Lisbon University Institute

School of Technology and Architecture (ISTA)

Automatic Generation of Descriptions for Prolog Programs

Diogo Miguel Hortêncio Farinha

A Project presented in partial fulfillment of the Requirements for the Degree of Master in

Computer Science and Business Management

Supervisor:

Doctor Luís Miguel Pina Coelho Teixeira Botelho

Associate Professor of ISCTE-IUL

August 2019

Abstract

It is often hard for students and newcomers used to imperative languages to learn a declarative language such as Prolog. One of their main difficulties is understanding the procedural component of Prolog. Despite being a declarative language, Prolog allows for the creation of procedures whose structure is very different from the more common imperative languages. To tackle this issue, we try to facilitate code comprehension of procedural Prolog through the generation of formal and natural descriptions. First, we represent the workflow of Prolog encoded procedures through formal descriptions similar to imperative languages. To do this, we identify programming patterns that represent the basic blocks of certain classes of Prolog programs. Then we view more complex Prolog programs as coherent compositions of instances of the basic patterns. By using formal templates, we formally describe these individual patterns into an intermediate formal language. Afterwards, we generate natural language descriptions by using templates to describe the formal constructs. Using this two-step approach, we obtain two descriptions (one formal and one in natural language) that are both explanatory of the original program.

Resumo

Normalmente é difícil para alunos e iniciantes que estão habituados a linguagens imperativas, aprender uma linguagem declarativa como o Prolog. Uma das suas principais dificuldades é entender a componente procedimental do Prolog. Apesar de ser uma linguagem declarativa, o Prolog permite a criação de procedimentos cuja estrutura é bastante diferente da usada nas linguagens imperativas. Para abordar este problema tentámos facilitar a compreensão de código do Prolog procedimental através da geração de descrições formais e em linguagem natural. Primeiro, representamos a lógica dos procedimentos em Prolog através de descrições formais similares a linguagens imperativas. Para isto, identificamos os padrões que representam os blocos básicos de certas classes de programas. Depois, consideramos os programas mais complexos como composições destes padrões básicos. Através da utilização de *templates* formais, descrevemos formalmente estes padrões individuais numa linguagem formal intermédia. Seguidamente, geramos descrições em linguagem natural utilizando *templates* para descrever os construtos formais. Ao usar esta abordagem de dois passos obtemos duas descrições (uma formal e uma em linguagem natural) que são ambas explanatórias do programa original.

Keywords: Code comprehension, Prolog, Procedural Prolog, Natural language description, Formal description, Programming pattern, Natural language templates, Formal templates

Palavras-chave: Compreensão de Código, Prolog, Prolog procedimental, Descrição em Linguagem Natural, Descrição Formal, Padrão de Programação, Templates de Linguagem Natural, Templates formais

Acknowledgements

First, I express my sincere gratitude to Professor Luís Botelho as he was fundamental in the completion of this project. He was very supportive throughout the whole project and invested much of his personal time to ensure that I always did the best work that I can. Additionally, his insights and expertise were crucial contributions for this project.

I would like to thank all my colleagues at Novabase, particularly Válter Caldeira and Marco Vicente. They helped me immensely in balancing work with my studies. I did this entire project as a working student, which without their support might not have been possible.

I extend my gratitude to every person that helped with evaluating the results of the project, as without them, it would not have been completed.

Finally, I am very grateful to my girlfriend and my family for their patience and support throughout this whole endeavour.

Table of Contents

1. Introduction 1
1.1. Objectives and Motivation 2
1.2. General Approach and Main Contributions 3
1.3. Research Methodology 5
2. Background 6
2.1. The basics of programming in Prolog 7
2.2. Recursion in Prolog 10
2.3. Procedural Prolog 12
3. State of the Art Review 15
3.1. Prolog to Java Translator 17
3.2. Generation of Summaries for Java Classes 18
3.3. Generating Summaries for Crosscutting Code 19
3.4. Summarization of Method Context 20
3.5. Using Prolog for English Grammar 21
3.6. Machine Learning approaches 22
3.7. Patterns in Prolog 23
3.8. State of the Art Conclusions 24
4. Generation of Formal Descriptions 25
4.1. Programming Patterns and Formal Language Used 27
4.2. Discarded Approaches 32
4.2.1. Information Extraction 32
4.2.2. Handling Variables 34
4.2.3. *Fail-Loop* Identification 35
4.3. Adopted Approach 38
4.3.1. Target Program Access 38
4.3.2. Pattern Tagging 39
4.3.3. Pattern Transformation 42
4.3.4. Formatting 45
4.3.5. Rendering 46
5. Generation of Natural Language Descriptions 48
5.1. Natural Language Transformation 50

5.1.1. Procedures 51
5.1.2. Actions 52
5.1.3. Formal Constructs 53
5.2. Text Formatting 54
6. Compositionality 57
7. Results' Evaluation 62
7.1. Program 1 – Simple *fail-loop* 63
7.2. Program 2 – Fail-loop with embedded loops 64
7.3. Program 3 – Fail-loop with sequencing loops 64
7.4. Program 4 – Simple *repeat-loop* 65
7.5. Program 5 – *Repeat-loop* with embedded loop 66
7.6. Program 6 – Simple *list-iteration-procedure* 67
7.7. Program 7 – *List-iteration-procedure* with embedded conditional 68
7.8. Results summary 69
8. Conclusion 71
9. References 74

Table of Figures

Table 1 - Prolog to Java Translator review 17
Table 2 - Generation of Summaries for Java Classes review 18
Table 3 - Generating Summaries for Crosscutting Code review 19
Table 4 - Summarization of Method Context review 20
Table 5 - Using Prolog for English Grammar review 21
Table 6 - Machine Learning approaches review 22
Table 7 - Patterns in Prolog review 23
Table 8 - Formal description templates 31
Table 9 - Tags and respective patterns 40
Table 10 - Natural language templates 54
Table 11 - Results summary 69

1. Introduction

The title of this project succinctly presents our main goal, which is the automatic generation of descriptions of Prolog software. Students and professionals often have backgrounds in imperative languages which may hinder their Prolog learning experience. The main intention of these generated descriptions is to make Prolog software easier to understand. To do this, we propose a methodology comprised of two main steps, generating a formal description of the program and generating a natural language description of the generated formal description.

This chapter includes the sections Objectives and Motivation, General Approach and Main Contributions and Research Methodology. In Objectives and Motivation, we define in a more precise manner our objectives, what we want to do in this project, we contextualize the problem and explain the reasons why we tackled it. In General Approach and Main Contributions, we give more insight to our chosen methodology for reaching the defined objectives and why we chose it, as well as what we contribute with it. Lastly, in Research Methodology we explain the Design Science Research Methodology, applied to our specific case, which is the methodology we chose for all the research done throughout the dissertation.

1.1. Objectives and Motivation

The first interaction of most Computer Science university students with programming languages comes from imperative ones such as Java or C++. Prolog, being the abbreviation for “PROgrammation en LOGique” [1], is as the name implies, a declarative programming language based on logic. The paradigm shift from these languages to Prolog can be daunting for most students and for software professionals that are already used to the imperative way of approaching and solving programming problems [2]. The major concrete goal of this project is to automatically generate natural language descriptions of Prolog programs with the purpose of helping students and other software professionals used to imperative languages to better understand those Prolog programs. Besides shortening the learning curve for Prolog newcomers, we also tackle the problem of code comprehension which can be a time-consuming and expensive process [6,7,8,16,17]. We intend to generate two types of descriptions for each Prolog program, a formal description (presented in a style similar to imperative code) and a natural language description. The former is an intermediate formalism (but can still be used for learning purposes) and the latter is the more explanatory of the two. These two descriptions can be looked at as a set and should definitely be more understandable if looked at together, but we consider that each description is intelligible enough to stand on its own (perhaps the formal descriptions will be more intuitive for senior professionals while the natural language descriptions will be better suited for students and junior professionals).

Despite being a declarative programming language, Prolog has a procedural element that is peculiarly different from procedures found in imperative programming languages. The difference stems from the fact that Prolog does not have dedicated syntactic constructs to specify procedures (e.g., *for* or *while*). In this dissertation we focused on describing the procedural component of Prolog. This decision was taken because the specification of procedures in Prolog seems to be particularly difficult to understand to newcomers and non Prolog programmers. This might be the case because someone familiar with imperative languages will inevitably try to inadequately use previously learned procedural constructs to understand a totally different kind of specification.

Eliminating communication barriers between programmers and programs using natural language is deeply connected to the discipline of artificial intelligence, in particular, with natural language generation. Natural language generation aka NLG can be described as “the subfield of artificial intelligence and computational linguistics that is concerned with the construction of computer systems than can produce understandable texts in English or other human languages from some underlying non-linguistic representation of information” [4]. According to A. Gatt, E. Krahmer et al, natural language generation should be used “whenever information that needs to be presented to users is relatively voluminous, and comes in a form which is not easily consumed and does not afford a straightforward mapping to a more user-friendly modality without considerable transformation” [5]. As such, our goal of making Prolog code easier to understand is a relevant and appropriate problem for the use of natural language generation.

1.2. General Approach and Main Contributions

In order to generate descriptions from code we need to have a specific approach planned so that the various descriptions generated from a variety of different Prolog programs are always predictable, comprehensible and accurate. The proposed approach consists of two steps. The first produces a formal description of the Prolog program. The second converts the formal description to natural language. The intermediate formal description generated in the first step adopts a specification style akin to that used in imperative programming languages. Thus, the generated natural language explanations of the intermediate formal representation provide an imperative-like view of the original Prolog program. The decision to first generate an intermediate formal description of the Prolog program enables us to easily generate natural language explanations of other programs, possibly written in other programming languages. Once they get translated to our intermediate formalism, the natural language generation will most likely already exist. This is the approach often used in multi-language machine translation systems that translate between many natural languages [9].

Prolog programs, like all computer programs, vary in size. With regard to the larger, more complex programs we will approach them using compositionality. First, we identify programming patterns that represent the basic blocks of certain classes of Prolog programs and then we decompose more complex Prolog programs as coherent compositions of instances of these basic patterns. Using this approach, we only need to program and define formal descriptions for these small, template-like blocks of code. Afterwards, we unite these processed fragments in a coherent fashion to create a bigger formal description that corresponds to the original Prolog program. Once the basic patterns, corresponding to the building blocks of the formal description, are identified, the several kinds of composition operations used in the original Prolog code (e.g., sequence and several kinds of embedding) are preserved in the generated formal description.

For the natural language descriptions, we use the same approach that we used to generate formal descriptions. We decompose the formal description in fragments (fragmentation is now much easier since the formal description was generated from fragments to begin with) and we process these fragments individually, producing small excerpts of text. Once we have successfully decomposed and processed every single fragment of the formal description, we can easily generate a natural language description by joining all the excerpts together in a coherent fashion, preserving as before the original composition operators (e.g., sequence and several kinds of embedding). In this way we can produce bigger texts from bigger formal descriptions by carefully combining smaller phrases and paragraphs.

We intend to not only help students, software engineers and other software professionals to understand Prolog better, but also to contribute to the area of study of code comprehension. Code comprehension is a discipline of study that has evolved throughout the years [3]. We want to contribute to the discipline through the ideas and approaches used in this project. Using descriptions (both formal and in natural language) to facilitate code comprehension and bridging the gap between programmer/student and code.

Generating natural language from code is something that, despite uncommon, has been done before [6,7,8]. However, as far as we could find in the literature review, generating natural language from Prolog code has not yet been done. This is one of the main contributions of this project. The other important contribution is the two-stage compositional symbolic approach proposed in this dissertation. The State of the Art chapter shows that currently most of the more salient approaches to the generation of natural descriptions of program behaviour rely on machine learning algorithms. These must be trained with previously existing large corpora of examples associating program behaviour sequences (or alternatively, program code fragments) with possible natural descriptions [15,17]. Our approach, while avoiding the need for the mentioned training corpora, offers two additional advantages. Firstly, it is more generalizable than current machine learning approaches because it mostly works with totally abstract domain independent program constructs, while clearly separating a small and restricted set of domain dependent descriptions that can easily be input to the system. Our approach is particularly strong when considering that most works done usually have limitations. For example, some approaches focus on samples of code that may not represent the majority of existing code [6] while others are limited in scope [7]. Secondly, the second stage of the proposed approach – namely, generating natural language from intermediate formal descriptions – can be used to generate natural language descriptions of programs written in programming languages, other than Prolog (as long as the generation of the intermediate formal descriptions for the different programming languages is provided).

In this project, we adopt the perspective that the generation of both formal and natural language descriptions should be fully automated. In this case, fully automated is intended as the lack of user input in the creation of the descriptions. As we will see ahead, this is not always possible or ideal (see chapters 4 and 5), however, we still believe this stance is of most benefit to the users. By adopting the maximum automation stance, newcomers to Prolog will find it much easier to use it and to generate their own descriptions without needing any prior knowledge.

All of the code produced was done in Prolog and is available in GitHub at https://github.com/diogomfarinha/Prolog_Description_Generator. Since we want to generate descriptions for Prolog programs it makes sense to use Prolog, seeing as the use of other languages would most likely not yield results that could justify the added trouble. Additionally, our research group focuses mostly on using symbolic processing for problem solving which also lead us to the use of Prolog.

1.3. Research Methodology

In any serious, detailed research work, it is important to have a defined methodology to help guide the process. A methodology is chosen to increase motivation and focus and for the sake of preventing the research to be done haphazardly. It is crucial to have a set of principles, practices and procedures to structure the way research is done. Possibly, one of the most important gains of a research methodology is the definition of the evaluation methodology and criteria most adequate to assess the achieved results.

Design Science is defined as the creation of artefacts to solve an observed problem. Design Science includes as part of its methodology, making research contributions, evaluating the designs and communicating the results to the appropriate audiences. Summarily, Design Science “includes any designed object with an embedded solution to an understood research problem” [10].

For this project, we chose the Design Science Research methodology because the end goal is to generate software to solve a specific problem. Additionally, due to the subjective nature of generated descriptions, they need to be evaluated by experts. The process of evaluation in Design Science is to “Demonstrate the use of the artefact to solve one or more instances of the problem” [10].

We can structure the work for this project following the Design Science steps. First, we identify the research problem, our motivation, in this case, to facilitate Prolog understandability for software students and professionals. Second, we define the objectives of our solution/artefact, namely, we want to produce a program that can analyse Prolog programs and produce descriptions. Third, we design and develop the artefact, our Prolog software. Fourth, we demonstrate and evaluate the results, which will be done through the analysis of experts. Finally, we will try to communicate it through scholarly publications.

2. Background

In this chapter, we explain in as much detail as is relevant, how programming in Prolog works. First, we introduce the basics of Prolog, explaining what constitutes a Prolog program. After introducing the basics, we tackle recursion, one of the main cornerstones of programming in Prolog and probably what makes or breaks most people's understanding of Prolog. Lastly, we address procedural Prolog. Despite being a declarative logic-based programming language, Prolog has procedural elements. Procedural Prolog is coincidentally the main issue approached in this project. In this chapter, we do not explain the Prolog language in its entirety, we just explain the concepts we consider crucial for the reader to understand the later chapters.

2.1. The basics of programming in Prolog

Newcomers to Prolog soon discover that the task of writing a program is very different from what they are used to from other programming languages. A declarative language such as Prolog, does not require the programmer to explicitly define its workflow. It does require however, for the programmer, to define its logic. “The Prolog approach is more about describing known facts and relationships about a problem, and less about prescribing the sequence of steps taken by a computer to solve the problem” [11]. Being largely based on formal logic, specifically on first order predicate logic, programming in Prolog can be summarized in 3 simple steps:

- Establishing the program’s knowledge base with facts
- Increasing the program’s knowledge through the use of rules
- Questioning the program

The most basic, but important component of programming in Prolog is the definition of facts. Facts can establish relationships between objects and characterize them. The following examples followed by possible interpretations (marked with “%”, the Prolog line-comment) demonstrate this.

```
rare(gold). %Gold is rare
rare(silver). %Silver is rare
man(john). %John is a man
woman(maria). %Maria is a woman
son(john,maria). %John is Maria’s son
```

If we query the program it will answer based on its knowledge. “Queries are a means of retrieving information from a logic program” [12].

```
?- rare(gold).
true.
?- rare(rock).
false.
```

Note that the names of the facts (e.g., rare, son) and their respective arguments (e.g., gold, john and maria) are established by beginning with a lower-case letter, as beginning with an upper-case letter denotes a variable in Prolog.

```
?- son(john,Mother).
Mother=maria.
```

Facts can be easily used to establish relationships and make statements. There is no restrictions about the number or order of the arguments. Once an order is set though, it must be respected.

```
?- son(Mother,john).
false.
```

Being a logic programming language, programmers need only ask the question and Prolog will handle the logic. When querying using variables, Prolog will always try to produce true statements, so it will search for an answer that makes a true statement. When querying for more results Prolog will try

to find alternative ways of producing a true statement until no more are available. Using ";" on the interpreter console asks for an additional result when possible.

```
?- rare(Item).
Item = gold;
Item = emerald.
```

Using rules, we can complement the pre-existing facts and create new knowledge. To exemplify, we define a mother as a woman that has a son, no matter who it is (using '_' we can express that the fact is true no matter the value of the corresponding argument).

```
mother(Mother):-
      woman(Mother), % a woman
      son(_,Mother). % that has a son.
```

If we want to ask if Maria or John are mothers, we use the following Prolog queries in the interpreter console:

```
?- mother(maria).
true.
?- mother(john).
false.
?-mother(Mother).
Mother = maria.
```

Prolog predicates can have many clauses (rules or facts), making it possible to define more than one way a predicate can produce a true statement. For example, we can use a fact to state that Lisa is a mother and then we can use rules to define that a mother is a woman with a son or a woman with a daughter.

```
mother(lisa). %Lisa is a mother.
mother(Mother):- % A mother is
      woman(Mother), % a woman
      son(_,Mother). % that has a son.
mother(Mother):- % A mother is
      woman(Mother), % a woman
      daughter(_,Mother). % that has a daughter.
```

Rules can be used in a variety of ways together with facts. Prolog ensures that all results presented are correct according to its knowledge base. Programmers should take into consideration though, that in Prolog if a predicate has several rules or facts, Prolog will try to process them in the order they were written.

Using predicates to establish relationships is not only useful for querying but also for listing.

```
woman(barbara).
daughter(barbara,maria).
child(Child,Mother):-
      daughter(Child,Mother).
child(Child,Mother):-
      son(Child,Mother).
```

We can list all the children of a mother using the child relationship. In this case, we defined a child of a mother as being her son or daughter.

```
?- child(Child,maria).
Child = john ;
Child = barbara.
```

Prolog offers a rich set of datatypes, although it is not possible to specify the types of the variables. The most commonly used simple datatypes in Prolog are atoms and numbers. Atoms are any sequence of characters (enclosed or not in single quotes). Atoms are sequences of characters that begin with a lower-case character or sequences of characters (upper and lower-case) enclosed in single quotes. Examples of atoms include john, maria, p90, pROLOG and 'Prolog'. Numbers also exist in Prolog, both integer and floating-point numbers. Examples of numbers include 100, 0, -10.6, 23.

Predicates are generally referred to by their named followed by slash ('/') and the number of arguments (technically called the predicate arity). For example, the child predicate would be referred to as *child/2*.

2.2. Recursion in Prolog

Recursion is the backbone of Prolog programming. A programmer's competence in using Prolog is highly related to how well he or she can use recursion and recursive definitions. "In Prolog, recursion is the normal and natural way of viewing data structures and programs" [11]. The concept of recursion can be troublesome to figure out. After all, it means we define a predicate in terms of itself.

To exemplify recursion, we will use a very common data structure, a list. A list is a sequence of ordered elements. In this case, ordered, means that the order in which the elements are placed matters. In Prolog a list is just a structure and the elements don't necessarily have to be of the same type (they can be atoms, numbers and any other kind of term, including other lists). Examples of lists include [1,2,7,4], [a,b,c,d], [a,1,b,4,100,foo, [X, Y]].

In Prolog, every list that is not empty has a head and a tail, the head being the first member of the list and the tail being the list of the other elements.

```
?- [1,3,4,a,b]=[Head|Tail].
Head=1,
Tail=[3,4,a,b].
?- [a,1]=[Head|Tail].
Head=a,
Tail=[b].
?- [3]=[Head|Tail].
Head=3,
Tail=[].
?- []=[Head|Tail].
False.
```

After understanding lists, we define a basic Prolog predicate, *member/2*. *member/2* is a predicate that returns true if the specified element belongs to the specified list and false otherwise.

```
?-member(2,[1,2,3]).
true.
?-member(4,[1,2,3]).
false.
```

We define the *member/2* predicate using the following statements. X is a member of a list whose head is X or X is a member of a list where X is a member of its tail.

```
member(X,[X|_]).
member(X,[_|Tail]):-
      member(X,Tail).
```

The predicate *member/2* is defined with a fact and a rule. In the rule, member/2 is defined recurring to itself, we call this recursion. To further exemplify recursion, we will also define *countElements/2*. This predicate will count the number of elements on a list (both numbers and atoms). Following the same line of thought from the member predicate we will define *countElements/2* using

head and tail. To define the predicate, we can use the following statements: an empty list has 0 elements; the number of elements in a non-empty list is 1 plus the number of elements in its tail.

```
countElements([],0).
countElements([_|Tail],Count):-
      countElements(Tail,CountTail),
      Count is 1+CountTail.
```

Alternatively, if we didn't want the predicate *countElements/2* to succeed for empty lists, we could also replace the first statement with the following one: a list with a single element has 1 element. (Now, this definition does not work (it fails) in the case of empty lists!)

```
countElements([_],1).
```

If we test it on the console:

```
?- countElements([1,a,b,c],Count).
Count=4.
```

Prolog tries to make the statement true and answers that Count is 4, since the list [1,a,b,c] has 4 elements. Testing for an empty list using the first definition we would get:

```
?- countElements([],Count).
Count = 0.
```

Using the second definition on the interpreter:

```
?- countElements([],Count).

false.
```

2.3. Procedural Prolog

Prolog is essentially a declarative language. Programming in Prolog revolves around querying for true statements instead of defining the program's control flow. Prolog does however, have a procedural element that is paramount to this project, as most descriptions will be generated for Prolog programs of procedural nature.

One of the most used procedures in imperative programming is the loop. Loops are a means for iterating through data structures and/or to execute actions and procedures in a repetitive way. In imperative languages, loops are commonly specified using special syntactic constructs (e.g., *while* or *for*) that, when compiled (or interpreted), are translated into (or executed as) a repetition of a sequence of instructions. In Prolog, however, those syntactic constructs do not exist, instead there are predicates to achieve iteration. The predicates more commonly used for causing Prolog execution to repeat (if the use of recursion is not adequate) are *repeat/0* and *fail/0*. For the sake of simplicity, we will call loops that use the *repeat* predicate "*repeat-loops*" and loops that use the *fail* predicate "*fail-loops*", although *repeat-loops* can only repeat while there is failure (and can end up not repeating at all).

Both loop types use the general computational process known as backtracking. When the execution of a Prolog program fails (or when the user asks for a possible alternative), the Prolog program execution backtracks in its logic and tries to re-satisfy its goals to produce a true statement [11].

```
?- rare(Item).
Item=gold;
Item=emerald.
```

When we use the semicolon (";"), asking Prolog for a possible alternative, Prolog tries to produce a true statement in an alternative way. Backtracking fails when there is no new way to satisfy its goal.

repeat/0 is a predicate that succeeds and that will always generate successful alternatives when backtracked to. Combining the *repeat* predicate with the way backtracking works in Prolog can create a loop that only stops when some desired condition is met. *repeat* must be used to repeat computations that would otherwise not have alternative solutions, usually in read-and-process loops.

```
input_password:-
    repeat,
    write('Input password:'),
    read(Input),
    password(Input).
password(12345).
```

The procedure[1] above will keep writing “Input password:” on the console and asking for user input until the user types the correct password “12345”.

```
?- input_password.
Input password: hello.
Input password: goodbye.
Input password: 12345.
true.
```

The program only ends (successfully) when the user inputs the correct password, otherwise it will keep looping (in this case, repeating) infinitely.

The *fail-loop* is also based on failure (as the name implies) but works differently from the *repeat-loop*. The *repeat-loop* will keep going forever while there is failure as it always finds an alternative solution when backtracked into. The *fail-loop* only runs while there are alternative solutions. The loop’s structure is comprised of a clause with at least one condition (one goal) that may potentially have more than one solution and a *fail/0* at the end. When Prolog arrives at the fail statement it fails, and so, it backtracks until an alternative solution is found (or no more backtracking is possible). The *fail-loop* is used to exhaust every solution possible of a specified condition (goal).

```
status(anna, online).
status(john, online).
status(michael, offline).
status(maria, offline).

displayStatus:-%first clause Prolog tries to execute
    status(Person,Status),%fact has more than one solution
    write(Person),write(' is '),write(Status),nl,%prints
    fail.%fails and backtracks
displayStatus.%When first clause fails, it tries to succeed with this one
```

If we execute the procedure *displayStatus/0* on the console, the following output will be generated:

```
?- displayStatus.
anna is online
john is online
michael is offline
maria is offline
true.
```

The use of *repeat* and *fail-loops* is very useful to approach problems that sometimes recursion can’t be used or is not the ideal solution. The use of these two predicates will be delved into more deeply in later chapters.

Variables in Prolog are not used in the same manner as in imperative languages. Since all statements in Prolog are evaluated for their truth, the common programming statement X=X+1 fails, as a number cannot be equal to itself plus one. In Prolog, the predicate *is/2* tries find a way that its two

[1] Although it is not uncommon to call predicate to any Prolog defined routine, we use the designation “predicate” only for Prolog definitions that set a property of a given entity or a relation between entities. We use the designation “procedure” for definitions that specify procedures.

arguments have equal values. In the case below, Prolog tries to ensure *is* is a true statement and so Count, presents 1+CountTail.

```
countElements([],0).
countElements([_|Tail],Count):-
      countElements(Tail,CountTail),
      Count is 1+CountTail.
```

Since it is impossible to change the value of a Prolog variable, Prolog programmers must instead dynamically create or modify facts, in the program database, that hold the desired quantities. The Prolog actions *assert/1* and *retract/1* can be used to create and remove clauses from the program database. In particular, they can be used within a procedure as a way to initialize and continuously update desired values.

```
?- assert(worker(tim)).
true.
?- worker(Worker).
Worker = tim.
?- retract(worker(_)).
true.
?- worker(Worker).
false.
```

3. State of the Art Review

To motivate the choice of the classes of related work that we have included in this overview, we begin this chapter with a brief summary of the general approach we sketched for solving the problem addressed in this dissertation. Our initial analysis of the targeted problem led us to propose to start with generating an intermediate formal description of the Prolog program to be processed. Then, the intermediate formal description is converted in possibly alternative natural language descriptions. This approach allows both to facilitate the generation of natural language descriptions of programs written in different languages, and also to facilitate the generation of alternative natural language descriptions in the same language (i.e., in English) or to produce descriptions in different natural languages (e.g., Portuguese). Another component of our approach consists of using compositionality. That is, more complex programs will be analysed as compositions of less complex blocks; formal descriptions of more complex programs will result of the composition of formal descriptions of less complex programs; and finally, natural descriptions of more complex intermediate formal descriptions will also result of compositionality. We are also interested finding out whether there is a simple mapping among the composition operators ate the three levels of processing. This state of the art review lends significant support to our envisioned proposal and, at the same time, helps to improve our approach by learning from other investigative endeavours.

The state of the art review pertains mostly to Prolog code patterns, code comprehension and natural language generation. Code patterns relate to pattern recognition in code which we use, together with compositionality, to simplify the production of formal descriptions. Code comprehension is the main field of study of this review and relates to our objective of making Prolog code easier to understand. Natural language generation is a field with many uses [5]. We are mostly focused on using it for purposes of code comprehension, namely, to generate natural language descriptions of intermediate formal descriptions. All topics are also delved into in the later chapters.

In terms of search strategy, specific criteria were defined for the topics in addressed in this project. The electronic databases used were mainly IEEE Xplore, ACM Digital Library, ScienceDirect-Elsevier and Springer, with the aid of Google Scholar. The title, abstract and keywords were queried for “Prolog”, “code comprehension”, “natural language”, “natural language generation”, “code patterns”, “design patterns” and combinations of these search strings (mostly with the “Prolog” search string). Occasionally, additional search strings were used in conjunction with the main ones, for instance, “Prolog” and “imperative”. In a first stage, we searched the Internet without temporal constraints. Since the number of achieved interesting results was relatively small, we decided that, in the general case, we would not specify time constraints in the searches that followed. Despite that, articles about general topics like natural language generation and all articles relating to machine learning, were filtered by recency so as to prevent excessive information and to stay on top of the current body of knowledge.

For the purpose of facilitating the evaluation and critique of the articles a taxonomy will be used. A taxonomy allows for a methodical, strict evaluation of all articles under equal criteria. In doing so, we

reduce haphazardness in our review and facilitate comparison between the articles. All articles will be evaluated in relation to the following criteria. Relation to Prolog (is the article directly related to Prolog?). Paradigm shift to imperative (does the article contain declarative to imperative conversion?). Code comprehension (does the article approach code comprehension?). Use of natural language generation (does the article address natural language generation (aka NLG)?). Approach used (Is the approach used similar to the 2-step natural language translations?).

Some of the articles were reviewed individually while others as a group. The individual critiques relate to those whose relevancy or density of information required an individual review. The articles that were grouped contain approaches that are very similar in nature and/or content.

3.1. Prolog to Java Translator

This article by Mutsunori Banbara, Naoyuki Tamura and Katsumi Inoue was written in 2006 with the purpose of presenting *Prolog Café* [13]. *Prolog Cafe* is a system that translates Prolog into Java using the WAM (aka Warren Abstraction Machine a Prolog compiler). *Prolog Cafe* surpasses previously existing benchmarks in Prolog to Java translation in portability and performance speed. The conversion roughly consists of converting Prolog terms to predefined Java classes (ex: VariableTerm, ListTerm) all united under an abstract Java superclass Term. Prolog predicates are also translated to Java by converting them to numerous classes based on the number of clauses and other factors.

Table 1 - Prolog to Java Translator review

Prolog	To imperative	Code comprehension	NLG	Approach
Yes	Yes	No	No	Conversion of terms and predicates to classes using a compiler.

Prolog Cafe is a system designed to convert Prolog code to Java using the WAM compiler[2]. It is concerned only with translating the logic and functionality of the Prolog programs to Java and does not consider code comprehension. Translating just the code's functionality does not equate to making it easier to understand. We aim a more emphatic focus on code comprehension. Making the shift from Prolog (procedural or not) to our imperative-style formal descriptions yields benefits in understandability. Besides, we also strengthen our approach by using natural language descriptions. As such, we consider that *Prolog Cafe* is not the best source of inspiration when the major goal is facilitating the programmers' understanding and learning of the language.

[2] Warren Abstract Machine, the most used model for Prolog compilers

3.2. Generation of Summaries for Java Classes

This paper presents a technique for automatic generation of natural language documentation for Java classes [6]. Due to the fact that documentation is not always present in code, programmers that are unfamiliar with it sometimes struggle and/or waste more time than necessary to understand it and assess its relevance for the task they face. To facilitate the process of documentation, making it automatic, a technique of summarization specific for Java code is proposed. The summarization works as follows. Both methods and classes are classified into stereotypes. These stereotypes were created according to common programming conventions and purposes. Each stereotype classified has an implicit programming intention. The contents of each class are then filtered based on their stereotype, this is done to guarantee the summaries are concise and only have the necessary and most important information. Finally, text is generated based on the filtered information. The text contains a general description of the class, a description of its stereotype declaring its intent and a description of its behaviour, describing the methods considered relevant. Lastly, the texts generated were evaluated by programmers and attained encouraging results.

Table 2 - Generation of Summaries for Java Classes review

Prolog	To imperative	Code comprehension	NLG	Approach
No	No	Yes	Yes	Classification of stereotypes, filtering and NLG

According to the article, the summaries produced were considered easy to read and understand by more than half of the participants and showed promising results on content adequacy and conciseness. This approach yielded good results but can still be improved upon. The identification of code stereotypes is similar (perhaps even equal in terms of thought process) to our proposed identification of patterns. We propose to improve upon this idea by using compositionality to improve understandability and facilitate the handling of bigger and more complex programs. In terms of actual descriptions produced, we also intend to add an intermediate formalism which not only opens the doors for more languages to be described in the future, but also complements the natural language descriptions bridging the transformation from code to text.

3.3. Generating Summaries for Crosscutting Code

Programmers often exert substantial effort analysing code so that they can identify the code that is appropriate for a given task, particularly when the code is crosscutting concern code (concern that affects or relies on many other parts of the system) [7]. To aid programmers in handling this difficult code, an automated natural language summary is generated. The approach used consists of three steps. The first step is information extraction, structural facts about code elements are extracted so that they may be used in the summaries. Information extracted describes the way methods interact with each other and with the rest of the system. The second step is generating abstract content. The extracted information is processed to find similarities between methods of the concern and source code elements deemed important. The final step is the generation of sentences for the summary. The concern summary includes four main parts: the listing of the methods, the description of the concern, the identification of salient code elements and the path of the method. The evaluation made to this approach notes that programmers have an easier time finding the appropriate code and perceive the task as less difficult.

Table 3 - Generating Summaries for Crosscutting Code review

Prolog	To imperative	Code comprehension	NLG	Approach
No	No	Yes	Yes	Generation of abstract content, generation of summary

The generation of abstract content to later generate the description parallels our use of an intermediate formalism. However, while these steps represent a similar idea, they are different in actual use. The abstract content generated is much rawer in nature and functions as a way to organize and structure the extracted information so that it can be transformed into a proper summary. We propose to go one step further and use this intermediate step not only as bridge between code and description but also as an actual stand-alone description. By using this description, we can complement our natural language one or even make it an alternative (perhaps more senior professionals will only need the intermediate level of abstraction to quickly understand the code. The organization of the natural language description in parts is also similar to our approach (even if it is applied to Java instead of Prolog).

3.4. Summarization of Method Context

Programmers rely on good software documentation to have a better and quicker understanding of code [8]. The process of documentation, however, is one that is not always done resourcefully, mainly due to how time consuming it is. Automatic documentation generators have been developed but they lack one important aspect, the context. Descriptions of methods without the surrounding context are considered incomplete and not sufficiently explanatory according to P. McBurney, C. McMillian et al (2014). The approach proposed in this paper corrects the mentioned shortage by including context in the methods' descriptions. The summary of a method is created in three steps. First, for the selected method, the most important methods in its context are found. Then, keywords and actions used by these most-important methods are extracted. Finally, this data is used to generate English sentences that describe what the method does and its context. It was found in their study that programmers benefited from these contextual descriptions.

Table 4 - Summarization of Method Context review

Prolog	To imperative	Code comprehension	NLG	Approach
No	No	Yes	Yes	Context analysis, keyword finding, generation of summary

This approach improves upon already existing descriptions by giving them context, making them more valuable. Despite recognizing the advantage of including context in the generated descriptions, it is not something that we plan to do at this moment, especially because of time constraints. In order to have descriptions to improve upon with context, we first need to generate those descriptions and that is what we prioritize right now, since there is no such work done for Prolog. We did add context components to some actions in the natural language descriptions, but they were very small in scope and did not pertain to the whole description. Nevertheless, we definitely consider context to be an important semantic source for the generated descriptions that we will consider for future work.

3.5. Using Prolog for English Grammar

Natural language processing is an interdisciplinary branch of artificial intelligence and linguistics that focuses mainly on deriving meaning from natural language inputs and producing natural language outputs [14]. Prolog can be particularly useful for natural language processing due to its declarative semantics, built-in search and pattern matching. A comparison between Prolog rules and grammar rules can be made, making Prolog an intuitive tool for language parsing. “Parsing is the process of converting a sentence into a tree that represents the sentence’s syntactic structure”. This structure reflects the grammar rules that can be defined in Prolog. As such, it is relatively straightforward to develop Prolog software that evaluates the grammatical validity of sentences.

Table 5 - Using Prolog for English Grammar review

Prolog	To imperative	Code comprehension	NLG	Approach
Yes	No	No	Yes	Parse tree, grammar rules

The process of creating a parse tree for a phrase is similar to the idea of compositionality, seeing as it coherently decomposes a bigger sentence into smaller parts. This paper demonstrates the suitability of Prolog for problems of natural language processing as well as using compositionality to solve problems. Despite focusing on the challenge of analysing natural language, it lays the foundation for text generation using Prolog. We want to continue working in that direction by using Prolog in our approach to generate descriptions.

3.6. Machine Learning approaches

Costa, Ouyang, Dolog, and Lawlor (2017) explain the use of recurrent neural networks using long-short term memory to generate reviews of items with explanations [15]. The items reviewed were beers and they were evaluated on appearance, aroma, palate, taste, and overall. The model was trained with approximately 1.5 million reviews. Using the aforementioned approach, the authors managed to generate reviews that were close to user-written reviews including misspellings and domain specific language.

McBurney, Liu, McMillan and Weninger (2014) demonstrate the use of topic modelling for code summarization [16]. The topics are organized into a hierarchy to condense information and to take into account the way programmers understand code. The idea is that programmers look at the code top-down, first analysing its high-level functionality and then looking at the lower-level, which supports the higher-level ones. The approach used can be summarized in four steps. First, software is represented as a call graph. Second, the call graph is prepared for topic modelling. Third, topic modelling is used. Finally, the hierarchical structure of the topics is displayed in a web interface.

Iyer, Konstas, Cheung and Zettlemoyer (2016) once again tackle the problem of code comprehension by using long-short term memory neural networks with a global attention mechanism [17]. The paper points out the huge quantity of code and appropriate comments stored in online repositories. The dataset uses almost a million posts from *StackOverflow* to train the model and generate descriptions for C# code snippets and SQL queries.

Table 6 - Machine Learning approaches review

Prolog	To imperative	Code comprehension	NLG	Approach
No	No	No [15] Yes [16,17]	Yes [15,17] No [16]	Topic modelling, long-short term memory networks [15,17]

Two of the reviewed machine learning approaches require large corpora of examples to be trained with [15,17]. The hierarchical topic modelling approach although explanatory in nature, is not as descriptive and informative as natural language summaries [16]. On one hand, the use of machine learning can have many benefits as is demonstrated by the aforementioned articles. On the other hand, machine learning needs large datasets and quality data in order to produce good results. Since we want to help with code comprehension not only for professionals but also for students, machine learning is not ideal. Machine learning needs a dataset whose recency and quality may be easily compromised, and the use of long-short term memory neural networks may be too convoluted for students' use. Our approach is also more generalizable since it uses totally abstract domain independent program constructs and by using our intermediate formal descriptions, we leave the doors open to easily describe other programming languages, which is not the case for the referred neural network approaches.

3.7. Patterns in Prolog

Kumar (2002) exposes a compilation of common misconceptions and pitfalls that imperative programmers are susceptible to when programming in a declarative language such as Prolog [2]. The paper also offers a series of programming patterns in Prolog to help students solve specific types of problems. Through the use of these patterns the students demonstrated superior performance in Prolog programming.

Sterling (2002) presents two classes of patterns for programming in Prolog: *Skeletons* and *Techniques* [18]. "*Skeletons* constitute the essential control flow of the program, and need to be understood procedurally". *Techniques* "capture basic Prolog programming practices, such as building a data structure or performing calculations in recursive code. Unlike skeletons, techniques are not programs but can be conceived as a family of operations that can be applied to a program to produce a program." Students' performance in Prolog increased when introduced to these patterns.

Le and Menzel (2005) demonstrate the use of a constraint-based modelling approach to diagnose and identify errors made by learners of logic programming [19]. "A pattern is a standard way to solve a recurrent problem" [19]. Patterns in Prolog are used to create constraints. According to Le and Menzel (2005), constrains have two parts, a relevance and a satisfaction. "The first part identifies the structural elements, for which a constraint is relevant. The latter examines if these elements satisfy the conditions of a constraint" [19]. This way we can create if statements "if pattern then conditions". Code that fails these statements can be diagnosed as erroneous. This constraint-based approach was used for a web tutoring program with positive results.

Table 7 - Patterns in Prolog review

Prolog	To imperative	Code comprehension	NLG	Approach
Yes	Yes [2] No [18,19]	Yes	No	Programming templates/patterns, Constraint-based approach [19]

The article by A. Kumar [2] is very useful for understanding the differences between Prolog and common imperative languages and helpful for the creation of our intermediate imperative-style descriptions. The approaches presented throughout the three papers are supportive of our own approach of recognizing programming patterns in Prolog. All 3 papers report beneficial results from introducing programming patterns as a learning tool [2,18,19]. The use of programming patterns and standardized ways of solving problems is very useful to help students and newcomers which is encouraging, however we can still improve on its effectiveness by utilizing descriptions to further cut down on the learning curve.

3.8. State of the Art Conclusions

The approach of using natural language descriptions to explain code and further enable code comprehension has been done before and mostly with encouraging results [6,7,8,15,17]. Despite being a mainly a declarative language, Prolog code can be successfully translated into imperative code [2,13]. The use of intermediate descriptions for natural language conversion has also been done successfully [7]. Due to Prolog's intrinsic qualities as a logic-based language it proves itself to be suitable for natural language processing as well as using compositionality in its solutions [14]. Finally, programming patterns in Prolog have been used for educational purposes and have helped many students and newcomers to understand the language better and quicker [2,18,19].

From the papers studied we extracted invaluable knowledge that supports our general approach and also enables its improvement. As stated, before on the critiques of individual research works, we not only take elements from the approaches used but improve upon them, so that we can produce a working system that satisfyingly supports code comprehension while leaving doors open for more varied and complex uses in the future. These future uses may include describing different programming languages and/or generating descriptions in various natural languages.

4. Generation of Formal Descriptions

The generation of formal descriptions is an important part of our two-step approach to generate natural language descriptions. The use of this intermediate formalism leaves the door open for other programming languages to be easily described in natural language, since as long as they can be translated into this formalism, the transformation into natural language will be assured. Additionally, these descriptions can also be more effective than natural language for more senior programmers. Looking at code, particularly, familiar looking code, can be faster and more evident than reading text in natural language depending on the reader.

Creating descriptions of Prolog programs in an imperative style can be very difficult if not impossible to do. Translating a declarative language which hinges on its logic being expressed through relationships into a control-flow like description can not only be unmanageable but also a questionable choice. Prolog, however, does have a procedural element that is often harder to understand for students and Prolog newcomers. Aside from recursion, Prolog offers two broad programming patterns that implement repetitive processes, both of which rely on the basic mechanism of repetition by failure. One of these two patterns is used when a certain fragment of the computation generates alternative solutions upon backtracking. This programming pattern uses failure to cause the execution of the program to loop through all alternative solutions of that multi-solution computation fragment. For simplicity, we will refer to this pattern as *fail-loop*. The other pattern is used when no fragment of the computation generates alternative solutions. In such cases, the repetition is achieved by the introduction of a special purpose built-in predicate, called *repeat*, that plays the role of a computation fragment with an indefinite number of alternative solutions. When the *repeat* predicate is introduced, the repetition is also caused by failure as in the previous code pattern. However, since the introduced *repeat* has an indefinite number of solutions, we cannot repeat while there are alternative solutions – that would lead to an indefinite repetition. Thus, instead of a simple failure, this pattern includes a computation fragment that fails in some circumstances and succeeds in others. This way, the computation repeats until the referred computation fragment succeeds. We call this the *repeat-loop* pattern. The description of these Prolog programs through the use of familiar imperative constructs makes it easier for beginners to understand their control flow.

A major challenge of this intermediate step is that some relational elements cannot be translated into an imperative-style language. Even though we are largely focused on the procedural element, there are inevitably some relational elements that cannot be ignored. To handle those cases, we created our own formal language for intermediate representation, which proved to be sufficiently self-evident in the results' evaluation.

Despite being an intermediate step, this by no means, denies the validity and accuracy of the description itself. The natural language description is generated from this description only and not the original code. Furthermore, both descriptions, formal and natural language are supposed to be sufficiently explanatory by themselves (even if they yield better results when taken together).

The general approach to generating formal descriptions from Prolog programs is comprised of five sequential steps. Target Program Access, Pattern Tagging, Pattern Transformation, Formatting and Rendering. By Target Program Access, we mean accessing the source code of the target program to be described, that is, making the target program's clauses available for further processing. Pattern Tagging is the identification, through the use of tags, of specific code fragments that belong to programming patterns. Pattern Transformation is the transformation of tagged code fragments into formal constructs resembling those of imperative programming languages. This transformation into imperative language constructs is done through the use of templates. For every tag, there is a pre-defined formal construct template. Formatting handles the transformation of certain Prolog predicates and procedures so that they are closer to what is commonly used in imperative languages, as well as defining the proper indentation for each formal construct. Lastly, Rendering encompasses the production of a readable formal description, from the transformed constructs and predicates using the corresponding indentation.

In this project, we adopt the perspective that it should be possible to fully automate the generation of convinent descriptions of Prolog programs, using only their structural / syntactic properties. We know that this is not always possible because we have faced several cases in which a convenient description can only be generated if the generation process uses semantic information about the code being described (see section 4.2.3). Nevertheless, in this project, we always try to push the syntactic stance as further as possible. Thus, we try to always define the most general possible methods to fully automating the description generation process from syntax alone. Given that we know that the used general criteria do not always lead to correct decisions, users can explicitly provide more semantic information about their code (see section 4.3.2).

This chapter has three main sections, some with corresponding subsections. The first section explains the formal language used in this project. All the formal descriptions generated are created by description templates utilizing this formal language. The second, presents the most relevant initial approaches that were later discarded, as well as the reasons for having abandoned them. The third, presents the detailed description of the final approach that was adopted and integrated all the stages of generation of formal descriptions.

4.1. Programming Patterns and Formal Language Used

Generating formal descriptions first requires a formal language to be defined. Our approach uses templates to translate the programming patterns into formal constructs. The use of templates makes it so that we can generate accurate descriptions for the same programming patterns regardless of the input programs. Each programming pattern we handled has a specific template with the proper formal equivalent. In this project we focused on four programming patterns. This selection was made due to their importance in understanding procedural Prolog, as well as time constraints. Two of the patterns chosen are the aforementioned *fail-loop* and *repeat-loop,* as they are the backbone of procedural Prolog. The next, is a procedural pattern in which a Prolog program recursively iterates through a list and performs actions. We called this pattern *list-iteration-procedure*. Finally, we handled some Prolog code fragments that can be described as *if-clauses*, which may not be as obvious to the untrained eye.

Despite being few, each pattern can have its own details to deal with. In the case of *fail-loops* and *repeat-loops* we can have nested loops inside one another as in the examples below:

```
displayStudents:-
    university(U),
    department(U,D),
    student(D,S),
    write(S),nl,
    fail.
displayStudents.
```

```
inputCredentials:-
    repeat,
    write('Username: '),nl,
    read(Name),name(Name),
    write('Password: '),nl,
    read(Password),
    password(Name,Password),
    write('Credentials accepted.').
```

The first program uses a *fail-loop* to iterate through every solution available of predicates *university/1, department/2* and *student/2*. The second, uses a *repeat-loop* to repeat the same actions while both the username and the respective password inputted are not true. In the case of the *fail-loop*, the code fragments that originate loops are nested loops if there are any preceding code fragments that also originate loops (i.e., the iteration through the departments is a loop nested inside the iteration through the universities). While in the case of *the repeat-loop* the code fragments that originate loops are nested loops if there are any succeeding code fragments that also originate loops (i.e., the loop occurring while the username is not valid is nested inside the loop occurring while the password is not valid). As such, for the *fail-loop*, the number of nested loops is equal to the number of Prolog computation fragments that can have alternative solutions (and subsequently generate different results when backtracked into) minus one. Similarly, in the *repeat-loop*, the number of nested loops is equal to the number of Prolog computation fragments that can fail (and subsequently cause repetition through backtracking) minus one. In this project, we define both the *fail-loop* and the *repeat-loop* as the overarching looping patterns (the occurrence of iteration/repetition equivalent to a loop) regardless of the number of nested loops.

To describe *fail-loops*, we chose the *for-loop* (commonly used in imperative languages) as its formal equivalent. *Fail-loops* are used to iterate through various solutions that satisfy a given predicate,

this logic is similar to the “for each x in set” entrenched in *for-loops*. We could have used other constructs, such as the *while-loop*, but ultimately it came down to preference and what we considered was more intuitive for the reader. Our formal language includes several syntactic constructs used to specify programming control flow. In the case of the *for-loop*, we used it in two distinct ways. One of them specifies an iteration over all elements of a list. The other specifies an iteration over all solutions of a certain condition, that is, the values of the free variables for which the condition is true. In here, we use this later construct:

```
for( Term : Condition ){
       Body
}
```

This construct specifies a repetitive process in which, for each value of *Term* that satisfies the specified *Condition*, repeats all instructions contained in *Body*. The pragmatics of the colon operator (:) was borrowed from the way a set may be specified by comprehension, for instance {x : Children(Eve, x)} which is the set of x that satisfy the condition of being children of Eve.

Taking the previously mentioned *displayStudents/3*, its resulting description in our formal language should look like the following:

```
displayStudents(){
     for(U:university(U)){
          for(D:department(U,D)){
               for(S:student(D,S)){
...
```

Each subsequent loop is nested in the former, and the construct used for each loop is in the format “`for(Arguments:Predicate)`”.

Regarding the *repeat-loop*, it is also a loop that use uses failure to cause repetition. The *repeat/0* clause is followed by code fragments that can fail. When these fragments fail and the program backtracks, *repeat/0* always generates alternative solutions, making it a loop. To describe this pattern, we chose a *do-while* construct equivalent. Originally, we thought about using a normal *while-loop* construct, but the resulting description was not accurate. The examples below, illustrate this:

```
while(not(username(Name))){
       ...
}
```

```
do{
       ...
} while(not(username(Name)))
```

While at first glance it may seem that both constructs are equivalent, there is a slight nuance that distinguishes them. The former, first checks the condition and then executes the loop body, repeating this process as many times as the checked condition is true. Whereas the latter, first executes the loop body and then checks the condition, repeating it again as many times as the condition is true. The do-while construct is a more accurate representation of the Prolog code because even if every single predicate is called successfully the program still runs once. The same happens in a *repeat-loop*, even if no predicate fails, the program still runs as intended. When using a *while-loop*, if the condition checks as false then the loop body is not executed. Comparatively, when using a *do-while* loop, the loop body is executed at least once regardless of its condition.

Using the previous *inputCredentials/0*, its expected description should look like the following:

```
inputCredentials(){
     do{
          do{
               ...
          } while(not(name(Name)))
          ...
     } while(not(password(Name,Password)))
     ...
}
```

Above, we can see the use of the *do-while* construct. The original procedure had two predicates that could fail, *name/1* and *password/2*, as such, there are two loops in the pattern. The addition of the negation "`not`" is used do depict that the loop will keep repeating until the condition (in this case, the name and/or password being valid) is true. If a procedure was identified as being able to fail instead of a fact, we would use "`not_successful`" instead "`not`" as a way to negate it. A procedure is not true or false, instead it can either be successful or fail.

For the formal description of the *list-iteration-procedure* we also use the *for-loop*. As already mentioned, we used the *for-loop* in our formal language for two different types of iterations. Iterating through all the solutions of a certain condition (i.e., *fail-loops*) and for iterating through all elements of a list. The *list-iteration-procedure* is a pattern in which there is a recursive iteration through a list with the purpose of doing actions. The procedure below exemplifies this pattern:

```
printList([X|Rest]):-
    write(X), nl,
    printList(Rest).
printList([]).
```

The procedure *printList/1* goes through every element of the receiving list, printing them on the console. We can observe it is recursive seeing as it calls itself. For this pattern, we borrowed the syntax from the Python programming language, using the "in" keyword. Besides making evident the different uses of the *for* construct, the use of "`for(X in List)`" makes it easy for the reader to understand the underlying iteration, as the syntax itself is already close to natural language. As such, the resulting formal description should look like the following:

```
printList(List){
     for(X in List){
          ...
     }
}
```

The *list-iteration-procedure* has a small detail however, that can greatly impact the resulting descriptions if one isn't careful. If a procedure that follows this pattern has more than one recursive clause iterating through the list, those clauses do not represent different iterations. All recursive clauses in a procedure that is a *list-iteration-procedure* simply state different actions to be done upon different conditions, the underlying list iteration is still the same. We can exemplify this with the following procedure:

```
biggerThan(N,[X|Rest]):-
    X>N,
    write(X),write(' is bigger than '),write(N),nl,
    biggerThan(N,Rest).
biggerThan(N,[X|Rest]):-
    X=<N,
    write(X),write(' is not bigger than '),write(N),nl,
    biggerThan(N,Rest).
biggerThan(_,[]).
```

The procedure *biggerThan/2* has three clauses. Two recursive, list-iterating clauses and one terminating clause (to stop the recursion). The first two clauses do not depict two different iterations, they both represent the same iteration. When one fails, it backtracks, fails and goes to the other one to continue the iteration.

The final pattern we identified is the *if-clause* pattern. While obvious to Prolog experts, it may be quite a surprise to newcomers the way Prolog programs implement the equivalent to *if-clause* constructs of imperative programming languages. In declarative Prolog, every single predicate call can be looked at as a condition. The "control-flow" of the program only proceeds if every predicate being called is true, otherwise it backtracks. For the sake of simplicity, as well as general comprehensibility of the resulting descriptions, we chose to purposefully limit what we considered an *if-clause.* (Our treatment of *if-clauses* would have to be greatly improved upon if we also wanted to tackle declarative Prolog.) Using the program above as an example, we can identify two programming patterns, *list-iteration-procedure* and the *if-clause* pattern. In this particular case, the conditions that represent *if-clauses* are opposite conditions, either X is greater than N or X is equal to or lesser than N. Seeing as they are opposite conditions and we wanted to make the descriptions easier and more intuitive for imperative programmers, we opted to use the syntax *else*, instead of the regular *if*. As such, the resulting description should look like the following:

```
biggerThan(N,List){
     for(X in List){
          if(X>N){
               ...
          }
          else{
               ...
          }
     }
}
```

Due to the scope and limited time of this project, we could not describe more programming patterns. Initially, we started by tackling the main procedural patterns of repetition through failure, the *fail-loop* and the *repeat-loop*. Seeing as we considered these were not enough (and to further prove that our approach was generalizable) we also added the *list-iteration-procedure* and the *if-clause.* The latter was added because of the importance recursion has in Prolog programming and the former

because of its pedagogical value for newcomers. Below are all the patterns identified, as well as, their equivalent formal descriptions:

Table 8 - Formal description templates

Pattern	Formal Description	Notes
fail-loop	`for(Arguments:Predicate)`	As many loops as predicates with more than one solution

repeat-loop	`do`
	`...`
	`while(negation(Condition)`
		As many loops as predicates that can fail

list-iteration-procedure	`for(Element in List)`	All recursive clause bodies are nested in the same loop
if-clause	`if(Condition)`	Opposite conditions are treated as an *if-else* pair

4.2. Discarded Approaches

In any research and/or development work, many iterations and attempts may have to be made until a satisfactory approach or conclusion is reached. Solutions may often be improved through iterated analysis, discussion and feedback. This section contains the most relevant issues we grappled with that suffered considerable changes throughout the course of this project. These various discarded approaches vary in scope and significance, as such, some will be more detailed than others. This explanation of past mistakes and less efficient methods serves the purpose of justifying decisions adopted in the final approach and also to highlight some work done that would otherwise never be mentioned.

The five phases of the general approach (i.e., Target Program Access, Pattern Tagging, Pattern Transformation, Formatting and Rendering) have clear goals but many possible ways of going about them. We chose to highlight only three scrapped approaches as these are the most significant in terms of work involved and/or discussion generated. The first relates to how much information we used to obtain in Target Program Access. The second, also belongs to the Target Program Access phase and was made deep into the project. We used to use our own generated names for the variables of the Prolog programs, but this created a disconnect between the descriptions and the original program. We ended up reading the programs directly from a file to keep the original variable names. Despite altering the code substantially, it also improved our results. Finally, the third change pertains to how we detect *fail-loops* in the Pattern Tagging phase.

4.2.1. Information Extraction

In order to generate descriptions from Prolog programs, the very first step is accessing their code and extracting information about them. During the exploratory phase, we looked at a lot of information to better familiarize ourselves with these patterns and the optimal way of processing them. Below is an example of a procedure and the respective information we used to extract.

```
status(mike,online).
status(john,offline).
status(barbara,online.

%Fail-loop
displayStatus:-
    status(Person,Status),
    write(Person),write(' is '),write(Status),nl,
    fail.
displayStatus.
```

Extracted information:

```
predicate:displayStatus/0 %Name of the predicate being analysed
clauses:2 %Number of clauses
rules:1 %Number of rules
fail:1 %Number of fails
failTasks:[status/2] %Predicates with more than one solution
repeat:0 %Number of repeats
repeatTasks:n/a %Predicates that can fail and originate a loop with repeat/0
recursion:0 %Existence of recursion
predicateList:[displayStatus/0,status/2,write/1,nl/0] %Predicates of the predicate
rule(at the time only one rule was being analysed)
totalPredicates:4 %Length of predicateList
totalBranches:3 %Total number of clauses of all non-built-in predicates belonging
to the main predicate being analysed (in this case, status/2 has 3 clauses and is
the only non-built-in predicate called by displayStatus/0)
```

Through the careful analysis of information of various example programs, we settled which characteristics of the program were truly important and which were not relevant. The current approach, comparatively to this first iteration, still looks at a lot of the same characteristics but with noticeable differences. The current software has the capacity to access all predicate rules and not just one (more on that shortly). The program information is no longer accessed and displayed in table report fashion. Instead, the code patterns are detected then tagged and later transformed into formal constructs. Useless metrics such as number of fails and total number of predicates were abandoned after being used in early description drafts and considered not useful for generating descriptions.

The information analysed was trimmed down and we started generating our first formal descriptions. In the early stages, we used specific predicates for each pattern for both detection and processing. This rigid approach made it more difficult to change the descriptions and add more patterns. Additionally, the test programs we were using were very similar to one another. This was very useful so that we could slowly increase the complexity of the software, but made us oblivious to important matters, such as the existence of programs with more than one clause and/or pattern. Fundamentally, we decided to change the way we were analysing and processing patterns as we wanted to guarantee that the software was easily maintainable and improvable. The final adopted solution allows for new patterns and descriptions to be easily added/changed and can process predicates with more than one clause and/or programming pattern.

Initial exploration also included trying to detect all types of recursion, but the endeavour was dropped. While we process a specific code pattern using recursion, the full processing of recursive programs, such as indirectly recursive ones (e.g., a calls b, b calls c, c calls a), is out of the scope of the current work. Despite its conceptual importance and practical use in Prolog, we decided to focus essentially on the procedural component of the Prolog language that relies on backtracking and not on recursion.

4.2.2. Handling Variables

Major changes are sometimes done late into a research and/or development endeavour when the end-result is clear and does not match the expectations. One of the main premises of our approach is that both descriptions, formal and natural, are equivalent to and explanatory of the Prolog program. Both generated descriptions should hold up individually and have synergistic value when used together. During the Program Access stage, we accessed the clauses of a given target Prolog procedure, using the predicate *clause/2*. Using it together with a *fail-loop* for example, allowed us to obtain all the clauses of a procedure. This approach had a main problem though. The non-instantiated variables of the target Prolog program, when accessed with *clause/2* have meaningless names automatically generated by the Prolog system. As an example, consider the following Prolog program and the result of the *clause/2* predicate shown afterwards, in the Prolog command line.

```
displayStatus:-
    status(Person, Status),
    write(Person),
    write(' is '),
    write(Status),
    nl,
    fail.
displayStatus.


?- clause(displayStatus, Body).
Body = status(_3542, _3544), write(_3542), write(' is '), write(_3544), nl,
fail
True
```

The original variables, Person and Status ware automatically named *_3542* and *_3544*. These internal variable names are not only visually unappealing but also incomprehensible if we were to use them in a description. Our first approach consisted of translating these variables into more usual variable names. Instead of the abstruse names Prolog uses for variables, we wanted to use variable names more that were more easily recognizable. For that, we created a variable name generator capable of generating usual variable names as x, y, z, x1, x2… In addition, we created a mapping between the Prolog variable names and our hopefully user-friendly names. This approach was effective, and the translated variables carried on through both the formal description and the natural language one.

Prolog program:

```
displayStatus:-
    status(Person, Status),
    write(Person),
    write(' is '),
    write(Status),
    nl,
    fail.
displayStatus.
```

Formal description:

```
displayStatus(){
     for((x,y):status(x,y)){
          print_line(x," is ",y)
     }
}
```

Natural language description:

For each x and y that satisfy status(x,y), displayStatus prints "x is y" on the console and breaks line.

There is however, a clear disconnect between the Prolog program and its descriptions. Since the variables have been renamed and do not match the original program, the descriptions, especially the natural language one, do not exactly match the original program. The variable translation brings two major flaws. Firstly, it makes it unclear if the descriptions do actually represent the corresponding program. Secondly, the original variables names can sometimes have meaning (e.g., Person, Status) which helps the reader to understand the code and when they are renamed that meaning is lost. This reflection made it imperative that we change the way to access the clauses of the Prolog programs to be described. In our current approach, the variable names that the programmer chose as appropriate are preserved and used in both formal and natural language descriptions. This improvement was achieved by accessing the programs directly from the file.

4.2.3. *Fail-Loop* Identification

The *fail-loop* can be computationally described as the use of failure to exhaust all possible solutions of a certain computation fragment with the help of backtracking. Even though it is simple to identify the existence of a *fail-loop*, there are still some harder details to handled.

```
printChildren:-
      parent(P),
      child(P,C),
      write(C),
      write(‘ is the child of  ’),
      write(P),nl,
      fail.
printChildren.
```

```
printAges:-
      person(P),
      age(P,A),
      write(P),
      write(‘ is ’),
      write(A),nl,
      fail.
printAges.
```

Taking into consideration their structure, both programs presented above are undoubtedly *fail-loops*. The troublesome part is now finding out how many loops are embedded in each program. The first program iterates through each parent and then iterates through each parent’s child. The second program just iterates through each person. It does not iterate through “each age” because there is no such thing as different ages for the same person. A person cannot have multiple ages, a person only has his/her age. This is the main conundrum of detecting *fail-loops*. Both programs have identical

structure, however a human programmer can easily conclude that *printChildren/0* has 2 loops (one of them embedded within the other) while *printAges/0* only has one. The ability of a person to distinguish between the two cases stems from real-life knowledge and context that goes far beyond the boundaries of the code. The problem becomes even more difficult if we have programs like the following one:

```
a:-
      b(X),
      c(X, Y),
      d(X, Y),
      fail.
a.
```

It is impossible to accurately deduce if this program implements a single, a double, or even a triple loop because the chosen names of the predicates *a/0, b/1, c/2* and *d/2* do not carry any information we can match against our *a priori* knowledge about the world. Both humans and machines can only make assumptions. This led us to define one of the first problems when generating formal descriptions: how to detect if a predicate can have multiple solutions.

Out first approach consisted of using the predicate *call_nth/2* from the SWI Prolog library *solution_sequences*. The predicate *call_nth/2* has 2 arguments, Goal and N. According to the SWI documentation, it is "True when Goal succeeded for the Nth time. If Nth is bound on entry, the predicate succeeds deterministically if there are at least Nth solutions for Goal" [21]. It seemed like a perfect tool to solve our problem. Despite its apparent appeal, we quickly realized *call_nth/2* was not fit to tackle this type of problem.

```
?-call_nth(person(_),2).
true.
?-call_nth(age(john,_),2).
false.
?-call_nth(age(_,_),2).
true.
?-call_nth(member(_,_),2).
true.
```

The console outputs above show us one of the inadequacies of *call_nth/2* to this type of problem. A predicate has at least N solutions if it succeeds N or more times when called. Calling predicates with non-instantiated variables though will most likely always succeed more than once. This is due to the fact that predicates like *age/2* have effectively more than one solution in the knowledge base (they only have one, when the variable Person is instantiated). The case is even worse with recursive definitions like *member/2*. Due to the relational nature of Prolog, *member/2* can be used to check if something exists in a list (if you instantiate both variables) or to get, through backtracking, all members of a list, if only the list parameter is instantiated. When calling *member/2* with both variables not instantiated it will have infinite solutions.

The use of *call_nth/2* could also create undesired side-effects which would be quite difficult to handle. By using *call_nth/2* we are calling predicates to count the number of times they succeed. When using it with input and output predicates like *write/1* and *read/1* for example, we can clutter the console with unsought outputs or even worse, bring the program to a halt. Other damaging side-effects could

also include the calling of predicates like *assert/1* and *retract/1* which could add/remove important facts to/from our knowledge base.

As a result of these interactions between *call_nth/2* and predicates with non-instantiated variables (which is mainly what we will be faced with when analysing code), as well as, the troublesome side-effects brought by its use, we discarded the use of this predicate.

We concluded that it was not possible to know for sure if a predicate has (or could have) more than one solution just by analysing the code without any additional context. Subsequent approaches involved using assumptions about predicates that most likely have more than one solution (e.g., *member/2*). The use of educated guesses and common programming practices does not ensure the correct classification of all predicates (e.g., facts usually have more than one solution, but as we saw, *age/2* only has one). Hence, we had to compromise by using the mentioned assumptions and, at the same time, allowing the user to explicitly state, by means of a simple ontology, if a certain predicate can have more than one solution or not.

4.3. Adopted Approach

The approach adopted includes a set of formal description templates that are used by our program to generate the formal descriptions of the target programs. These templates, defined by us, are formal equivalents to the various code patterns identified in procedural Prolog. The used description templates do not correspond to unique formal descriptions. It is possible to describe the same Prolog fragment in alternative ways. However, in this project, we decided to propose a single description for each identified Prolog code pattern. The proposed description templates reflect our own encoding preferences (e.g., using a *do-while* instead of *repeat-until* construct as the equivalent of a *repeat-loop* code pattern, or a *for-loop* instead of a *while-loop* as the equivalent of a *fail-loop* pattern).

The entirety of the software for generating Formal descriptions was made in Prolog (SWI Prolog). The use of Prolog as the programming language for this task was chosen for two main reasons. Firstly, by using the same programming language to describe the programs we are analysing, we make the solution simpler. The use of other programming languages could add unnecessary complexity and bring little to no benefit in return. Secondly, the investigation work of our group focuses on the use of symbolic processing to solve problems. These two reasons lead us to choose Prolog, which will also facilitate the integration of this work in other work of our research group.

The next subsections describe in more detail the five stages comprising the formal description generation process: target process access, pattern tagging, pattern transformation, formatting and rendering.

4.3.1. Target Program Access

In order to describe a program, we first have to access it. Accessing a program consists of acquiring its header and the clauses of its definition for subsequent processing. As a simple example, consider the following program:

```
printList([X|Rest]):-
      write(X),nl,
      printList(Rest).
printList([]).
```

The header of the program is printList(L). Its two clauses are (i) printList([X|Rest]):- write(X), nl, printList(Rest), and (ii) printList([]):- true. Instead of the full clauses, we are interested only in their bodies: (i) write(X), nl, printList(Rest), and (ii) true. We are also only interested in bodies of rules: write(X), nl, printList(Rest). In the procedural programs we are focused on, the facts are usually just terminating clauses for the loops. Regardless, these clauses hold little value for the generation of descriptions.

The solution could have easily received a header and respective clause bodies as input and produce a description as output. However, that would not be a convenient solution as it would be too troublesome to use with big programs. Ideally, we want a solution in which we can simply input a procedure name and its respective arity (seeing as different programs can have the same name).

```
?-prolog_to_imperative(printList/1).
```

Therefore, we need a way to access a program's header and clause bodies just from its name and arity. We could use predicates like *clause/2* to extract a procedure's clauses. But as we displayed in section 4.2, using this method leaves us with incomprehensible variable names, which would lead to descriptions that do not exactly parallel the original program. Alternatively, the only way to maintain the original variable names is to read the programs directly from a file. By using *read_term/3* to read terms directly from a file we can specify reading options in its third argument. One of the available options is *variable_names* which provides us with a list containing all the variables and their respective original names, for example:

```
?-open('Test_Programs.pl',read,Stream),
read_term(Stream,Term,[variable_names(Names)]).

Stream = <stream>(000002D5F828AC30),
Term =  (displayStatus:-status(_5644, _5646), write(_5644), write(' is '),
write(_5646), nl, fail),
Names = ['Person'=_5644, 'Status'=_5646].
```

Once read, we process every term of each clause using the variable names to replace their variables with their original names (in atom format). If a variable has no translation available in the variable names list, then it is an anonymous variable and so we swap it with an underscore (also, in atom format). We repeat this process for every clause and end up with a program's body ready for further processing.

4.3.2. Pattern Tagging

Our approach to pattern detection suffered some changes throughout the project. The current approach facilitates future work and addition of future patterns. We detect patterns by iterating through the clauses of the target program. The various code fragments that constitute a programming pattern are then identified by using specially designated tags. Tags are identifiers used to transition between the programming patterns and the equivalent formal constructs. Each programming pattern uses one or more tags that are later transformed into the proper formal constructs. Future patterns can also use these tags if the equivalent formal constructs are the same. Here are all tags used, as well as, their equivalent programming patterns:

Table 9 - Tags and respective patterns

Tag	Pattern	Notes
do_while	*repeat-loop*	Identifies repeat/0, subsequent predicates that can fail are tagged with the can_fail tag
can_fail(Predicate)	*repeat-loop*	The predicate can fail and is inside a *repeat-loop*
for_loop(Args,Predicate)	*fail-loop*	Args belong to Predicate which can have more than one solution. This tagging only occurs inside a clause that ends with *fail/0*
iter_loop(Arg)	*list-iteration-procedure*	Identifies iteration of whole list through recursion. Arg is head variable used to iterate through the list. Tag is placed at the beginning of the clause body to nest the rest of the code
if_clause(Predicate)	*if-clause*	Predicate is considered equivalent to an *if-clause*

In the software, tags are identified by the keyword “tag”, followed by a colon “:” and one of the respective names above (e.g., tag:do_while). These tags are used to facilitate the posterior transformation of the programming patterns into the appropriate formal constructs using the aforementioned description templates.

The *fail-loop* pattern is identified as evoking at least one predicate that can have more than one solution in a clause body that ends with *fail/0*. In the present case, we try to define the most general syntactic criteria for determining if a predicate has or could have several solutions, in spite of being aware of the insufficiency of this approach. We assume that predicates defined as sets of facts, the predicate *member/2* and any predicate that contains at least one of the former are predicates that can have more than one solution. Given that this criteria is obviously not completely accurate (e.g., the previously seen *age/2* in section 4.2.3 does not have more than one solution despite being a fact), we allow the user to explicitly specify predicates that have or do not have more than one solution.

```
predicate_has_more_solutions(a(_)).
%a/1 will now be tagged as having more than one solution
predicate_does_not_have_more_solutions(age(_,_)).
%age/2 will not be tagged despite being a fact
```

The tag “for_loop” indicates that the predicates that can have more than one solution (in a clause that ends with *fail/0,* constituting a *fail-loop*) will be transformed into a *for-loop* (see section 4.1). The tag identifies the arguments and the predicate “for_loop(Args,Predicate)” to further facilitate the subsequent formal transformation into “for(Args:Predicate)”. Below is an example of a *fail-loop*, and the tagged clause body:

```
displayStatus :-
    status(Person, Status),
    write(Person),
    write(' is '),
    write(Status),
    nl,
    fail.
displayStatus.
```

Tagged clause body:

```
tag:for_loop([Person,Status], status(Person,Status)), write(Person),
write(' is '), write(Status), nl
```

We identify the *repeat-loop* as any clause body that has a *repeat/0* and subsequent predicates that can fail. The repeat/0 is swapped with a “do_while” tag indicating the beginning of a *do-while* loop and the subsequent predicates that can fail are tagged with a “can_fail” tag. Further down the line, in Pattern Transformation these tags will be used to create a proper *do-while* loop. Identifying which predicates can fail can be a challenging task though. A procedure like *write/1* for example cannot fail. All it does is output to the current stream whatever its argument is. A non-declarative procedure like *write/1* cannot fail and the same applies for other non-declarative procedures existing in Prolog. Once again, we try to create general criteria that identifies built-in and user-defined predicates and procedures that can fail. We started with a (yet to be completed) list of built-in predicates that can fail. Then, we use the general rule that a predicate or a procedure can fail if its definition contains another predicate that can fail. These criteria are not bulletproof but contain useful and very common cases. Strengthening these rules by adding more cases will be very useful in improving the quality of the program. Below is an example of a repeat-loop and its tagged clause body:

```
inputPassword:-
    repeat,
    write('Insert password:'),
    nl,
    read(Password),
    processPassword(Password).
```

Tagged clause body:

```
tag:do_while, write('Insert password:'), nl, read(Password), tag:canfail(
processPassword(Password))
```

As before, due to the possible wrong assumptions done by this automation, we allow the user to specifically state if a predicate can fail or not:

```
predicate_cannot_fail(z(X)).
%z/1 will not be tagged as being able to fail
predicate_can_fail(p(X)).
%p/1 will be tagged as being able to fail
```

The *list-iteration-procedure* is a Prolog programming pattern where a procedure calls itself in order to iterate through a list and performs actions throughout the iteration. The corresponding tag “iter_loop” is used when a procedure is calling itself (same name, same arity) in its clause body with the tail of the originally received list. Additionally, we decided to consider only the cases in which the defined program iterates through all elements of the list (not just some of them). After confirming that the program is calling itself and that the variable passing is as expected, we check for the existence of a terminating clause with an empty list. The following procedure exemplifies this pattern:

```
printList([X|Rest]):-
      write(X),nl,
      printList(Rest).
printList([]).
```

The recursive call uses the tail (Rest) of the originally received list ([X|Rest]), and the procedure has an empty list terminating clause, meaning it iterates through the whole list. Throughout the iteration it performs actions, in this case, writing on the console. Thus, *printList/1* is properly tagged as a list iteration with the tag "iter_loop" and the respective iterating variable (i.e., X). The tag is placed at the beginning of the tagged clause body to indicate that the rest of the clause body happens inside the iteration loop. Below is the tagged clause body of *printList/1*:

```
tag:iter_loop(X), write(X), nl
```

Lastly, the final tag denotes the existence of *if-clauses*. Currently we are only tagging "math" predicates (e.g., *==/2*, *>/2*). This choice was done for the sake of simplicity, as well as general comprehensibility of the resulting descriptions. This final pattern was added only to enrich the formal descriptions. If we wanted to be truly rigorous than a lot of predicates would have to be tagged as *if-clauses*, seeing as in Prolog, the control-flow of the program only proceeds if a predicate is true (otherwise it backtracks). In the future, if we want to describe declarative Prolog, this is a pattern whose tagging method will be changed considerably. Below is an example of a very simple Prolog procedure and its respective tagged clause body:

```
isBigger(X,Y):-
    X>Y,
    write('The first argument is bigger').
```

Tagged clause body:

```
tag:if_clause(X>Y), write('The first argument is bigger')
```

The mathematical predicate *>/2* is tagged as an *if-clause* with the "if_clause" tag. After all programming patterns are properly tagged they move on the next stage, Pattern Transformation.

4.3.3. Pattern Transformation

Pattern transformation consists in taking the previously tagged code fragments and transforming them into the equivalent formal constructs. The transformation is done using the templates and formal language previously defined (see section 4.1). Before the transformation some pre-processing is done if needed. Next, is an example of a *fail-loop* with two nested loops:

```
displayStudents:-
    university(U),
    department(U,D),
    student(D,S),
    write(S),nl,
    fail.
displayStudents.
```

Before transforming the tags and the original code to which they are applied into formal constructs, *for-loop* arguments are processed to maintain correction. When identifying the *fail-loop* pattern, both the arguments and the predicate are tagged for posterior transformation. The arguments and predicate are then transformed into the proper construct "`for(Arguments:Predicate)`". This simplistic approach can create some incorrection in the descriptions though. Using the procedure above, *displayStudents/0* as an example, if we don't do any pre-processing the resulting description could look like this:

```
for(U:university(U)){
      for((U,D):department(U,D)){
            for((D,S):student(D,S)){
. . .
```

The description above is not correct as it shows both nested loops iterating through two variables when in fact, they only iterate through one. In *displayStudents/0* the first loop iterates through every university, the second through every department belonging to that university and the third through every student belonging to that department. As such, the variables in the *for-loop* tags are filtered so that nested loops do not have variables belonging to outer loops. After this filtering, the tags with the now correct arguments and predicate are simply transformed into the equivalent "`for(Arguments:Predicate)`". The resulting transformed clause body will look like this:

```
for(U:university(U)), for(D:department(U,D)), for(S:student(D,S)),
write(S), nl
```

Regarding the *repeat-loop* pattern, its formal equivalent *do-while* two components, the "do" and the "while". The do encapsulates the loop body and the while states the looping condition. If there is more than one loop, we have to make sure they are nested in the resulting description. The first step is counting the number of existing loops. The number of loops is equal to the number of Prolog computation fragments that can fail (and consequently cause repetition through backtracking). In this description, a computation fragment can be either a predicate evocation or a procedure evocation. So, we count the number of existing tags that denote failure possibility (*can_fail* tag) and add the respective amount of "do" constructs to nest the subsequent code. Afterwards, we process every single computation fragment that can fail by adding the condition component of the *do-while* construct for each one. When the computation fragment that can fail is a predicate evocation (P), we add a *while* constraint negating P: "`while(not(P))`". When the computation fragment that can fail is a procedure evocation (P), it makes no sense speaking about it being true or false. Instead, a procedure call can either be successful or fail. In such circumstances, the *while* construct applies the "`not_successful`" operator to the procedure call: "`while(not_successful(P))`". Below is an example of a *repeat-loop* and what it will look like once its patterns are transformed:

```
inputPassword :-
    repeat,
    write('Insert password:'),
    nl,
    read(Password),
    processPassword(Password).
```

Transformed clause body:

```
do, write('Insert password:'), nl, read(Password),
while(not_successful(processPassword(Password)))
```

The *list-iteration-procedure,* just like the *fail-loop,* may also require pre-processing before the conversion into formal constructs. If a *list-iteration-procedure* has more than one recursive clause then all of them are tagged as iteration loops. This can create wrong descriptions as we have already seen that all recursive clauses belong to the same iteration. Below is an example of a *list-iteration-procedure* with more than one clause:

```
biggerThan(N,[X|Rest]):-
    X>N,
    write(X),write(' is bigger than '),write(N),nl,
    biggerThan(N,Rest).
biggerThan(N,[X|Rest]):-
    X=<N,
    write(X),write(' is not bigger than '),write(N),nl,
    biggerThan(N,Rest).
biggerThan(_,[]).
```

Tagged clause bodies:

```
tag:iter_loop(X), tag:if_clause(X>N), write(X), write(' is bigger than '),
write(N), nl
```

```
tag:iter_loop(X), tag:if_clause(X=<N), write(X), write(' is not bigger than
'), write(N), nl
```

The above program, *biggerThan/2*, has two clauses and both were rightly tagged as list iterations (seeing as they both correspond to the *list-iteration-procedure* programming pattern). Yet, if we do not pre-process the tagged clause bodies, the resulting description would have the following format:

```
biggerThan(…){
     for(…){
          …
     }
     for(…){
          …
     }
}
```

The above description is wrong. In the original Prolog predicate, despite it having two different recursive clauses, there is only one iteration being made. If the condition in the first clause fails, it backtracks, fails, and proceeds to the next clause, but the underlying list iteration is the same. However, the description that would be generated would specify two loops, occurring one after the other, which is inaccurate. As such, when processing a *list-iteration-procedure* with more than one recursive clause we need to pre-process it before generating formal constructs. The pre-processing involves creating a single clause body with every clause body that was tagged as a list iteration and tagging it as the only *list-iteration-procedure.* Afterwards, the transformation is done as usual, using a preset template, in this case, transforming “tag:iter_loop(X)” into “`for(X in List)`”. In the transformation process we create

a mock variable called “List” which is equivalent to the list being iterated (i.e., [X|Rest]). This was done so that the resulting description would be easier to understand as it is closer to imperative languages. In order to maintain coherence in the description, the iterated list is renamed “List” in the whole description. This obviously includes the description header, so instead of “`biggerThan(N,[X|Rest])`” we will have “`biggerThan(N,List)`” in the final description.

The procedure *biggerThan/2,* has more than one programming pattern though. In both clauses the value of N is tested against X. Both mathematical predicates (i.e., *>/2* and *=</2*) are tagged as *if-clauses*. In this particular case, to improve the comprehensibility of the resulting description, instead of transforming the tagged predicates into two *if-clauses,* they will be transformed into an *if-else* pair. This transformation only occurs when the two conditions are exact opposites. If the conditions were for example, X>N and X<N they would be transformed into two *if-clauses* and not into an *if-else*.

Below is the transformed clause body of *biggerThan/2*:

```
for(X in List), if(X>N), write(X), write(' is bigger than '), write(N), nl,
else, write(X), write(' is not bigger than '), write(N), nl
```

After transforming all the patterns in the clause bodies into their imperative equivalent constructs, we move on to Formatting.

4.3.4. Formatting

In a formal description similar to imperative code, the content is very important but proper formatting is also essential to comprehensibility. In this final step before Rendering, we handle indentation as well as the transformation of some console output actions (i.e., *write/1, nl/0*) and predicates/procedures with zero arity (e.g., *displayStatus/0*), to ensure that the formal description is as understandable as possible.

The indentation of the description is also handled with tags. The software tags every part of the clause body to indent with “tag:indent” and every part to reduce indentation with “tag:unindent”. Subsequently in Rendering, whenever the program finds an indentation tag it will print every following element of the clause body with an extra paragraph of space. Inversely, whenever it finds the unindent tag, it will reduce a paragraph of space for every following element. The indentation tags are used next to *for-loops*, the *do* component of the *do-while* and *if-clauses/else-clauses*. The unindent tag is used to align the *while* component of the *do-while* construct with its respective *do,* as well as, helping with the formatting of procedures with many clauses.

Besides handling indentation, we also considered it was useful to translate the console output actions *write/1* and *nl/0*. While one could easily argue that *write/1* is easily understood by the average programmer, *nl/0* is definitely not as self-evident. Furthermore, we want to make the descriptions as familiar-looking as possible for imperative programmers. So, instead of using *write/1* and *nl/0* we transform them into either “`print`” or “`print_line`” as they are names that imperative programmers

will be much more familiar with. For the sake of simplicity, as well as making the description more visually appealing, we group consecutive actions. If we have two or more consecutive *write/1*, then the resulting print is unified for the formal description (e.g., `write(A), write(B)` is converted into `print(A,B)`). If we have some consecutive calls to *write/1* followed by *nl/0* then the resulting syntax should be "print line" (e.g., `write(A), write(B), nl` is converted into `print_line(A,B)`). Besides grouping together these actions, we also take into consideration their arguments. When printing an argument, if that argument is not a variable then it should be encompassed by quotation marks (e.g., `write('my name is '),write(Name)` is converted into `print("my name is ",A)`).

Below is the simple procedure *printList/1,* and its clause body after the respective stages.

```
printList([X|Rest]) :-
    write(X),
    nl,
    printList(Rest).
printList([]).
```

Clause body:

```
write(X), nl, printList(Rest)
```

Clause body after tagging:

```
tag:iter_loop(X),write(X), nl
```

Tagged clause body after pattern transformation:

```
for(X in List), write(X), nl
```

Transformed clause body after formatting:

```
for(X in List), tag:indent, print_line(X)
```

Finally, in this step we also transform predicates/procedures with zero arity by adding parentheses "()" to them. In Prolog, calling a predicate/procedure with no arguments uses just its name (e.g., displayStatus.). We add parentheses to these calls so that they are closer to what an imperative programmer would expect to see in a program (e.g., `displayStatus` -> `displayStatus()`).

After formatting, the results are rendered into readable formal descriptions.

4.3.5. Rendering

After everything is duly processed, and we can finally generate the formal description. We have two styles for description generation, java style and python style (both inspired by the respective programming languages). Both styles process indentation exactly the same way, using the proper tags to indent and unindent the text when needed. The only difference between both styles is that the python style uses a colon ":" to separate some components of syntactic constructs while java style uses braces "{ }". Despite the processing being very similar, descriptions printed in java style have the added

challenge of closing the braces that were opened and making sure that those closing braces are also properly indented. Here is the formal description of *printList/1* in both java style and python style:

```
printList([X|Rest]) :-
    write(X),
    nl,
    printList(Rest).
printList([]).
```

Formal description in java style and python style respectively:

```
printList(List){
     for(X in List){
          print_line(X)
     }
}
```

```
printList(List):
     for(X in List):
          print_line(X)
```

The generated formal descriptions can easily be displayed in the Prolog console or sent to a file. If we want them to be further processed by a Prolog program, we have to use their internal encoding instead of their textual renderings. In this project, the generation of a formal description is only an intermediate step for the final goal, namely the generation of the natural language descriptions. This second major step takes a formal description as input and generates the corresponding natural language description. Being implemented by a program, this last major step must receive the internal encodings of the formal descriptions. The next chapter describes the generation of natural language from the formal descriptions discussed in this chapter.

5. Generation of Natural Language Descriptions

The generation of natural language descriptions is the final step in our two-step approach to describe Prolog programs. We generate natural language descriptions from the generated formal descriptions. Just like in the previous step, we also use templates to transform the formal constructs into readable natural language expressions. The use of the formal descriptions instead of the original Prolog programs allows us to generate descriptions from any programming language as long as they are converted into this intermediate formalism. Since the constructs of formal descriptions are very similar to those of imperative programming languages (e.g., Java, Python), the foundations are already half-built so that they could also be described in the future. Similarly, the use of this intermediate formalism, also allows the natural language descriptions to be easily generated in any language (e.g., English, Portuguese) as long as there are corresponding templates for that language. In this project, we only created English templates and as such, can only generate natural descriptions in English. However, the use of the two-step approach allows us to easily add other languages in the future.

In terms of formatting and general understandability, generating a natural language description has different challenges from generating the intermedium formal ones. Before, the main concern was that the code was properly indented so that it would be equivalent to the original program and be understandable to the reader. Now the main concern is the proper use of punctuation (e.g., commas and periods), conjunctions (e.g., “and”) and adverbs (e.g., “then”, “subsequently”). Without punctuation and discourse connectors to smoothly link the various smaller descriptions, the final description will make no sense.

Besides formatting, creating natural language descriptions that are understandable to humans also requires that we have additional concerns with the content itself. We want to make descriptions that are close to what a human would write. In these descriptions, the text should not contain statements which are redundant or vague. By redundant we mean statements in which words are repeated without any need to. For example, if we have the procedure *processPassword/1* and its variable argument is named “Password” (e.g., `processPassword(Password)`), we don’t want to write “processes password Password”. Despite being correct (the procedure processes the password whose variable name is Password), it’s unnecessarily confusing and makes the reading harder. In this case, we prefer to write simply “processes Password”. The meaning is still conveyed, and the reading is much easier. Furthermore, these descriptions are supposed to be explanatory of the original programs and so, we want to, as much as possible, reduce vague statements that may leave the reader feeling lost. For example, if we have the action *read* and the variable name does not offer much information, the resulting excerpt can make the reader think he is missing something (e.g., `read(X)` is translated into “reads X” | What is X? Why is the program reading X?). As such, we want to complement these excerpts by adding them context. We do this by searching the following procedures/actions for additional information that seems suitable (e.g., “reads X” -> “reads password X” | Now the reader knows that X is a password). This

automatic approach is not always possible or ideal, but it is our stance that we should be able to automate and make the program gather information by itself as much as possible and reducing the need of requiring the user/programmer to constantly provide additional information (e.g., descriptions of procedures, context of certain actions).

The generation of natural language descriptions is only comprised of 2 steps: Natural Language Transformation and Text Formatting. The first step consists of transforming the previously generated formal description and its various parts into their natural language equivalent. The final step is formatting the resulting mix of textual excerpts and phrases into a proper readable text using punctuation and discourse connectors such as conjunctions and adverbs.

5.1. Natural Language Transformation

After the formal description is generated, it is further transformed into natural language using templates. This transformation is done by iterating through every element of the formal description (i.e., procedures, actions and formal constructs) and translating them into small natural language descriptions using the proper template. This transformation is done at the most granular level and does not take into consideration the overall understandability of the final text, which is later taken care of in the formatting stage. For programs that have arguments, a small introduction phrase is also added to the beginning of the text stating the name of the program and the arguments it receives (e.g., "`biggerThan(N,List)`" has the corresponding introduction "biggerThan receives N and List").

The transformation of the formal description's elements into natural language concerns itself only with procedures, actions and formal constructs. The indentation tags are ignored as they serve no purpose for the resulting text. Initially, we experimented with preserving the indentation tags and using them to place punctuation and/or discourse connectors, but the results were not satisfying. Seeing as they were not considered very helpful, we discarded their use in this step.

The use of templates to create natural language is useful for three main reasons. The first is that the templates can easily be changed in case there is a need to improve the descriptions (see chapter 7). The second is that it allows us to create descriptions in many different languages. In this project, all the templates were created for the English language, but we could have easily added other languages. The third is that it makes it simple to reutilize code and have the same underlying description generated with different verb conjugations.

Throughout this whole step, the natural language templates are supported by a lexicon containing nouns, verbs, verb conjugations and phrase connecting adverbs. The enriching of the lexicon would greatly improve the descriptions generated and the number of programs it can cover. Future work in improving natural language descriptions would also involve adding more vocabulary to the lexicon.

All the descriptions generated are in the active voice (with the described program being the subject). This choice was made as in initial draft descriptions the ones that used active voice were more understandable to us. Additionally, the descriptions will also be in the present tense as other conjugations did not make as much sense to us. Below is an example of the use of active voice and present tense:

```
p{
    doSomething
}
```

Possible description:

p does something.

5.1.1. Procedures

The first challenge in generating natural language descriptions was to describe regular procedures. Since the formal constructs from formal descriptions are generated by templates they can also easily be described with templates, as their structure and format are always the same. In the case of actions (e.g., *print, read*), there is a finite number of them, and we can also create templates for each one (or as many as is convenient). This does not hold true for procedures though. There is a potentially infinite number of possible procedures and so we can't make specific templates for every single one.

The solution is to use general templates that allow us to describe as many procedures as possible. Originally, we tried using a basic template like writing "executes" or "calls" followed by the procedure name. While it may still be a viable solution for many cases, using it to describe every procedure just generates uninspiring descriptions which read more like a listing and less like a proper text. So, in order to automate the descriptions as much as possible while still making them unique, we took advantage of a certain naming stereotype. If usual coding good practices are used, most procedures are named using a basic structure which contains a verb plus a noun. Examples of this are: `processPassword, printCredentials, calculateResults` etc. If we take into consideration this naming format, it becomes easier to automate descriptions for a large number of procedures. First, we extract the individual words from a procedure's name that is using either camel case (e.g., `processPassword`) or underscore naming convention (e.g., `process_password`). Then, we detect the verb(s) in those words and conjugate it(them) in the present tense. Finally, if there are any arguments, we also add them to the description. For example, "`processPassword(X)`" becomes "processes password X". This approach can also produce very satisfying descriptions for more complex procedures with multiple verbs and names. For example, "`readAndProcessResultsOfGame(X)`" becomes "reads and processes results of game X". This automation is very useful but has very clear weaknesses. The first and most obvious is that it works poorly with any procedure that is named without a verb. For example, "`status(X)`" becomes simply "status X", which may leave the reader even more confused. The second is that it can produce unreadable descriptions from procedures with many arguments. For example, "`processResults(X,Y,Z,W)`" is transformed into "processes results X Y Z W". The third is that it can create unneeded redundancy and confusion when describing procedures whose named variables denote parts of the meaning of the procedure name. For example "`calculateResultsOfGame(Game)`" is translated into "calculate results of game Game". In order to mitigate these flaws, we improved the templates by adding two additional features. The first is to add commas "," and the conjunction "and" to arguments. The "and" conjunction is used to separate the last two arguments and commas are used to separate the rest (obviously none of this is used if the procedure only has one argument). For example, "`sum(X,Y)`" is translated into "sums X and Y" and "`sums(X,Y,Z)`" is translated into "sums X, Y and Z". The second change was the elimination of words from the procedure description if they are equal to argument names. For example, "`calculateResultsOfGame(Game)`" is now translated into "calculate results of Game". These small changes helped considerably in creating better, more readable

descriptions. Despite its weaknesses, this automation practice still helps us to create a lot of satisfying and readable descriptions without needing any type of user input.

Of course, just like in the generation of formal descriptions we allow users to create their own custom description for any procedure if they are not satisfied with the automated ones. All they need to do is add them to the program with *procedure_description/3.* Below is an example:

```
procedure_description(calculateResults(X),present,"sums goals of game").
```

In this case, the resulting formal description for "`calculateResults(X)`" would be "sums goals of game X" instead of the automatically generated "calculates results X".

5.1.2. Actions

The natural language description of actions seemed very straightforward but offered us an unexpected challenge. Most (if not all) actions such as *print* for example, come with additional contextual information (e.g., printing an argument of a procedure, printing a procedure's result, printing the variable of a fact). Actions like *read* on the other hand are not as linear. While *print* always uses a previously used variable, *read* creates a variable that is then used by its following procedure calls. This created a problem since the resulting descriptions would often be vague or not very explanatory. If the action has a self-explanatory variable name then there would be no problem (e.g., "`read(Password)`" is translated into "reads Password"). But often the variables are not conveniently named, and the resulting descriptions can be vague and very confusing for the reader. If we have, for example "`read(X)`" the resulting description "reads X" is not clear because the variable X is not informative of what is being read. The reader will be left wondering "What is X and why is the program reading it?". In order to try and solve this problem in a scalable and automated fashion, we added context to these descriptions. Context is obtained by searching all following procedures and formal constructs for a predicate/procedure that has a variable in common with *read* and a noun in its name. The first noun found is assumed to be the entity that is being denoted by the read variable. For example, if we have "`read(X)`" followed by "`processPassword(X)`" the resulting description for *read* will be "reads password X", in which "password" is the noun appearing in the name of the next procedure. This way, the user will know what X is. Context is not added to the natural language description if it is equal to the variable name though (e.g., "`read(Password), processPassword(Password)`" is translated into "reads Password"). Despite seeming overly simplistic, this approach helped us create more understandable descriptions. Even though it may not always work, we do not have a customizable version for user input as that would be difficult to do, considering the time it would take to create and use a general approach. The user would have to do something similar to creating a fact with the whole formal description and stating its context which would not be acceptable because it would require that the user/programmer would learn a complex description language. Currently we only applied this approach to *read* but the results were better than expected, so it may have other applications in the future.

For the rest of the common actions, such as *print* and *print_line,* we use pre-set templates without any extra processing (e.g., "`print(X)`" is translated into "prints X on the console" and "`print_line(X)`" into "prints X on the console and breaks line").

5.1.3. Formal Constructs

The translation of formal constructs to natural language was not hard on an implementation level, as it only required that we created a description template for each formal construct. The main difficulty was creating a natural language equivalent of the formal construct that was satisfactory to the reader.

For the *for-loop* equivalent of the *fail-loop* (i.e., `for(Arguments:Condition)`), we used the commonly used translation "for each". Since this version of the *for-loop* uses a condition (i.e., a predicate applied to at least one variable argument), we added the expression "that satisfies", to state that the arguments originate from possible solutions of the specified condition (i.e., the values of the variable argument that convert the specified condition in a true statement). So, for example, if we have "`for(X:status(X))`" the resulting natural language description will be "for each X that satisfies status(X)". Similarly, if we have more than one argument, we use commas and the conjunction "and" to separate the arguments and "satisfies" changes to "satisfy" as it is plural: "`for((X,Y,Z):status(X,Y,Z))`" is translated to "for each X, Y and Z that satisfy status(X,Y,Z)".

The *for-loop* equivalent to the *list-iteration-procedure* has a similar translation to the previous *for-loop*. We also use the keywords "for each" but this time with: *element* + "that belongs to" + *list* (e.g., "`for(X in List)`" is translated into "for each X that belongs to List"). The use of "for X in List" also seemed appropriate, as it is very similar to the Python language and is pretty understandable by itself. Still, we considered it was better to use "for each … that belongs to …" as it is slightly more explicit and perhaps more intuitive to programmers that are not familiar with Python. Note that currently, the template used in the generation of formal descriptions always uses the name "List" for the list variable. However, in the generation of natural language descriptions, the program reads the list variable name as it may change in the future.

In the *do-while* construct, we chose to ignore the *do* component and simply list the loop body followed by a description of the *while* component. The descriptions of the *while* component of the do-while construct are different for conditions and procedures. If the *while* component contains a condition (e.g., `while(not(name(N)))`) the description is: "if" + *condition description* + *subject* + "stops, otherwise, it repeats the same process". For example, "`while(not(name(N)))`" is translated into "if Name N exists it stops, otherwise, it repeats the same process". If the *while* component contains a procedure the template is: "if" + *subject* + "manages to successfully" + *procedure description in infinitive* + *subject* + "stops, otherwise, it repeats the same process". For example, "`while(not_successful(processPassword(P))`" is translated into "if it manages to successfully process password P it stops, otherwise, it repeats the same process". Note that the subject is handled in the next step, Text Formatting. The subject can be either the procedure's name or the pronoun "it" (for the sake of simplicity, we used "it" in the examples above). Also, both the fact and the procedure's

descriptions list variables using commas and the conjunction “and”, and eliminate redundant words in case of repetition (just like previously on section 5.1.1).

Finally, the *if-clause* is the most straightforward. In case of an *else* it simply swaps the word with “otherwise” (e.g., “else” is translated into “otherwise”). In case of an *if*, it simply adds “if” + *condition description* + “then”. We already have pre-set condition description templates for all mathematical predicates. For example: “`if(X<N)`” is translated into “if X is smaller than N then”.

Below is a compilation of all the formal constructs and their respective natural language templates:

Table 10 - Natural language templates

Formal Construct	Translation Template
`for(Arguments:Condition)`	"for each"+*arguments description*+"that satisfies(y)"+*condition*
`while(not(Condition))`	“if” + *condition description* + *subject* + “stops, otherwise, it repeats the same process”
`while(not_successful(Procedure))`	“if” + *subject* + “manages to successfully” + *procedure description in infinitive* + *subject* + “stops, otherwise, it repeats the same process”
`for(Element in List)`	*element* + “that belongs to” + *list*
`if(Condition)`	“if” + *condition description* + “then”
`else`	"otherwise"

After translating everything in the formal description into natural language, we move on to the next step. The text only needs formatting to join together the various excerpts and small descriptions into a complete and coherent description.

5.2. Text Formatting

Having translated every single formal element into natural language, the text only needs to be formatted in order to be considered a proper natural language description. The text formatting consists of four main components, capitalization, adding punctuation, adding subjects and adding discourse connectors.

As we have shown before with arguments (see sections 5.1.1 and 5.1.3) we prefer to complement every single listing regardless of whether it is a series of procedures, arguments or even loops, by adding the conjunction “and” to link the last two elements, and commas to unite the rest. In the case of procedures and loops, we use this listing technique to not only increase the

understandability of the text, but also to infer that there is an underlying sequence. Examples: ‘prints “Insert name” on the console reads Name prints Name one the console’ is formatted into ‘prints “Insert name” on the console, reads Name and prints Name on the console’ ; “for each U that satisfies university(U) for each D that satisfies department(U,D) for each S that satisfies student(D,S)” is formatted into “for each U that satisfies university(U), for each D that satisfies department(U,D) and for each S that satisfies student(D,S)”.

In terms of punctuation, besides listings, commas are added after *for-loop* descriptions (both kinds) and before subjects in *while* descriptions. Periods are added at the end of sentences. A sentence usually corresponds to the top level of indentation (besides the header). The only exception to this is the *while* component of the do-while construct, which begins a new sentence after its description (because its description is usually the most verbose). Below we have two examples of sentence splitting with indentation:

Formal description:

```
displayStudents(){
     for(U:university(U)){
          for(D:department(U,D)){
               for(S:student(D,S)){
                    print_line(S)
               }
          }
     }
}
```

Natural language description:

For each U that satisfies university(U), for each D that satisfies department(U,D) and for each S that satisfies student(D,S), displayStudents prints "S" on the console and breaks line.

Formal description:

```
displayPeople(){
     for(Person:gender(Person,female)){
          print_line(Person)
     }
     for(Person:gender(Person,male)){
          print_line(Person)
     }
}
```

Natural language description:

For each Person and female that satisfy gender(Person,female), displayPeople prints "Person" on the console and breaks line. Then, for each Person and male that satisfy gender(Person,male), displayPeople prints "Person" on the console and breaks line.

The natural language description of the first program (*displayStudents*) only has one sentence, as it only has one top level indentation block. The second program’s (*displayPeople*) natural language description has two sentences, seeing as it has two top level indentation blocks. (Note that both

programs have zero arguments and so they do not have the introductory phrase stating what arguments they receive).

After the sentences have been properly split using periods, the first letter of every phrase is capitalized except if the first word of the phrase is a program name (e.g., "displayStatus prints…" would not be capitalized but "it prints …" would be "It prints …"). Additionally, subsequent phrases after the first one (excluding the first phrase that states a program's arguments if they exist) begin with an adverb (except if the phrases have been split by the *while* template). These adverbs are contained in the lexicon and are meant to communicate sequence to the reader (e.g., "then", "subsequently").

Finally, subjects are either the program name or the pronoun "it" (usually the program name is used only once per phrase. The following references to the subject use the pronoun). They are placed at the beginning of phrases if the phrase does not start with a *for-loop* description, otherwise they are placed after the *for-loop* (or after the last one if there is more than one in a row, see *displayStudents*' example). Subjects are also placed after *if-clause* descriptions and in the middle of *while* descriptions (see section 5.1.3).

After the text has been properly formatted the final natural language description is finally showed to the user. Next, an additional example of a formal description and its corresponding natural language description are provided:

```
biggerThan(N,List){
     for(X in List){
          if(X>N){
               print_line(X," is bigger than ",N)
          }
          else{
               print_line(X," is not bigger than ",N)
          }
     }
}
```

Natural language description:

biggerThan receives N and List. For each X that belongs to List, if X is bigger than N then biggerThan prints "X is bigger than N" on the console and breaks line, otherwise it prints "X is not bigger than N" on the console and breaks line.

The natural language descriptions of bigger programs can sometimes become too verbose or overly complex. For example, the formal description above, is arguably easier to understand than the natural language one. We delve into this topic further in chapter 6.

The generation of a natural language description concludes our two-step approach for description generation. Finally, to validate the quality of our descriptions (and consequentially the two-step approach) they were evaluated by experts (see chapter 7).

6. Compositionality

Compositionality is the principle that a bigger, more complex expression can be looked at as the composition of simpler, less complex components. There are basically two kinds of compositionality: sequencing and several types of embedding. Embedding consists of creating a more complex block by nesting one block within another one. Sequencing consists of the simple concatenation of various blocks in a row. In this project, compositionality is applied in the creation of bigger descriptions, both formal and natural from instances of smaller description templates. Since it is used in both stages of our two-step approach, we can look at the various levels of composition and if they maintain themselves throughout the various transformations.

The main programming patterns we have tackled in this project were *fail-loops*, *repeat-loops* and *list-iteration-procedures*. This chapter explains all the most important concepts regarding our analysis of compositionality, thus it exemplifies compositionality using all three patterns.

To make the explanation easier, we start with a simpler version of the *fail-loop* pattern and then compare it with its more complex versions. In its simplest version, a *fail-loop* pattern consists of a Prolog code fragment with a sequence of two clauses defining the same procedure. The body of the first clause must contain a code fragment (most likely a predicate applied to its arguments) that can have more than one solution and must end with the invocation of *fail/0* predicate. The second clause is a simple clause, which, for the moment, we assume to be a simple fact.

```
displayStatus:-
    status(Person, Status),
    write(Person),
    write(' is '),
    write(Status),
    nl,
    fail.
displayStatus.
```

(a)

```
displayStatus(){
    for((Person,Status):status(Person,Status)){
          print_line(Person," is ",Status)
     }
}
```

(b)

The formal description corresponding to the code fragment (a), which is an instance of the simplified version of the *fail-loop*, is the formal description fragment (b). Now we explain that this simplified version of the *fail-loop* pattern is not enough for preserving compositionality.

Let us consider, for a moment, the formal description (d):

```
displayChildren() {
   for( Parent : parent(Parent) ) {
      print_line(Parent),
      for( X : child(Parent, X) ) {
         print_line(X)
      }
   }
}
```

(d)

The formal description (d) is a case of compositionality in which one simpler block (the nested for construct) is embedded in a second block (the outer for construct) yielding a more complex description. Thus, in (d), the compositionality operator is embedding (the inner for construct is nested within the outer for construct). If the transformation from Prolog to the corresponding intermediate formal description would preserve compositionality, the original Prolog code that would be converted to (d) would have to be something as (c1):

```
displayChildren:-
   parent(Parent),
   write(Parent),nl,
   ((child(Parent, X), write(X), nl, fail); true)
   fail.
displayChildren.
```

(c1)

However, this is not the usual Prolog code pattern. The usual Prolog program that is converted to (d) is not (c1). The usual program is (c2), which does not correspond to the same kind of embedding of (d):

```
displayChildren:-
   parent(Parent),
   write(Parent),nl,
   child(Parent, X), write(X), nl,
   fail.
displayChildren.
```

(c2)

Regardless, we can preserve compositionality, in the conversion from Prolog to the intermediate formal description, if we consider that the *fail-loop* programming pattern is much more general than the simplified version used so far in the explanation. The new more general version of the *fail-loop* pattern is a sequence of n+1 clauses defining the same procedure. The body of each of the first n clauses must include m > 0 code fragments with more than one solution each, and must end with a call to the *fail/0* predicate. The last clause (clause n+1) is a simple clause, most often a simple fact (in its most general case, a clause that does not fail, in case of successful execution). The *displayStudents/0* Prolog procedure corresponds to this more general pattern. It has two clauses. The first contains three code fragments with more than one solution (i.e., university(U), department(D, U), student(U, D, S)) and ends with fail. The second clause is just a fact.

Original program:

```
displayStudents:-
    university(U),
    department(U, D),
    student(D, S),
    write(S),
    nl,
    fail.
displayStudents.
```

Formal description:

```
displayStudents(){
     for(U:university(U)){
          for(D:department(U,D)){
               for(S:student(D,S)){
                    print_line(S)
               }
          }
     }
}
```

The intermediate formal description of a Prolog instance of this more general *fail-loop* pattern is definition containing 3 for-loops, each one of the last two embedded in the previous one. Each of these for-loops will have as many embedded other for loops as m-1, in which m is the number of code fragments in the clause that may have several solutions. In the case of *displayStudents/0*, we can observe that the program is composed of two clauses originating a sequence of one loop. The existing loop is comprised of two embedded loops since it has three code fragments in the clause that have several solutions. To further exemplify, this general pattern, here is the program *displayPeople/0* and its corresponding formal description:

Original program:

```
displayPeople:-
    gender(Person, female),
    write(Person),
    nl,
    fail.
displayPeople:-
    gender(Person, male),
    write(Person),
    nl,
    fail.
displayPeople.
```

Formal description:

```
displayPeople(){
     for((Person,female):gender(Person,female)){
          print_line(Person)
     }
     for((Person,male):gender(Person,male)){
          print_line(Person)
     }
}
```

As, we can see in this example the fail-loop pattern is composed of three clauses, two clauses containing m=1 code fragments with possibly more than one solution and one simple fact. This is equivalent to two loops in a row in the formal description. Both loops have no embedded loops as their respective clause only has one code fragment with more than one solution. We can then take these basic, indivisible code patterns and compose bigger programs with them by using the sequence compositionality operator. Next, we show an example program composed of *displayStudents/0* and *displayPeople/0*, as well as a mock formal description (this description is not generated by our software, and it is purely to demonstrate sequencing compositionality):

Original program:

```
displayAll:-
    displayStudents,
    displayPeople.
```

Formal description:

```
displayAll(){
     for(U:university(U)){
          for(D:department(U,D)){
               for(S:student(D,S)){
                    print_line(S)
               }
          }
     }
     for((Person,female):gender(Person,female)){
          print_line(Person)
     }
     for((Person,male):gender(Person,male)){
          print_line(Person)
     }
}
```

Similar logic applies to both the *repeat-loop* and the *list-iteration-procedure*. All of these patterns (i.e., *fail-loop, repeat-loop* and *list-iteration-procedure*) are looked at as more general patterns that can contain multiple clauses. In the case of the *repeat-loop* it is comprised of a single clause with m-1 embedded loops where m is the number of code fragments that can fail that appear after *repeat/0* in the body of the clause. Next is an example of a repeat-loop with one clause and one embedded loop:

Original program:

```
inputCredentials :-
    repeat,
    write('Username:'),
    nl,
    read(Name),
    name(Name),
    write('Password:'),
    nl,
    read(P),
    password(Name, P),
    write('Credentials accepted').
```

Formal description:

```
inputCredentials(){
     do{
          do{
               print_line("Username:")
               read(Name)
          } while(not(name(Name)))
          print_line("Password:")
          read(P)
     } while(not(password(Name,P)))
     print("Credentials accepted")
}
```

We can sequentially compose many Prolog programs using any combination of these various patterns (e.g., *fail-loop+repeat-loop*, *fail-loops+list-iteration-procedure* etc).

Regarding natural language descriptions, compositionality is much more straightforward. Since each description template handles only a single loop (instead of the overarching programming pattern in the case of the *fail-loop* and the *repeat-loop*), we have simple embedding compositionality. Below is the formal description of *displayStudents/0* and its corresponding natural language description:

Formal description:

```
displayStudents(){
     for(U:university(U)){
          for(D:department(U,D)){
               for(S:student(D,S)){
                    print_line(S)
               }
          }
     }
}
```

Natural language description:

For each U that satisfies university(U), for each D that satisfies department(U,D) and for each S that satisfies student(D,S), displayStudents prints "S" on the console and breaks line.

As we can observe, the natural language template, describes every single loop individually, and the final description is created by simply nesting each successive description (and formatting them). Below is an additional example of a formal description and its resulting natural language description:

Formal description:

```
inputCredentials(){
     do{
          do{
               print_line("Username:")
               read(Name)
          } while(not(name(Name)))
          print_line("Password:")
          read(P)
     } while(not(password(Name,P)))
     print("Credentials accepted")
}
```

Natural language description:

inputCredentials prints "Username:" on the console, breaks line and reads Name from user input. If Name exists, it stops, otherwise it repeats the same process. Then, it prints "Password:" on the console, breaks line and reads password P from user input. If password Name, P exists, it stops, otherwise it repeats the same process. Finally, it prints "Credentials accepted" on the console.

The programming patterns identified are the basic blocks that constitute many procedural Prolog programs. They can be used to compose more complex programs, and the compositionality will maintain itself from the previous level (i.e., from Prolog program to formal description and from formal to natural language description). Despite compositionality maintaining itself from one successive level to another, it may not be the same in the original Prolog program and the natural language description. This is because the programming patterns at the Prolog level are too abstract and when converted to formal constructs they are more easily decomposable into smaller fragments (e.g., each loop imbedded in an overarching *fail-loop*). The use of compositionality proved itself to be very effective in simplifying the processing of bigger programs. Despite this, using compositionality for the creation of natural language descriptions may, in more complex cases of embedding, not be the more adequate approach for the creation of helpful, understandable texts. The use of deep embedded compositionality in natural language originates descriptions too complex to be easily understood. In the example above, the natural language description is arguably confusing and too verbose compared to the formal description. Additionally, it is hard to express that the loops are embedded in one another. In the shown example, it can be hard to understand if the two loops described are done in sequence or one is embedded in the other. Still, the use of compositionality in the creation of formal descriptions, appears to be a valid methodology and provides satisfying results.

7. Results' Evaluation

The purpose of the generated descriptions, both formal and in natural language, is to help the reader understand the original Prolog code. As such, the most pressing concern is to evaluate if the generated descriptions do in fact, represent the original Prolog programs. For the evaluation of the produced descriptions we chose a panel of experts. Only after the descriptions' accuracy has been validated by experts, can we start to investigate their pedagogical value in students and newcomers learning Prolog.

The chosen panel of experts consists of:

- Two Professors from DCTI (Department of Science and Information Technologies) in ISCTE-IUL, both of which, have worked or currently work with Prolog
- One Professor from the Artificial Intelligence course at ISCTE-IUL, where Prolog is taught and practiced
- One Senior Software Engineer from Oracle who has worked extensively in Prolog and even developed a Java implementation of the Prolog interpreter [22]
- Seven former students of the Artificial Intelligence course at ISCTE-IUL whose grades were superior to 18 (out of 20)

Additionally, we also published a form with identical questions in the online community *r/Prolog* (https://www.reddit.com/r/prolog/). Being an online Prolog community, we considered their feedback, as non-expert users of Prolog, could be helpful to improve our descriptions. Unfortunately, r/Prolog is a small community with few members. The community only has 2433 members (as of 30/07/2019), and very few active members. In the published form, we only obtained a total of three answers. Three answers do not have statistical relevance, thus we did not consider them, except for the case of a qualitative comment about one of the generated natural language descriptions.

In order to evaluate the generated descriptions, we used a Likert scale with values ranging from 1 to 5, with 1 being strong disagreement that the presented description is accurate and 5 being strong agreement. The three main questions asked aimed to find out: 1- if the formal description accurately describes the Prolog program; 2- if the natural language description accurately describes the Prolog program; 3- if the natural language description accurately describes the formal description. Additionally, we allowed the experts to comment on the presented descriptions (these comments were completely optional). Finally, we presented some of our experts with just our generated natural language descriptions, together with a small introduction (explaining the definitions previously contained in the Prolog knowledge base) and challenged them to recreate the equivalent Prolog program. The objective was to find if the recreated programs matched the original programs, meaning that the natural language descriptions accurately represent the original program.

A total of seven programs (and respective descriptions) were evaluated. The seven programs contain all the programming patterns investigated in this project, as well as various levels of complexity

for each one. Each expert answered only a limited number of questions according to their availability. The number of questions were not equal for all experts. The results shown below do not have any statistical display (e.g., a graph, a table), because of two reasons. Firstly, the answers are few (i.e., we had a total of eleven experts for this evaluation). And secondly, this is an evaluation by experts and not a general public survey.

Below, we present the various Prolog programs and their respective formal and natural language descriptions, followed by the evaluation of our experts.

7.1. Program 1 – Simple *fail-loop*

Original program:

```
displayStatus:-
    status(Person, Status),
    write(Person),
    write(' is '),
    write(Status),
    nl,
    fail.
displayStatus.
```

Formal description:

```
displayStatus(){
    for((Person,Status):status(Person,Status)){
          print_line(Person," is ",Status)
     }
}
```

Natural language description:

For each Person and Status that satisfy status(Person,Status), displayStatus prints "Person is Status" on the console and breaks line.

Questions:

- The formal description accurately describes the Prolog code:
  A total of 3 experts replied. One expert rated 4 and two experts rated 5.

- The natural language description accurately describes the Prolog code:
  A total of 1 expert replied. The expert rated 5.

- The natural language description accurately describes the formal description:
  A total of 1 expert replied. The expert rated 5.

- Generate the original program based on the natural language description:
  This problem was presented to only one expert. The expert generated an identical program to the original.

Additional comments:

"Your descriptions offer no information on the order in which the Person/Status values are displayed (which is defined in Prolog)." – member of r/Prolog

7.2. Program 2 – Fail-loop with embedded loops

Original program:

```
displayStudents:-
    university(U),
    department(U, D),
    student(D, S),
    write(S),
    nl,
    fail.
displayStudents.
```

Formal description:

```
displayStudents(){
     for(U:university(U)){
          for(D:department(U,D)){
               for(S:student(D,S)){
                    print_line(S)
               }
          }
     }
}
```

Natural language description:

For each U that satisfies university(U), for each D that satisfies department(U,D) and for each S that satisfies student(D,S), displayStudents prints "S" on the console and breaks line.

Questions:

- The formal description accurately describes the Prolog code:
  A total of 3 experts replied. One expert rated 4 and two experts rated 5.

- The natural language description accurately describes the Prolog code:
  A total of 5 experts replied. All five experts rated 5.

- The natural language description accurately describes the formal description:
  A total of 3 experts replied. One expert rated 4 and two experts rated 5.

- Generate the original program based on the natural language description:
  This problem was presented to only one expert. The expert generated an identical program to the original.

7.3. Program 3 – Fail-loop with sequencing loops

Original program:

```
displayPeople:-
    gender(Person, female),
    write(Person),
    nl,
    fail.
displayPeople:-
    gender(Person, male),
    write(Person),
    nl,
    fail.
displayPeople.
```

Formal description:

```
displayPeople(){
     for(Person:gender(Person,female)){
          print_line(Person)
     }
     for(Person:gender(Person,male)){
          print_line(Person)
     }
}
```

Natural language description:

For each Person that satisfies gender(Person,female), displayPeople prints "Person" on the console and breaks line. Then, for each Person that satisfies gender(Person,male), displayPeople prints "Person" on the console and breaks line.

Questions:

- The formal description accurately describes the Prolog code:
  A total of 4 experts replied. All four experts rated 5.

- The natural language description accurately describes the Prolog code:
  A total of 3 experts replied. One expert rated 4 and two experts rated 5

- The natural language description accurately describes the formal description:
  A total of 2 experts replied. One expert rated 4 and one expert rated 5.

- Generate the original program based on the natural language description:
  This problem was presented to one expert only. The expert generated an identical program to the original.

Additional comments:

"As variáveis do programa são apresentadas com ou sem aspas. Seria melhor se fosse escolhido um grafismo qualquer sempre igual para distinguir variáveis de outros símbolos (mas uniforme), por exemplo <Pessoa>" – Expert [The comment pertains to the use of quotes in the description to distinguish text from variables. The expert suggests it would be better to use some sort of symbol (e.g., <Person>)]

7.4. Program 4 – Simple *repeat-loop*

Original program:

```
inputPassword:-
    repeat,
    write('Insert password:'),
    nl,
    read(Password),
    processPassword(Password).
```

Formal description:

```
inputPassword(){
     do{
          print_line("Insert password:")
          read(Password)
     } while(not_successful(processPassword(Password)))
}
```

Natural language description:

inputPassword prints "Insert password:" on the console, breaks line and reads Password from user input. If inputPassword manages to successfully process Password, it stops, otherwise, it repeats the same process.

Questions:

- The formal description accurately describes the Prolog code:
  A total of 5 experts replied. All five experts rated 5.

- The natural language description accurately describes the Prolog code:
  A total of 4 experts replied. All four experts rated 5.

- The natural language description accurately describes the formal description:
  A total of 3 experts replied. All three experts rated 5.

- Generate the original program based on the natural language description:
  This problem was presented to two experts. The two generated identical programs to the original.

7.5. Program 5 – *Repeat-loop* with embedded loop

Original program:

```
inputCredentials:-
    repeat,
    write('Username:'),
    nl,
    read(Name),
    name(Name),
    write('Password:'),
    nl,
    read(P),
    password(Name, P),
    write('Credentials accepted').
```

Formal description:

```
inputCredentials(){
     do{
          do{
               print_line("Username:")
               read(Name)
          } while(not(name(Name)))
          print_line("Password:")
          read(P)
     } while(not(password(Name,P)))
     print("Credentials accepted")
}
```

Natural language description:

inputCredentials prints "Username:" on the console, breaks line and reads Name from user input. If Name exists, it stops, otherwise it repeats the same process. Then, it prints "Password:" on the console, breaks line and reads password P from user input. If password Name, P exists, it stops, otherwise it repeats the same process. Finally, it prints "Credentials accepted" on the console.

Questions:

- The formal description accurately describes the Prolog code:
  A total of 3 experts replied. All three experts rated 5.

- The natural language description accurately describes the Prolog code:
  A total of 4 experts replied. One expert rated 1 and three experts rated 5.

- The natural language description accurately describes the formal description:
  A total of 3 experts replied. One expert rated 1 and two experts rated 5.

- Generate the original program based on the natural language description:
  This problem was presented to one expert only. The expert generated an identical program to the original.

Additional comments:

“If foo exists, it stops" seems like a poor choice of words. Presumably you mean something like "If foo exists, it stops repeating the attempt to get foo", but saying just "it stops" makes it sound like the program stops executing altogether. Also the 'repeats same process' is ambiguous about the 'process'.” – member of r/Prolog

“ "if ... exists it stops" can be interpreted as "inputCredentials stops". The description should be " repeat ... until ... exists" “ – Expert

7.6. Program 6 – Simple *list-iteration-procedure*

Original program:

```
printList([X|Rest]):-
    write(X),
    nl,
    printList(Rest).
printList([]).
```

Formal description:

```
printList(List){
     for(X in List){
          print_line(X)
     }
}
```

Natural language description:

printList receives List. For each X that belongs to List, printList prints "X" on the console and breaks line.

Questions:

- The formal description accurately describes the Prolog code:
  A total of 4 experts replied. All four experts rated 5.

- The natural language description accurately describes the Prolog code:
  A total of 4 experts replied. One expert rated 4 and three experts rated 5.

- The natural language description accurately describes the formal description:
  A total of 2 experts replied. One expert rated 4 and one expert rated 5.

- Generate the original program based on the natural language description:
  This problem was presented to a single expert. The expert generated an identical program to the original.

7.7. Program 7 – *List-iteration-procedure* with embedded conditional

Original program:

```
biggerThan(N,[X|Rest]):-
    X>N,
    write(X),
    write(' is bigger than '),
    write(N),
    nl,
    biggerThan(N, Rest).
biggerThan(N,[X|Rest]):-
    X=<N,
    write(X),
    write(' is not bigger than '),
    write(N),
    nl,
    biggerThan(N, Rest).
biggerThan(_,[]).
```

Formal description:

```
biggerThan(N,List){
     for(X in List){
          if(X>N){
               print_line(X," is bigger than ",N)
          }
          else{
               print_line(X," is not bigger than ",N)
          }
     }
}
```

Natural language description:

biggerThan receives N and List. For each X that belongs to List, if X is bigger than N then biggerThan prints "X is bigger than N" on the console and breaks line, otherwise it prints "X is not bigger than N" on the console and breaks line.

Questions:

- The formal description accurately describes the Prolog code:
  A total of 2 experts replied. All two experts rated 5.

- The natural language description accurately describes the Prolog code:
  A total of 2 experts replied. All two experts rated 5.

- The natural language description accurately describes the formal description:
  A total of 1 expert replied. The expert rated 5.

- Generate the original program based on the natural language description:
  This problem was presented to two experts. The two generated identical programs to the original.

7.8. Results summary

Although the number of expert evaluations was not large, here we present a table to better summarize the results.

Table 11 - Results summary

	Total answers	Rate 1	Rate 2	Rate 3	Rate 4	Rate 5	Average
Formal description's accuracy of Prolog code	24	0	0	0	2	22	4.92
Natural description's accuracy of Prolog code	23	1	0	0	2	20	4.74
Natural description's accuracy of formal description	15	1	0	0	3	11	4.53

Overall the results were clearly positive. Most programs' descriptions were agreed to be accurate by our experts, with most of them receiving 5 (the maximum). The only description to get a negative rating was the natural language description of program 5 (i.e., *repeat-loop* with embedded loop). Despite the formal description being very high rated, the natural language one was harshly criticized with the lowest rating possible. One of experts (presumably the one that left the negative

rating), as well as, one of the members of r/Prolog commented on the generated descriptions. Both comments criticized the use of the word “stops”, as it misleads the reader into thinking the whole program stops (not just the current loop). This choice became even more confusing since the description had one outer loop and one embedded loop, and as such, the word stops was repeated twice in the description.

Taking the feedback from our experts we changed the natural language template for the *do-while* pattern. Now instead of using: “if” + *success description* + “stops, otherwise, it repeats the same process”, we use simply: “repeats this process until” + *success description.* After generating some test descriptions, this template seemed to fit much better and does not use the word stop which was considered incorrect/misplaced by some of our experts. As an example, here is the natural language description of program 5 using the new template:

inputCredentials prints "Username:" on the console, breaks line, reads Name from user input and repeats this process until Name is true. Then, it prints "Password:" on the console, breaks line, reads password P from user input and repeats this process until password Name, P is true. Finally, it prints "Credentials accepted" on the console.

One of our experts also commented on making clearer the difference between variables and text, by using some sort of symbol and a member of r/Prolog commented on giving additional information about the facts in the knowledge base. The first feedback is helpful to make the descriptions more understandable. A special purpose symbol could be used to better identify variables. Due to the limited time in this project we did not implement those changes but consider them a possible improvement to do in the future. Regarding the second feedback, currently the descriptions are generated only through the clause bodies of the given procedure and do not describe the various code fragments in them. That is also a future improvement we would like to tackle as it would make our descriptions better suited for bigger and more complex programs (a brief example of how a resulting description could look like was shown in chapter 6).

Regarding the questions in which the experts recreated the programs, they were a complete success. In these questions the experts had to recreate the original Prolog programs based only on our natural language description. All nine responses created programs that were identical to the original program. Surprisingly, even program 5 whose natural language description was criticized, ended up with a correct answer. This leads us to conclude that despite some wording problems, the descriptions in general are understandable and considered as equivalent to the original Prolog programs.

8. Conclusion

Code comprehension can be a hard problem to tackle. Through our two-step approach we managed to create descriptions in order to help newcomers better understand procedural Prolog. In our two-step approach we generate a formal description and a natural language description. The generated descriptions were considered accurate by our experts, which leads us to conclude that they are explanatory of the original Prolog programs. The accuracy of the generated descriptions also validates the use of our two-step approach, as a possible way of tackling this problem. However, the proposed approach can only be completely validated when we apply it to a significantly larger subset of the Prolog language.

Compositionality maintains itself throughout each consecutive level (Prolog program to formal description to natural language description), but the compositionality present in the natural language description may not be the same as the one in the original program (mostly due to the use of very abstract programming patterns as the basic blocks of compositionality at the Prolog level). The use of compositionality proved itself to be effective in processing bigger programs and descriptions. However, natural language descriptions of bigger programs created with compositionality seem to be harder to understand.

In regard to future work, there are many possible directions this project can be improved. Since the two-step approach handles so many different topics (i.e., code comprehension, description generation, natural language generation, compositionality), it can be improved in various ways, many of which could be projects by themselves.

The first, and most obvious, is that the approach can be complemented with templates for different languages, making it a multi-language solution. In the future, we could generate descriptions in Portuguese, Spanish, etc. We could, for example, add equivalent natural language templates in different languages and allow the user to choose a language when generating a description.

The proposed two-step approach opens the door for future description generation of other languages besides Prolog. This could be done by the addition of the relevant code patterns (in Python, Java, Lisp or another language) and transforming them to our formal language. This would even be easier to do for imperative languages than for Prolog because of the intended similarity between them and the language we have designed for the intermedium formal descriptions. For example, the *list-iteration-procedure* pattern formal equivalent is already very similar to Python code:

Formal description (printed in Python style)

```
printList(List):
     for(X in List):
          print_line(X)
```

Equivalent Python code

```
def printList(List):
     for X in List:
          print(X)
```

Additionally, since the generation of natural language from the intermedium formal descriptions is already done, extending our approach to other programming languages would not require any changes to the second step.

The similarity between our formal descriptions and other programming languages could also be used to convert procedural Prolog into other programming languages. Using this approach, we could write code in Prolog and transform it into working code in Python for example.

Another improvement that could be made is to compose bigger descriptions using smaller descriptions using compositionality. For example, if we have the following program:

Prolog program:

```
displayAll:-
    displayStudents,
    displayPeople.
```

Formal description:

```
displayAll(){
     displayStudents
     displayPeople
}
```

Currently the software only describes the clause body of *displayAll/0*, but the body itself may not be sufficiently explanatory. We could improve upon this by having the software recursively call itself and describe *displaysStudents/0* and *displayPeople/0*'s bodies and using those descriptions to compose *displayAll/0*'s description. An example of what this could look like is shown below:

Formal description of *displayStudents/0*:

```
displayStudents(){
     for(U:university(U)){
          for(D:department(U,D)){
               for(S:student(D,S)){
                    print_line(S)
               }
          }
     }
}
```

Formal description of *displayPeople/0*:

```
displayPeople(){
     for(Person:gender(Person,female)){
          print_line(Person)
     }
     for(Person:gender(Person,male)){
          print_line(Person)
     }
}
```

Prolog program:

```
displayAll:-
    displayStudents,
    displayPeople.
```

Equivalent formal description:

```
displayAll(){
     for(U:university(U)){
          for(D:department(U,D)){
               for(S:student(D,S)){
                    print_line(S)
               }
          }
     }
     for((Person,female):gender(Person,female)){
          print_line(Person)
     }
     for((Person,male):gender(Person,male)){
          print_line(Person)
     }
}
```

This approach could be very useful for formal descriptions, but we fear it may be inadequate to generate natural language descriptions, since, as we have seen throughout this project, natural descriptions of big programs can become too verbose and confusing.

All of the tools the program uses to generate descriptions could also be expanded. This includes the various code patterns, formal representation templates, natural representation templates and lexicon. The addition of new programming patterns would most likely involve the addition of new templates and respective tags. Programming patterns that are similar in control flow could also reuse the already existing tags and formal representation templates. The lexicon is the cornerstone of the software's natural language generation and expanding it will greatly improve its capacity to describe various programs.

Our current approach for the finding contextual information that improves the semantic understanding the program creates about the diverse code elements (e.g., variable names) is only used for the *read* action. In the future we could contemplate additional actions and even more complex procedures. This could be done by creating specific templates for those actions/procedures and adding some sort of flag or tag that signalized that the description needs extra contextual information. We could also add complementary semantic and or pragmatic information manually, which currently the software does not allow, except in a few cases (e.g., stating if a code fragment can have more than one solution or stating if a code fragment can fail).

This project focused on procedural Prolog. One natural next step would be to also describe declarative Prolog, moving one step closer of the final goal which would be to describe Prolog as a whole. This would possibly involve the definition of new relational constructs for the language used in intermedium formal descriptions

Finally, if we want in fact to use these descriptions for educational purposes, their pedagogical value would have to be tested. Despite these descriptions being established as accurate, that does not necessarily mean they are the best tool to help newcomers learn Prolog. Further investigation into their educational use would have to be done.

All of the code produced throughout this project is available for further improvements and/or experimentation at https://github.com/diogomfarinha/Prolog_Description_Generator.

9. References

[1] Colmerauer, A., & Roussel, P. (1992). The birth of Prolog. *History of Programming Languages---II*, (November), 331–367. https://doi.org/10.1145/234286.1057820

[2] Kumar, A. N. (2002). Prolog for imperative programmers. *J. Comput. Small Coll.*, *17*(6), 167–181. Retrieved from http://portal.acm.org/citation.cfm?id=775742.775771

[3] Storey, M. A. (2006). Theories, tools and research methods in program comprehension: Past, present and future. *Software Quality Journal*, *14*(3), 187–208. https://doi.org/10.1007/s11219-006-9216-4

[4] Reiter, E., & Dale, R. (1997). Building applied natural language generation systems. *Natural Language Engineering*, *3*(1), 57–87. https://doi.org/10.1017/S1351324997001502

[5] Gatt, A., & Krahmer, E. (2018). Survey of the State of the Art in Natural Language Generation: Core tasks, applications and evaluation. *Journal of Artificial Intelligence Research*, *61*(c), 65–170. https://doi.org/10.1613/jair.5477

[6] Saddic, G. N., Ebert, M. B., Dhume, S. T., & Anumula, K. R. (2002). Automatic Generation of Natural Language Summaries for Java Classes. *Methods in Molecular Biology*, *194*, 23–36. https://doi.org/10.1109/ICPC.2013.6613830

[7] Rastkar, S., Murphy, G. C., & Bradley, A. W. J. (2011). Generating natural language summaries for crosscutting source code concerns. *IEEE International Conference on Software Maintenance, ICSM*, (Section II), 103–112. https://doi.org/10.1109/ICSM.2011.6080777

[8] McBurney, P. W., & McMillan, C. (2014). Automatic documentation generation via source code summarization of method context. *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*, 279–290. https://doi.org/10.1145/2597008.2597149

[9] Wu, H., & Wang, H. (2009). Revisiting pivot language approach for machine translation. *ACL-IJCNLP 2009 - Joint Conf. of the 47th Annual Meeting of the Association for Computational Linguistics and 4th Int. Joint Conf. on Natural Language Processing of the AFNLP, Proceedings of the Conf.*, (August), 154–162. https://doi.org/10.3115/1687878.1687902

[10] Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

[11] Clocksin, W.F. and Mellish, C.S 2003 Programming in Prolog (5th Ed.), Springer Verlag

[12] Sterling L. and Shapiro E. 1994. The Art of Prolog (2nd Ed.): Advanced Programming Techniques. MIT Press, Cambridge, MA, USA.

[13] Banbara, M., Banbara, M., Tamura, N., Tamura, N., Inoue, K., & Inoue, K. (2006). Prolog Cafe: a Prolog to Java translator system. *Lecture Notes in Computer Science*, *4369*, 1. https://doi.org/http://dx.doi.org/10.1007/11963578_1

[14] Dr. R.S. Kamath, Mrs. S.S. Jamsandekar, D. R. K. K. (2015). EXPLOITING PROLOG AND NATURAL LANGUAGE PROCESSING FOR SIMPLE ENGLISH GRAMMAR.

[15] Costa, F., Ouyang, S., Dolog, P., & Lawlor, A. (2017). Automatic Generation of Natural Language Explanations. https://doi.org/10.1145/3180308.3180366

[16] McBurney, P. W., Liu, C., McMillan, C., & Weninger, T. (2014). Improving topic model source code summarization. In *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014* (pp. 291–294). New York, New York, USA: ACM Press. https://doi.org/10.1145/2597008.2597793

[17] Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016). Summarizing Source Code using a Neural Attention Model. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[18] Sterling, L. (2002). Patterns for Prolog Programming. *Program*, 374–401.

[19] Le, N., & Menzel, W. (2005). Constraint-based Error Diagnosis in Logic Programming. *Towards Sustainable and Scalable Educational Innovations Informed by the Learning Sciences - Sharing Good Practices of Research, Experimentation and Innovation*, 220–227.

[20] “For loop,” Wikipedia, 21-Jun-2019. [Online]. Available: https://en.wikipedia.org/wiki/For_loop [Accessed: 23-Jun-2019].

[21] “Prolog -- call_nth/2,” SWI. [Online]. Available: https://www.swi-prolog.org/pldoc/man?predicate=call_nth/2 [Accessed: 24-Jun-2019].

[22] *Nazguûl Prolog home page*. [Online]. Available: https://na.zgul.me/~prolog/nazgul/ [Accessed: 30-Jul-2019].