suggestion of the students to disable the feature by default, and therefore updated



(a) The results on how easy to understand the order errors feature is

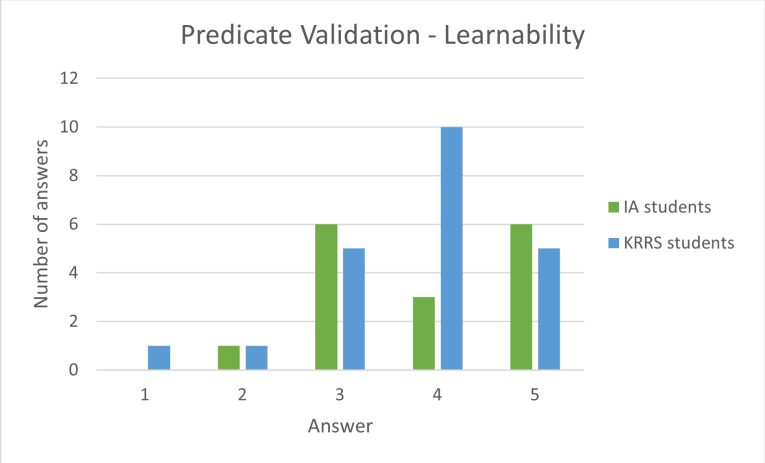


(b) The results on how practical to use the order errors feature is

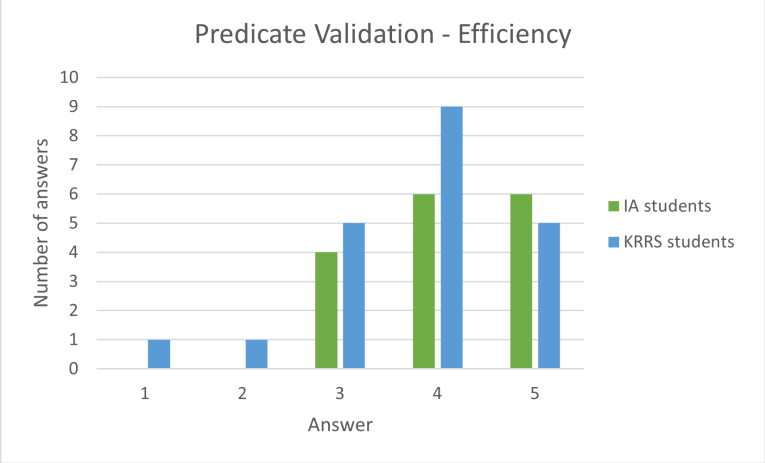


(c) The results on how helpful to use the order errors feature is

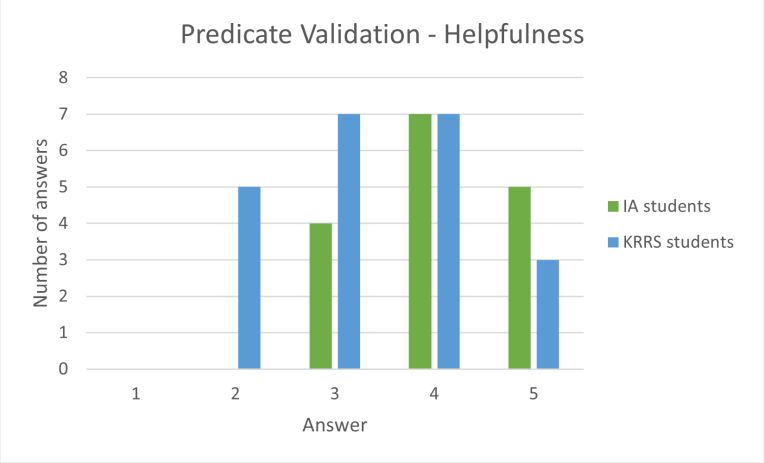
Figure 5.4: A visual representation of the results of the questionnaire for the warnings about order of rule errors feature.



(a) The results on how easy to understand the predicate validation feature is

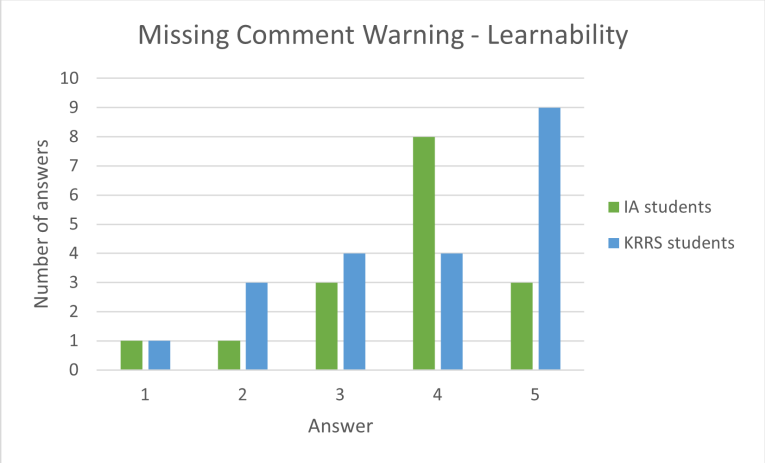


(b) The results on how practical to use the predicate validation feature is

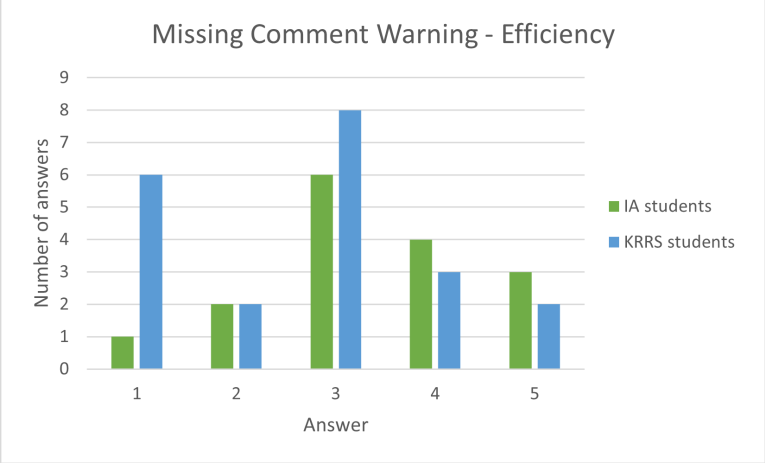


(c) The results on how helpful to use the predicate validation feature is

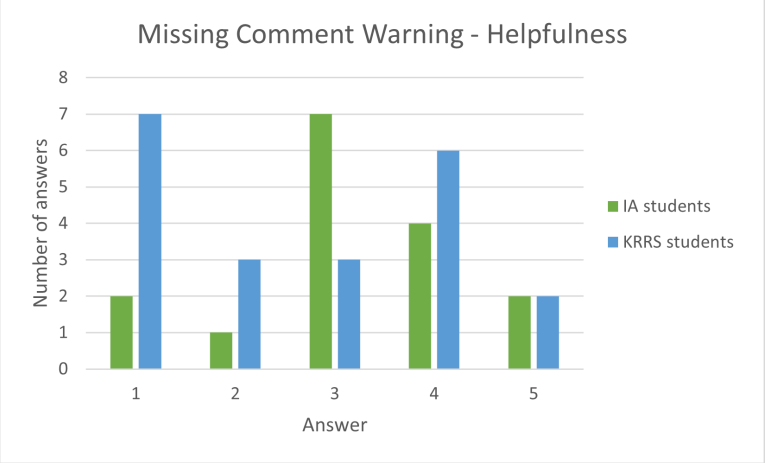
Figure 5.5: A visual representation of the results of the questionnaire for the predicate validation feature.



(a) The results on how easy to understand the missing comment warning feature is



(b) The results on how practical to use the missing comment warning feature is



(c) The results on how helpful to use the missing comment warning feature is

Figure 5.6: A visual representation of the results of the questionnaire for the missing comment warning feature.

the extension so that if there is no config.json file the feature is disabled, and when a new config.json file is created the boolean flag for the disablement of this feature is set to "true". Additionally we also and lowered the opacity of the yellow underline to reduce on the intrusiveness of the feature if the user chooses to enable it.

5.2.5 On-Hover Predicate Information

Figure 5.7 shows the results of the questionnaires for the on-hover predicate information feature, measuring learnability (5.7(a)), efficiency (5.7(b)), and helpfulness (5.7(c)).

The average scores for these metrics were 3.5 for learnability, 3.5 for efficiency, and 3.3 for helpfulness. Interestingly, the most common answer for helpfulness, in the IA student group is 4, when in the KRRS group is 3. Students who have more of an understanding of ASP are more likely to remember the meaning of their predicates, which explains this dispersion. Additionally, since this feature only incentivises commenting, and does not directly impact the reasoning of the programs, it makes sense that this feature has so many answers with a median score for helpfulness. With this in mind we understand that this feature was well received and can be considered effective.

5.2.6 Feature Disablement

Figure 5.8 illustrates the results of the questionnaires for feature disablement, measuring learnability (5.8(a)), efficiency (5.8(b)), and helpfulness (5.8(c)).

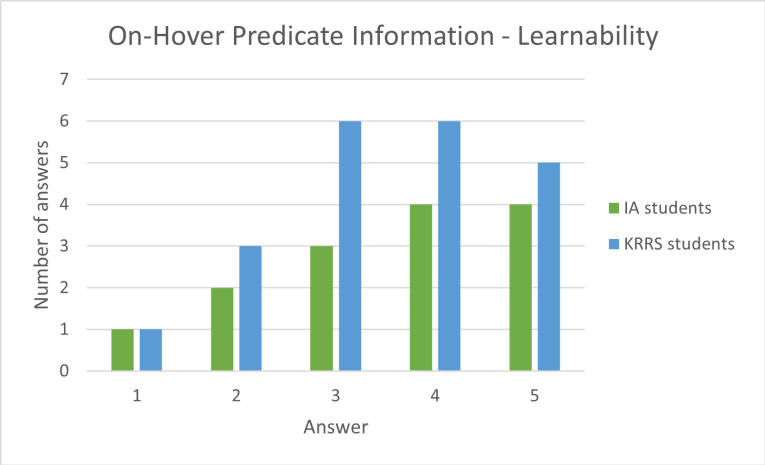
First of all, it must be noted that for this feature we are evaluating not only the fact that we can disable features, but also the usage of the config.json file and the usage of multiple files into one.

The average scores for these metrics were 3.1 for learnability, 3.3 for efficiency, and 3.2 for helpfulness. Here, it must be noted the dispersion of results for each metric. In learnability, the most common answer was 3 for the IA students, when for KRRS was between a 3 and a 4. For efficiency, for IA students was a 3 when for KRRS students was a 4, and for helpfulness for IA students was yet again a 3 when for KRRS students was between 3 and 4.

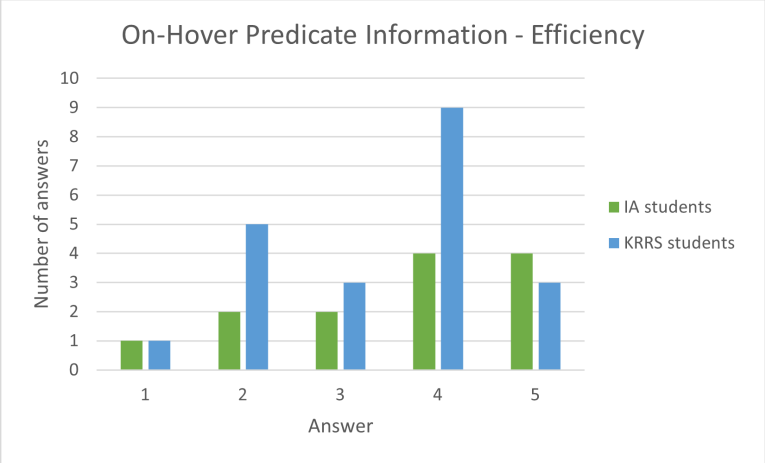
This feature, as seen in the results for learnability, was not very intuitive for students to understand, mostly on the IA group. Given that we expanded on the features of an existing extension, to keep these functionalities we had to keep their way to run multiple files with the config.json file. However some students found this to not be very intuitive, with a comment on the open question suggesting us to import them directly from a line in the file, similar to how it is done in the C programming language. While this suggestion could enhance how intuitive it is to import files, implementing it would require modifications to the file structure, which may would not be compatible with other ASP editors and could potentially cause confusion. Additionally, our extension needed to remain compatible with the two Visual Studio Code extensions that we are working with, making such alterations impractical.

Hand in hand with the results for learnability, the results for efficiency make sense that aren't high, specifically in IA students, given that the students did not find the feature to be easy to use.

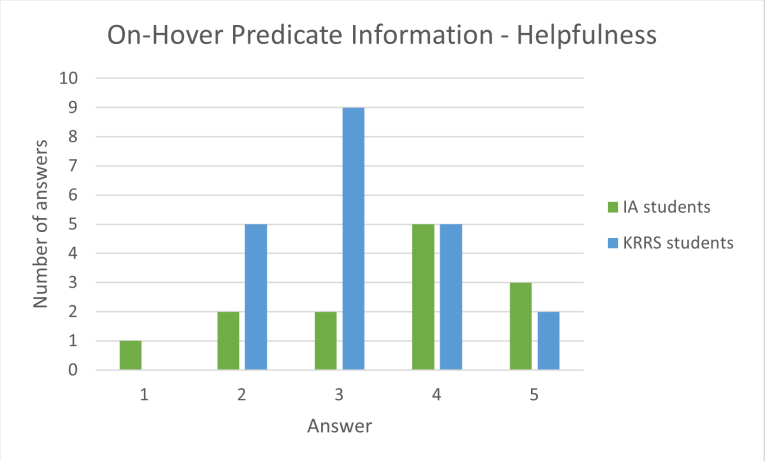
However, there is a bigger discrepancy in the helpfulness feature, because there are more answers of 3 for this metric in the IA students, and most answers for the KRRS students are positive. With this in mind, it should be noted that in the KRRS subject the students work with more advanced answer set programs, making the projects inherently more challenging. Consequently, the usage of multiple



(a) The results on how easy to understand the on-hover predicate information feature is

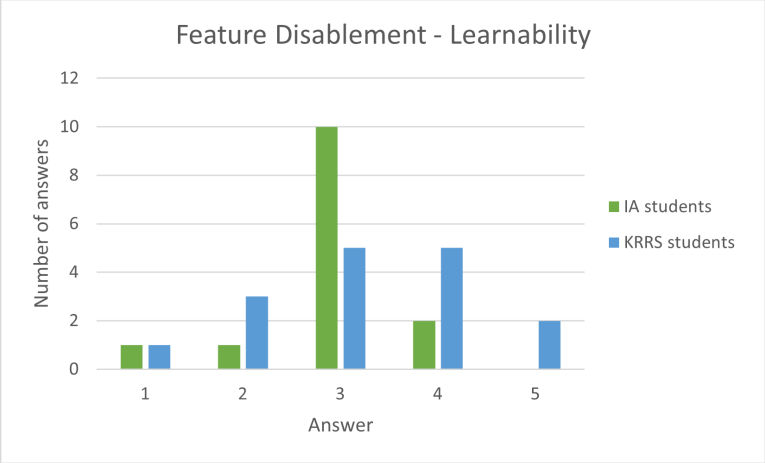


(b) The results on how practical to use the on-hover predicate information feature is

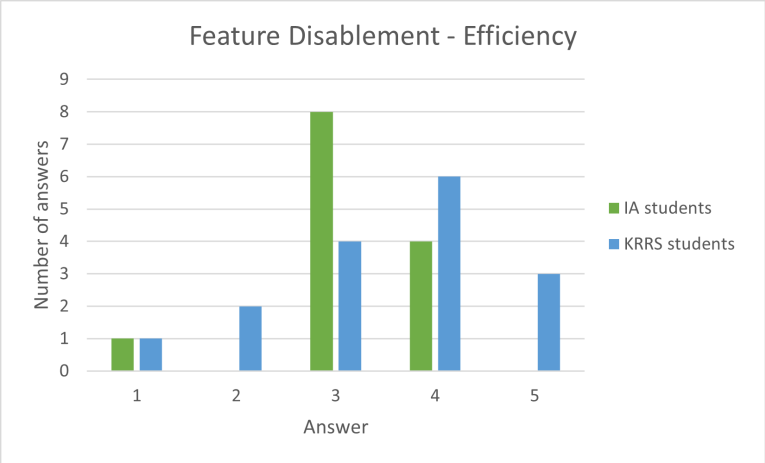


(c) The results on how helpful to use the on-hover predicate information feature is

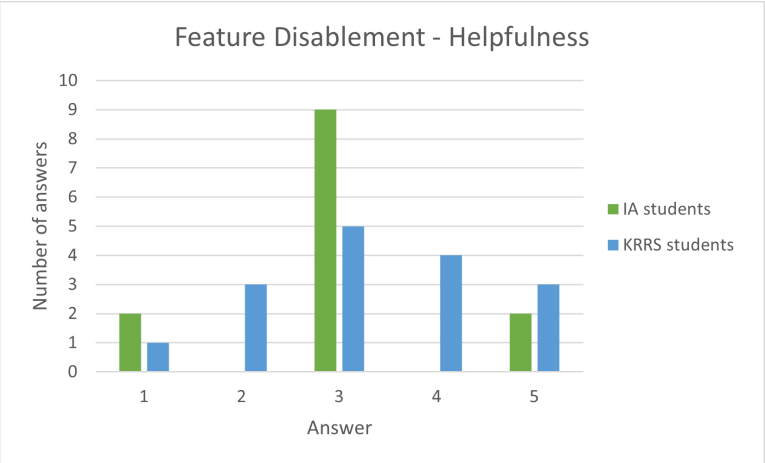
Figure 5.7: A visual representation of the results of the on-hover predicate information feature.



(a) The results on how easy to understand disablement of features is



(b) The results on how practical to use the disablement of features is



(c) The results on how helpful to use the disablement of features is

Figure 5.8: A visual representation of the results of the evaluation on the disablement of features.

files and the config.json file is be more useful, while the IA the exercises can more easily be completed with a single file.

We agree that for students who are starting to learn ASP, the config.json file is not the most efficient solution. However, since we are extending upon the functionalities of another extension, we believe that it is the best solution. Moreover, the results show that despite the difficulties for the AI course students, their feedback was positive enough for us to evaluate this feature positively.

5.3 Discussion

First of all, it is important to highlight that the average scores for each feature range from 2.9 to 3.9. This limited range is partly due to the relatively low number of responses received, which when combined with the averaging process that, to some extent, oversimplifies the data, it may not fully capture all potential perspectives. However, since the range is approximately between the numbers 3 and 4, we can conclude that the extension in general was well received, only with some features being more successful than others.

The feature that brought the most discussion around it was the Missing Comment Warnings, since it had the lowest score out of any feature in any metric, 2.9 for helpfulness. As discussed in Section 5.2.4 this was derived from the feature being somewhat intrusive, and therefore this feature was the only one we adapted based on the feedback by reducing its brightness, and disabling it by default. Interestingly, this feature also had the second highest score out of any feature, being learnability with an average of 3.8.

The feature with the highest score was the Predicate Validation, with a 3.9 for efficiency, and with it we understood that this feature was the most practical to use and, by looking at the other metric was also the feature that was better received in general.

Comparing the features with one another, the Predicate Validation was rated the highest, followed by Order Errors, Syntax Checking, On-Hover Predicate Information, Feature Disablement, and finally Missing Comment Warnings. This ranking is noteworthy as it suggests that, despite not being the focus of this dissertation, the Syntax Checking feature was appreciated by students, possibly indicating the potential for the implementation of a more complete checker with more detailed error messages. Additionally, the Predicate Validation and Order Errors features which were two of our more successfully evaluated features, both implementing the Easy Answer Set Programming methodology, positively suggest the methodology's utility in aiding students learning ASP. Finally, both features that incentivize documentation, namely Missing Comment Warnings and On-Hover Predicate Information, were harder to accept by the students, which combined with the open answers we got, shows that the users do have a resistance to commenting.

6

Conclusion

In conclusion, this dissertation set out to create a specialized tool aimed at assisting newcomers to Answer Set Programming in comprehending their encodings and promoting good documentation practices.

To achieve this objective, we developed a Visual Studio Code extension that implements the Easy ASP methodology and advocates for the importance of structured programming to improve comprehension and maintenance of answer set programs.

The extension incorporates error detection functionalities for identifying syntax errors, rule order discrepancies, and misplaced predicates. Additionally, it includes warning prompts for predicates lacking preceding comments, and provides hover messages displaying comments associated with each predicate. Furthermore, users have the flexibility to enable or disable specific features according to their preferences.

Through an evaluation process involving students from relevant courses, our extension was put to the test, with participants providing both quantitative and qualitative feedback.

The results yielded positive outcomes, affirming the effectiveness of the Easy Answer Set Programming approach in aiding newcomers to ASP in grasping the language. Nevertheless, the findings also highlighted a persistent reluctance among students to engage in commenting practices, suggesting areas for further improvement and refinement in future iterations of the tool.

6.1 Future Work

This work represents the initial step in the realm of support tools for ASP learners, with numerous other explorations to be possible.

Potential future projects can be the creation of individualized error messages for each syntax error, accompanied by suggestions on how to fix them. This extension of our work can be easily integrated into the code written in this dissertation, which makes it a very promising possibility.

Another potential direction, based on the work proposed by Lifschitz in [19], is the implementation of the verification of each block of code, as described in between comments, aligning with its intended functionality. This project could use technologies based on Artificial Intelligence via prompts which are now on the rise. It may even be possible to create one of these Artificial Intelligence bots specifically crafted to assist in Answer Set Programming problems. This possibility is more challenging given that

the AI system needs to be trained with many answer set programs, and that it also needs to have a proper understanding of what the user means with each comment.

Additionally, there is also potential for the development of a debugger for ASP programs, offering significant improvements to the management of large-scale projects. However, this avenue is the most challenging among the proposed projects, necessitating a profound understanding of ASP systems, debugger construction, and research into integration methods. Fortunately, some existing tools, such as Visual Studio Code's internal debugging features, could serve as basis for this project.

In essence, this dissertation is the first step in a multitude of possibilities aimed at enhancing the ASP learning and work experience.

Bibliography

[1] M. Alviano, C. Dodaro, and M. Maratea. "Shared aggregate sets in answer set programming". In: *Theory Pract. Log. Program.* 18.3-4 (2018), pp. 301–318. DOI: 10.1017/S1471068418000133. URL: https://doi.org/10.1017/S1471068418000133 (cit. on p. 1).

[2] A. Belcour. *Answer Set Programming Syntax Highlighting*. URL: https://github.com/ArnaudBelcour/asp-syntax-highlight (visited on 2023-11-10) (cit. on pp. 21, 22, 28).

[3] R. Bihlmeyer et al. *DLV User Manual*. URL: https://www.dlvsystem.it/dlvsite/dlv-user-manual/ (visited on 2023-02-05) (cit. on pp. 1, 17).

[4] A. Bochman. "Here and There among Logics for Logic Programming". In: *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Ed. by E. Erdem et al. Vol. 7265. Lecture Notes in Computer Science. Springer, 2012, pp. 87–101. DOI: 10.1007/978-3-642-30743-0_7. URL: https://doi.org/10.1007/978-3-642-30743-0%5C_7 (cit. on p. 13).

[5] J. Bordalo. *ASP Sudoku Solver*. URL: https://github.com/jbordalo/clingo-sudoku-solver (visited on 2023-01-16) (cit. on pp. 5, 12).

[6] P.-A. BUSONIU et al. "SeaLion: An eclipse-based IDE for answer-set programming with advanced debugging support". In: *Theory and Practice of Logic Programming* 13.4-5 (2013), pp. 657–673. DOI: 10.1017/S1471068413000410 (cit. on p. 2).

[7] R. Carnevali and F. Pacenza. *ASP Language Support for DLV2 System*. URL: https://github.com/fpacenza/dlv2code (visited on 2023-02-07) (cit. on pp. 3, 19).

[8] T. Eiter, G. Ianni, and T. Krennwallner. "Answer Set Programming: A Primer". In: *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*. Ed. by S. Tessaris et al. Vol. 5689. Lecture Notes in Computer Science. Springer, 2009, pp. 40–110. DOI: 10.1007/978-3-642-03754-2_2. URL: https://doi.org/10.1007/978-3-642-03754-2%5C_2 (cit. on p. 1).

[9] A. A. Falkner et al. "Industrial Applications of Answer Set Programming". In: *Künstliche Intell.* 32.2-3 (2018), pp. 165–176. DOI: 10.1007/s13218-018-0548-6. URL: https://doi.org/10.1007/s13218-018-0548-6 (cit. on p. 1).

[10] J. Fandinno et al. "Answer Set Programming Made Easy". In: *CoRR* abs/2111.06366 (2021). arXiv: 2111.06366. URL: https://arxiv.org/abs/2111.06366 (cit. on pp. 2, 6, 13, 14, 23).

[11] F. Frankreiter, R. Hegewald, and S. Killen. *ASP Language Support*. URL: https://github.com/CaptainUnbrauchbar/ASP-Language-Support (visited on 2023-11-10) (cit. on pp. 2, 3, 19, 21, 22, 27, 28, 35, 37).

[12] M. Gebser et al. *A user's guide to gringo, clasp, clingo, and iclingo.* URL: https://sourceforge.net/projects/potassco/files/guide/ (visited on 2023-01-23) (cit. on pp. 1, 17, 32).

[13] M. Gebser et al. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. DOI: 10.2200/S00457ED1V01Y201211AIM019. URL: https://doi.org/10.2200/S00457ED1V01Y201211AIM019 (cit. on pp. 5, 6).

[14] S. Germano et al. *LoIDE*. URL: https://github.com/DeMaCS-UNICAL/LoIDE (visited on 2023-02-09) (cit. on pp. 2, 18, 20).

[15] S. Hahn et al. *Clingraph*. URL: https://github.com/potassco/clingraph (visited on 2023-11-23) (cit. on pp. 24, 35).

[16] R. Henriques. *Github page of the EZASP Visual Studio Code extension*. URL: https://github.com/rmr-henriques/ezasp (visited on 2023-12-14) (cit. on pp. 3, 28).

[17] R. Henriques. *Marketplace page of the EZASP Visual Studio Code extension*. URL: https://marketplace.visualstudio.com/items?itemName=RamiroHenriques.ezasp (visited on 2023-12-14) (cit. on pp. 3, 28).

[18] A. Holyer. "Methods For Evaluating User Interfaces". In: (1993-11). DOI: 10.13140/RG.2.2.20290.94406 (cit. on p. 39).

[19] V. Lifschitz. "Achievements in answer set programming". In: *Theory Pract. Log. Program.* 17.5-6 (2017), pp. 961–973. DOI: 10.1017/S1471068417000345. URL: https://doi.org/10.1017/S1471068417000345 (cit. on pp. 2, 16, 51).

[20] J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987. ISBN: 3-540-18199-7. DOI: 10.1007/978-3-642-83189-8. URL: https://doi.org/10.1007/978-3-642-83189-8 (cit. on p. 1).

[21] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[22] *N-Queens problem*. URL: https://en.wikipedia.org/wiki/Eight_queens_puzzle (visited on 2023-02-09) (cit. on p. 16).

[23] OpenAI. *OpenAI's GPT-3 and ChatGPT*. 2023. URL: https://www.openai.com (visited on 2023-02-09) (cit. on p. 31).

[24] Potassco. *Running clingo*. URL: https://potassco.org/clingo/run/ (visited on 2023-01-20) (cit. on pp. 2, 18).

[25] *Potassco, the Potsdam Answer Set Solving Collection*. URL: https://potassco.org/ (visited on 2023-02-09) (cit. on p. 18).

[26] Pottasco. *Clingo Release*. URL: https://github.com/potassco/clingo/releases/ (visited on 2023-01-14) (cit. on pp. 2, 19).

[27] F. Ricca et al. "Team-building with answer set programming in the Gioia-Tauro seaport". In: *Theory Pract. Log. Program.* 12.3 (2012), pp. 361–381. DOI: 10.1017/S147106841100007X. URL: https://doi.org/10.1017/S147106841100007X (cit. on p. 1).

[28] D. S.R.L. *DLV download*. URL: https://www.dlvsystem.it/dlvsite/dlv-download/ (visited on 2023-01-29) (cit. on pp. 2, 19).

[29] *Sudoku*. URL: https://en.wikipedia.org/wiki/Sudoku (visited on 2023-01-20) (cit. on pp. 5, 12).

[30] *Visual Studio Code*. URL: https://code.visualstudio.com/ (visited on 2023-02-09) (cit. on pp. 2, 19–22, 35).