UCLouvain

École polytechnique de Louvain

Logic meta-programming to detect coding flaws in student Python source code

Author: **Nathan Corbisier**
Supervisors: **Kim Mens, Julien Liénard**
Reader: **Nikita Tyunyayev**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Contents

Abstract

In introductory computer science courses, instructors often use auto-graders to offer automated feedback and corrections to their students. However, these auto-graders typically rely on unit tests, which only evaluate the code's output. As a result, they may miss bad coding practices or reject almost correct solutions.

One possible solution to this problem is to extract and analyze information directly from the source code written by the students. This approach would enable us to automatically answer questions such as "Does this code contain bad practices?" or "Has the student substituted a 'print' for a 'return'?". This master thesis proposes to investigate the technique of logical meta-programming to create queries that target specific coding patterns in student code. These patterns may represent either common coding idioms or rather frequent coding flaws occurring in code produced by the students. The basic idea is to translate Python code into a logical database and then use a logical programming language to write different queries that can be applied to this database. With this tool, instructors can easily detect bad practices and provide more accurate feedback and corrections to their students.

Acknowledgements

I want to thank everyone who supported me during my studies and this past year.
Thanks to Professor Kim Mens and Julien Liénard for their help, proofreading, advice, and, most importantly, their good mood throughout the year during our frequent meetings.
I would also like to thank Olivier Goletti and all the attendees of the Didapro conference who showed interest in my work.
Thanks to all the students who participated in the end-of-year experiment.
Finally, a special thank you is dedicated to my family and friends, who supported me and have been good listeners and motivators this whole year.

Chapter 1

Introduction

1.1 Context

Learning to code can be a real challenge for students, as it requires a specific way of thinking that can be difficult to get at first. Mechanics that may be obvious to experienced programmers are not necessarily clear to those taking their first steps in the world of programming. For example, loops, arrays, and IF-conditions can be particularly challenging for beginner programmers to master.

At École Polytechnique de Louvain-La-Neuve, the introductory computer science course (LEPL 1401 - Informatique 1 and LINFO1101 - Introduction à la programmation) teaches students the basics of computer science through theoretical lectures and programming exercises on the INGInious platform [1]. This course, taught in Python, will be referred to as CS1 in this thesis, as it is commonly called in the literature.

To assist students in their programming assignments, the Inginious platform provides automated feedback designed to help students learn from their mistakes and address them. Nevertheless, the primary function of these grading systems is to verify the correctness of the program result in accordance with the given task. But, it is at least equally important to address bad practices, such as reliance on hard-coded solutions, to help students better understand programming.

One way to ensure fairness and provide detailed feedback is to have teachers and assistants review the codes alongside the auto-graders. However, this solution is impractical due to its tedious nature, limited productivity, and susceptibility to human error. Additionally, with the continuously growing enrollment in introductory computer science courses, this approach becomes entirely unfeasible.

1.2 Problem

K. Mens et al. [2] have developed an algorithm to mine patterns in source code. They used this algorithm to analyze the source code of approximately 500 students taking the CS1 programming exam. With this algorithm, they can mine different recurring patterns and classify them based on whether they appeared more often in the code of students who received at least 50% or more often in the code of students who received less than 50%. It allows them to discover recurring good or bad practices in programs written by students.

While this algorithm can discover recurring bad practices in a set of multiple codes, it cannot detect these patterns in a single new code. With this understanding, the next step in enhancing automated feedback for future students is to develop a flexible automated tool to detect predefined bad practices in their source code.

1.3 Motivation

Currently, at the École Polytechnique de Louvain, unit tests are used to automatically correct the exercises and even the exam programming questions of the introductory computer sciences course. However, it is important to note that these automated tests are not a foolproof way to assess a student's overall programming knowledge. To illustrate this, let's take an example of an exam question:

Exam question: Approximate the value of π. Several formulas exist to approximate the value of π. One of these formulas is the Gregory-Leibniz series:

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ...)$$

Write a Python function which takes as argument an integer $i \geq 0$ and computes an approximation of π based on the first $i+1$ terms of this series:

```
1 def approx_pi(i):
2     """
3     @pre:   i is an integer such that i >= 0
4     @post:  returns an estimate of pi by summing
5             the first i + 1 terms of the Gregory-Leibniz
                  series
6     """
```

The expected "standard" solution is as follows:

```
1     sum = 0
2     for x in range(i+1):
3         sum += (-1)**x/((2*x)+1)
4     sum = 4*sum
5     return sum
```

With this solution, the student would have a result of 100% for the question.

But sometimes, a simple error can change the calculation's result and, therefore, the student's grade.
For example, consider this solution :

```
1     result = 0
2     for x in range(i+1):
3         res = ((-1)**x / 2*x + 1)
4         result += res
5     return 4*result
```

The student omitted to put brackets around his denominator, changing the result of the sum. The student only scored 8% on this question. This is a very low mark for an error that could have been due to the stress of an exam and for a student

who seems to have managed to handle the basics of programming well.

Contrariwise, consider this code :

```
1     if i == 0 :
2         return 4
3     if i == 1 :
4         x = 4*(2/3)
5         return (x)
6     if i == 2:
7         x = 4* (2/3 + 1/5)
8         return (x)
9     if i == 3:
10        x = 4* (2/3 + 1/5 + 1/7)
11        return x
12    if i == 1000:
13        return 3.1425916543395442
14    if i == 4:
15        x = 4* (2/3 + 1/5 - 1/7 + 1/9)
16        return x
17    if i == 5:
18        x = 4* (2/3 + 1/5 - 1/7 + 1/9 - 1/11)
19        return x
20    if i == 6:
21        x = 4* (2/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13)
22        return x
23    ...
```

This student scored 52% on this question with only hardcoded answers. This is an extreme case, but a student providing this type of code probably has not properly assimilated the basics that the course teaches.

The purpose is not to offer a supposedly "fairer" automated correction by indicating that the first code deserves more points than the second. Instead, the aim is to provide a method to simplify the process for teachers to identify good or bad practices and enable them to evaluate different codes more effectively based on their own standards. To achieve this, we want to provide a simple and effective tool for detecting bad practices in student source code.

1.4 Objectives

This thesis aims to explore the logic metaprogramming technique for creating a structure and a set of predicates to detect bad practices in students' source codes. To achieve this, we have converted each code into its Abstract Syntax Tree representation, which is then transformed into a set of logic facts. Through this technique, we aim to provide an efficient and declarative way of analyzing student codes. Our goal is to establish a logic foundation that allows users to develop their queries to detect bad practices.

1.5 Roadmap

The remainder of this thesis is structured as follows:
We will start by exploring the theoretical aspects and examining various scientific literature resources on different techniques for identifying and detecting bad practices and patterns in source code. This will provide the reader with a comprehensive understanding of the different domains covered in this thesis.
Next, we will discuss the development of our logic programming tool for detecting bad practices in source code.
We will present the initial queries created in our logic programming tool to analyse the source code of exam questions from previous years to detect bad coding practices in these codes.
Additionally, we will analyze the results of an experiment conducted by Master students who used our tool to identify common bad practices in some programs containing coding flaws. We will discuss their results and feedback to evaluate the difficulty level of handling our tool.

Chapter 2

Background Material

This chapter explores the themes and techniques employed throughout this thesis. Our first point of discussion is the concept of "coding flaws", where we offer an explanation and definition to ensure clarity on this term used in the title of this master thesis. Next, we examine in the literature the various techniques employed in this thesis, such as program analysis, (logic) metaprogramming and abstract syntax tree, providing a theoretical basis and justification for our choices. Finally, we end this chapter with an overview of the tools and projects developed, whose aim is somewhat similar to the tool we aim to develop for our master's thesis, highlighting the differences with our work.

2.1 Coding flaws

The term "flaw" is defined by the Cambridge dictionary[1] as: "a fault, mistake, or weakness, especially one that happens while something is being planned or made, or that causes something not to be perfect." By using the term "coding flaws", we aim to convey an idea or image that most people can understand intuitively. But readers seeking a clear definition of the term may find inconclusive results when googling it, with references to coding mistakes and security vulnerabilities, but without a specific definition for the term "coding flaws". In our case scenario, we typically use the term "coding flaws" to represent imperfections in code, which are often specific to students or apprentices in programming. In this section, we will define the term "coding flaws" to provide clarity to the reader and ourselves.

A literature review conducted by Combéfis on techniques for scoring student code [3] highlights three distinct types of errors: **Syntax**, **Runtime**, and **Logic errors**. Syntax errors, such as typing errors, are usually detected by the compiler during

[1] dictionary.cambridge.org/dictionary/english/flaw

program compilation, while runtime errors occur during program execution. These errors are generally easy to identify, resulting in compilation failure or program crashes.
However, logic errors, which emanate from poor understanding or misusing of certain language elements and can lead to incorrect program behaviour without generating error messages, are more challenging to detect.

In this thesis, we are particularly interested in detecting logic errors in student code. Unlike syntax and runtime errors, logic errors require deeper analysis to uncover. Our tool aims to assist in detecting these types of errors in student programs.

In the same literature review [3], Combéfis divides code verification into five stages:

1. Verifying that the code is syntactically correct.
2. Identifying any plagiarism issues.
3. Verifying that the code is functionally valid.
4. Verifying the good performance of the code.
5. Verifying if the code respects certain style and quality conventions.

In this master thesis, we will not be dealing with plagiarism problems. Various techniques for that already exist, some using Abstract Syntax Trees that we will also use for different purposes, which we will discuss in the following sections of this chapter. Compilers determine whether a code is syntactically correct, so we assume that the programs we will analyse are indeed syntactically valid. An area for future improvement would be to consider these syntactically incorrect programs or to create a system like the one proposed by Saikkonen et al. [4], which reduces the language syntax for specific exercises to prohibit (or force) the use of particular statements or constructions. Since the programs analysed are from a CS1 course and are, therefore, relatively small in size and solve simple problems, we are also not too concerned with the performance of the programs. And the unit tests already check the functional correctness of the code.
That leaves us with the code quality. The reader may wonder whether it is helpful to check the quality of the code in an introductory computer science course. Code quality includes many aspects, such as style, readability, maintainability, complexity, etc. Some of these aspects may seem excessive in an introductory computer science course. The students' codes don't need to be maintainable, for example, as no one is going to use the code further, but poor quality code can lead to numerous problems in advanced software, and teaching first-year students certain essential qualities can be beneficial to avoid students making these mistake in future projects

[5][6][7]. Our work will thus focus on detecting such bad practices in student source code.

So let us now define more concretely what we mean by "coding flaws". Coding flaws refer to a broad spectrum of errors, inadequacies, or instances of poor practice within the source code of a software program. These flaws encompass issues ranging from overt syntax errors to more subtle, non-syntactical problems such as poor code smells, problematic coding idioms, anti-patterns, and practices stemming from common misconceptions held by novice programmers. The term "coding flaws" highlights areas in code that, while not necessarily leading to immediate execution errors, represent a divergence from best coding practices. Such divergences signal potential underlying problems in program logic, structure, readability or maintainability and may indicate a deeper lack of understanding or proficiency in coding principles and programming language concepts.

2.2 Program Analysis

In this thesis we aim to create a tool for detecting coding flaws, as explained in the previous section. This tool will go beyond simply checking that student programs produce the expected results and will identify both desirable and undesirable quality properties within the code written by students.
Two primary techniques can be used to analyze code: dynamic and static analysis [8]. This section will delve into both methods and explain why our approach will focus on static analysis only.

2.2.1 Dynamic Program Analysis

Dynamic program analysis technique gathers information about program properties at runtime [9]. Testing and profiling are standard methods used in dynamic analysis, which reason over program behaviour during execution.
Dynamic analysis is particularly useful for object-oriented programming languages like Java, where static analysis is less effective due to dynamic binding and polymorphism. It can handle complex tasks such as memory analysis, identifying deadlock, race detection, etc. It can also be used to measure various metrics [10] like program performance, thanks to its ability to capture the program's runtime behaviour.
An example of a dynamic analysis tool that is widely used is Valgrind. Valgrind [11] is a notable framework for building dynamic analysis tools, offering memory management, threading bugs, and detailed program profiling capabilities.
Readers interested in the various techniques and tools used for dynamic program analysis can learn more in the survey by Gosain and Sharma [12], where they

provide an overview of and compare different existing techniques.

While dynamic analysis can precisely examine the actual runtime behaviour of a program, it has its limitations. The main disadvantage is that it depends on input stimuli and cannot be generalized for all executions. In addition, Ball [9] mentions that dynamic analysis can detect violations of properties and provide helpful information to programmers about the behaviour of their programs but cannot "prove" that a program satisfies a particular property.

As the CS1 course we use as input data for this thesis focuses on Python without a strong focus on the object-oriented paradigm, a dynamic approach may not be the best fit. Dynamic analysis is more relevant for analyzing object-oriented languages and addressing complex problems like memory management, which are typically beyond the scope of an introductory programming course. It can also be argued that the program already undergoes dynamic analysis through unit testing. It involves executing the code to verify that the implementation results are the expected ones. Our objective is to create a complementary analysis that can detect finer-quality details overlooked by unit tests.

2.2.2 Static Program Analysis

Static program analysis examines a program's source code without executing it, intending to detect potential errors, security vulnerabilities, and coding flaws by analyzing the code structure, syntax, and semantics [8]. While the term is usually applied to analysis performed by automated tools, a human analysis, generally called code review, can be considered the most basic form of static analysis. Interpreters and compilers can also be considered basic static code analysis tools because they can detect syntax errors in code that render it unusable. Specialized static analysis tools are used instead to address issues encountered in syntactically correct code that can be compiled and may even produce the expected output.

Various static analysis techniques are employed to assess code syntax, anti-plagiarism, and code quality aspects. Lint [13] is one of the first examples of a static analysis checker for C programs, and its name is now even sometimes used as a common name to identify static analysis tools. Pylint[2] is a static code analyser for Python that checks for errors and for code snippets that break the PEP8[3] conventions. The PEP8 conventions are the standard style guide for the Python language. Static analysis checkers have been written for many other programming languages as

[2]pylint.readthedocs.io
[3]peps.python.org/pep-0008/

well. Checkstyle[4] and PMD[5] are two standard static analysis checkers for Java. CheckStyle, as its name suggests, focuses on Java style, and PMD focuses more on programming flaws such as unused variables, non-simplified expressions, and empty statements. Although these tools were initially designed for software professionals, they are also valuable for educational purposes [14].

Several experiments have been carried out to highlight the effectiveness of static analysis on student source code. For instance, Czerép [15] executed various static code analyses on the C source code of more than 5000 students to highlight some of the most severe and typical issues. Edwards et al. [16] used CheckStyle and PMD on over 500 thousand program submissions made by students over a five-semester period and examined the differences between the error rates of CS major and non-major beginners and also discussed how these patterns change over time as students progress through the CS major course sequence. They observed that identified coding flaws are strongly correlated to producing incorrect programs, even when the problems are eventually fixed. Albluwi and Salter [14] use static analysis tools to detect issues that are orthogonal to problems detected by the compiler or that cause programs to crash and see on an introductory computer science course which are the issues that are the most present in the programs written by the students. They found that the students fixed most of the problems the static analysis tools raised and learned to avoid repeating their mistakes in subsequent exercises.

Many other experiments have been carried out on the subject [17][18]. Striewe and Goedicke [19] review some of these approaches and highlight that a system evaluating submissions with static analysis has to check for mistakes, violated coding conventions, missing code structures, and provide hints on how to address these issues.

Given the success and effectiveness of static analysis in analyzing student source code, the use of static analysis for identifying and addressing issues in student programs has also been chosen as the main focus of this thesis.
Establishing the approach for building our tool is crucial, as different static analysis tools operate differently and may analyze different input types. Some tools analyze bytecode or machine code for languages other than Java. In contrast, others analyze code directly as natural language or transform it into an abstract syntax tree (AST) or an abstract syntax graph (ASG) [19]. Each approach offers distinct detection possibilities, and the choice made may significantly impact the effectiveness and capabilities of our tool.

[4] checkstyle.sourceforge.io
[5] pmd.github.io

2.3 Abstract Syntax Trees Analysis

An Abstract Syntax Tree is a data structure used to represent the form of a program into a branching of operators [20]. An AST is a tree representation of the program, usually the result of the syntax analysis of the source code. The abstract syntax tree is useful in our domain because it is an abstract representation of the code, reducing inessential syntax or formatting that does not impact the program behaviour.

For example, the following simple Python code:

```
1       x = x + 1
2       print("Hello␣World")
```

Produce an AST in Python as can be seen on the Figure 2.1

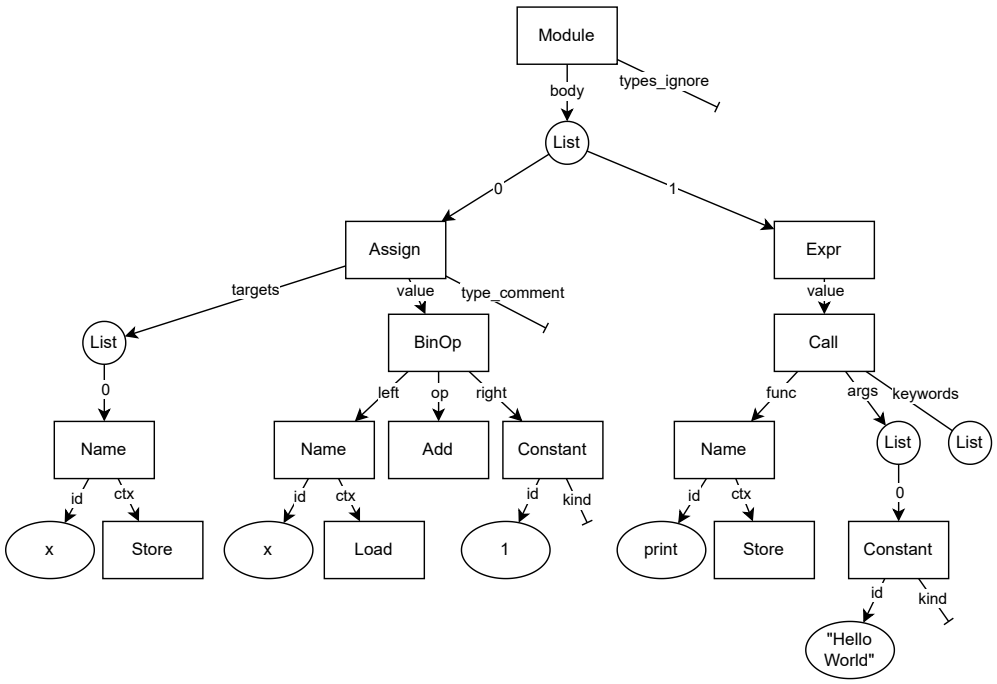

Figure 2.1: Python AST for a simple program

The program source code is represented by a set of nodes, each associated with a certain type describing an element of the program's operation. For example, in figure 2.1, the "Assign" node represents an "assign statement" (a = b). It is

important to note that there is no single kind of AST. In general, each language has its own version with different types and some nuances, but the main principle remains the same.

Manipulating such a data structure has proven very useful in the domain of code clone detection [21] [22] either between different student programs, to avoid cheating or within the same large program, to avoid duplication of code and make the software easier to maintain. This is possible because ASTs provide a structure for comparing source code fragments that focuses on the operations occurring within programs rather than the string labels used to identify the parts. But using the same process, it can also be used in a positive way by computing the distance between the code of a student and the solution [20]. In an introductory programming course, students generally need to solve simple problems, and their solutions can be seen as some kind of code clone of the possible solutions. Such a system enables customised hint generations by considering similar solutions from other students.
Another way to use the AST is to mine for properties that we expect (or not) in the source code. Klinik et al. [23] use such a system to develop their automatic code review writer for Java assignments to detect if certain code properties (such as a coherent class hierarchy or the use of design patterns) are met. In our work, we will be focusing on detecting bad practices or properties in students programs. We will translate common coding flaws into their abstract syntax representation to be matched in the AST of the different programs.

Since mining information in the AST is extremely useful, we need a technique for traversing this structure. In Python, obtaining the AST of a program is simple thanks to the *'ast'* module, but browsing the tree requires some more complex work. In his master thesis, Eric Arts examined different techniques for exploring a Python AST [24]. From using Python *'ast'* module and *'NodeVisitor'* class to using an XML-representation for XPath but also using a Python Wrapper library: PyTraPal. Although these techniques may prove effective, they often require extensive coding, and Arts pinpointed a key issue: it gets quite complex to navigate the AST in all approaches. He identified that Python 3.10's match function, which implements pattern matching in Python, is the most flexible way of navigating this structure. One of the future works proposed in his master thesis is to develop a domain-specific language for exploring the AST. Logic programming has this pattern-matching feature, as we will present in chapter 3.2. Our project to use this property to mine the AST in an efficient and declarative manner could be a solution to Arts problem.

We would also point out that the Computer Engineering department at UCLouvain

has a project called Pyttern, an expressive wildcard language designed to resemble Python syntax. This language allows instructors to define patterns and detect occurrences of common coding flaws in student code. It searches in the AST of the various student programs to find whether the pattern created can be detected in the different codes. Later in this document, when we present the experiment results in which students used our tool, we will compare these results with those of students who used Pyttern to complete the same task.

2.4 (Logic) Meta-programming

Meta-programming is the process of writing computer programs that take other computer programs as their input and/or produce other programs as their output. The program so created is called a meta-program. When writing a meta-program, a fundamental distinction is drawn between the language employed by the input programs, referred to as the object language and the language utilized for the meta-program itself, known as the metalanguage. These two languages are not necessarily distinct: for example, it is entirely plausible to have a Java program manipulating another Java program, in which case the metalanguage and the object language are identical. We identify such a program as homogeneous meta-programming. Conversely, when the metalanguage differs from the object language, we identify such a program as heterogeneous meta-programming. In their survey of Metaprogramming Languages [25], Lilis and Savidis mention that metaprogramming is a concept present in literature for several decades, with numerous languages and with multiple models capable of serving as metalanguage.

As this thesis aims to detect coding flaws in students' source code, our program will have student Python programs as input, which is why we will use meta-programming.
More precisely, the student Python programs will be represented as Abstract Syntax Trees (ASTs). The previous section developed the theme of ASTs, and we will manipulate these Abstract Syntax Trees to detect defined patterns.

Logic Programming is a programming paradigm based on formal logic by building upon and extending one of the simplest yet most powerful logics, the logic of Horn Clauses [26]. The main implementation of such a language is Prolog (named after the French "PROgrammation LOGique").
Such a language has inherent search tree capabilities, including pattern matching through unification, backtracking and recursion, making it a good choice of technology to detect the convenient nodes in a tree. It is also a declarative programming language, allowing us to create easily readable queries.

For our thesis, we choose SWI-Prolog as the implementation of Prolog. We explain this choice for several reasons. Firstly, it is the Prolog version taught at the Louvain School of Engineering as part of the Programming Paradigms course instructed by Professor Kim Mens, allowing us to have a number of users in the university community of Louvain-la-Neuve who already have a basic knowledge of this language. Additionally, SWI-Prolog is a well-maintained implementation with an active community, ensuring continued support and development.

By representing indirectly the AST of our base programs as a set of logic propositions, we achieve logic meta-programming [27]. Logic meta-programming is not a new concept, and numerous examples of its application can be found in scientific literature [27] [28] [29] [30]. What these examples have in common is the conversion of input programs into a sequence of logical facts, enabling interrogation of this logical database to extract pertinent information [29]. Sometimes, for performance reasons and using a variant of Prolog such as SOUL [31], there is the use of "virtual" logic facts, which are actually a kind of rules that dynamically query the input programming languages.

The relationship between a program and its representation in terms of logic facts is generated by a program that parses the source code to produce the data to be translated into logical facts. However, the mapping scheme between the logic representation and base program may vary depending on the kind of information necessary for the analysis of the program. For example, in their work proposing a logic meta-system as a framework for aspect-oriented programming, De Volder and D'Hondt represent their program by having each class be represented by one logic fact [27]. Where Flederer et al. [29] needed to go deeper into the analysis of the source code for their project, and so they chose to transform each node of their program's AST into logical facts. Since in our work, we also want to detect good or bad practices in students' source code, we also need to represent the program at a fairly low level, and that's why we're going for the AST as well.

2.5 Program query languages

The aim of our work is to detect common coding flaws in student code. To do this, we are going to create new predicates in Prolog so that we can manipulate and mine information from the Abstract Syntax Trees of these programs. We will use Prolog as what is called in the literature a Program Query Language. Program Query Languages (PQLs) are programming languages or tools specifically designed for querying information from a program's source or binary code, abstract syntax

tree or other program representations. PQLs allow for expressing queries about code structures, execution paths, patterns, idioms, as well as potential flaws within program source code
In the previous sections of this chapter, we have already mentioned and presented some related work. In this section, we're going into more details and looking at some of the work that has already been done on program query languages and on Static Program Query Language in particular.

2.5.1 Program Query Languages

In the previous section, we stated that Arts' master thesis [24] mentioned XPath as a query language for XML. When source code is converted into XML format, XPath can be used as a query language. Additionally, variants of this language exist, such as XSLT, a transformation language converting XML documents into other formats, or XQuery, a query language for XML designed to perform more complex tasks than those achievable by using XPath.

Regular expressions, also known as RegExp, can be an effective tool for identifying patterns in strings. They are commonly utilized for pattern matching within text-based source code. However, their analysis capabilities may be limited when it comes to detecting more intricate code representations. This technique is one of the approaches explored in the experience later in this thesis.

Rascal[6] [32] [33] is a meta-programming language and analysis tool that focuses on developing domain-specific languages (DSLs). It provides a comprehensive framework for parsing, analyzing, transforming, and generating source code. With Rascal, users can create custom parsers and analysis tools to handle the syntax and semantics of multiple programming languages.

SOUL [31], an acronym for Smalltalk Open Unification Language, is a declarative logic programming language that serves as a query language for software analysis and manipulation. The language was designed to work seamlessly with the Smalltalk programming environment, and it leverages logic programming concepts to enable users to query and reason about the structure of Smalltalk programs. SOUL operates on a representation of the program abstract syntax tree (AST), offering users a high-level, declarative way to express queries regarding the program structure, behaviour, and patterns.

[6] rascal-mpl.org/

Pyttern is an academic research prototype designed at the Louvain School of Engineering to query Python programs for specific structural patterns in Python source code. It was created as an expressive wildcard language with a syntax that closely mirrors Python, with the aim of being easy to use for anyone who has already used Python. Its primary goal is to provide an intuitive query language that instructors can effortlessly utilize to define patterns and detect occurrences of common coding flaws in student code. We later use it in an experience of this master thesis.

Later in this thesis, we will compare our logic metaprogramming approach with some of the above-mentioned PQL.

2.5.2 Use of logic (meta) programming

Existing literature shows several projects using logic programming to query useful information from the source code [27] [29]. CodeQuest [30] uses DataLog [34], a database query language based on the logic programming paradigm, to query source code. JQuery [35] uses logic programming to explore Java code. However, these projects predominantly focus on "advanced" issues, such as verifying similar methods in classes and subclasses, the detection of design patterns or refactoring opportunities [29] which may not be encountered by CS1 students until one of the last chapter seen in the course. Our objective is to have a tool to detect some simple flaws typically occurring in exercises in introductory programming courses.

One project that also resembles our research technique is the implementation of a Prolog variant, referred to as ASTLOG, designed for efficient browsing of the AST [36]. Unfortunately, we have been unable to find access to this project, likely due to its publication date in 1997.
One of the main distinctions between this project and our thesis work lies in the approach taken to avoid importing the entire AST of a program as logical facts. The authors justify this decision by mentioning the performance penalty caused by the large Prolog Database representing large programs (10^5 - 10^6 lines). They developed their own version of Prolog, altering the functioning method to make the predicates applicable to external objects.
In contrast, this thesis focuses on analysing student programs encountered in CS1 courses. The programs are concise, rarely exceeding 50 lines, with longer ones often indicative of poor programming practices. The mentioned problem of a too large logic database is thus less relevant for our use case. We don't have the need to create a new language like ASTLOG, and we are rather developing a set of predicates using a logic programming language that is more accessible to other users, particularly to those with a fundamental understanding of logic programming.

Chapter 3

Solution

This chapter will explore our proposed solution for integrating logic programming with the search for good and bad practices in student source code.
We will begin by outlining our approach to develop the solution. Then, we will delve into the specifics of logic programming and its advantages for the task at hand. Finally, we will explain the design process and how we arrived at the proposed solution and present a simple example of what can be achieved.

In this chapter, we will not go into the implementation details, which will be covered in greater detail in the following chapter.

3.1 Approach

A way to detect bad practices in student code is through the use of static source code analysis. In the literature presented in Chapter 2, logic meta-programming is one of the techniques employed for this purpose. By converting the source code of a program into logical facts, we can create queries to detect desired practices (or their absence) in various student programs.
However, the tools mentioned earlier are generally intended for code written by programmers with some programming experience. As a result, common errors made by beginners may not be detected by these tools. Moreover, some of these tools are specially designed for object-oriented programming, but this paradigm is only presented in the later chapters of the CS1 course, making using these tools unnecessary for the earlier chapters.
We aim to develop a general-purpose tool allowing teachers and other users to add custom queries to perform their desired analysis. This was the reason that made us turn to logic programming and a declarative language, as this enables us to create a high-level declarative language that allows the identification of good or

bad practices in student source code.
As with Mens et al.[2], we have several student exam questions from previous years at our disposal. However, first, we need to develop a way to convert these code sources into data that could be processed by logic programming. The next step is to create a number of 'standard' queries, which would allow us to detect recurring bad practices and determine which functions our tool needs to manipulate the data in a way that makes it more readable and user-friendly.

3.2 Logic programming benefits

In the previous chapter on background material, we briefly introduced logic programming. This section will provide a more in-depth discussion of this programming paradigm and its interesting properties for its application in the context of this thesis.

In this paradigm, logic is used to reason about a knowledge base. A knowledge base consists of logic facts and rules.
The rules are written as follows:

```
A :- B.
```

and are read as A is true if B is true. Where A is called the head of the rule and B the body of the rule.
There exists a special form of rule called facts and written as follows :

```
A.
```

In this case, A is always true. In fact, it is equivalent to a rule :

```
A :- true.
```

Facts are the basis of the logic knowledge, it represents the data of a logic program while the rules represent the relationships between different logic elements.
To illustrate how logic programming works, we are going to present a simple example using the following statements :

- François is parent of Benoît.
- Jean is parent of François.
- A grandparent is the parent of a parent.

In logic programming, it can be represented by two facts and one rule:

```
1    parent(francois, benoit).
2    parent(jean, francois).
3    grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

Variables in Prolog are represented by identifiers beginning with upper case letters, while constants and predicate names begin with lower case letters.
We can now create queries to interrogate our knowledge base. For example, we can ask the following questions :

- Is Jean a grandparent of Benoît?
- Who is the parent of Benoît?
- Who is a grandchild?
- Is Jean a grandparent?

Such question can be represented in logic programming by the following queries (with their answer on the right):

```
1    ?- grandparent(jean, benoit).      true
2    ?- parent(X, benoit).              X = francois
3    ?- grandparent(_,Z).               Z = benoit
4    ?- grandparent(jean,_).            true
```

In this simple example, three major properties of logic programming are already present: **Multi-Directionality**, **Unification** and **Resolution**

Multi-directionality refers to the ability of a logic predicate to be used in multiple directions. Logic predicates describe mathematical relations and can be used in multiple directions. For instance, the grandparent rule can be used to determine if someone is a grandparent but also if someone is a grandchild, depending on which argument is left as a variable in the rule.

- `?- grandparent(jean, X).` asks who Jean's grandchild is.
- `?- grandparent(Y, benoit).` which is also correct, asks who is the grandparent of Benoît.

Unification is the mechanism of binding variables in a matching clause with the goal of making them identical. Unification is the property that allows the desired fact to be retrieved from a variable. For example :

- `?- parent(X, benoit).` returns as answer for X francois because `parent(francois, benoit)` is true.

- the _ symbol called the anonymous variable is used to indicate there is a value, but we do not care about it. In line 3, `?- grandparent(_,Z).` The question we try to answer is "Who is a grandchild?". We search for grandchildren, but each one of these grandchildren is bound to a grandparent, but we do not care about who are these grandchildren.

- It is not shown in the previous example, but a more complex structured term unifies with another term if it has the same number of arguments, which could be recursively unified.

  `?- foo(A, triplefoo(B, C, D) = foo(t1, triplefoo(t2, t3, max(2, 3)).`
  where `A = t1, B = t2, C = t3 and D = max(2, 3)`

Resolution is the concept that connects multiple clauses. When two clauses with complementary literals are resolved, they produce a new clause. If we apply this concept to a hypothetical scenario, it means that if we state that all turtles are green and that Billy is a turtle, we can conclude that Billy is green. In the example with the grandparents, it would be similar: if we know that Jean is Benoit's grandparent and that François is Benoit's parent, we can infer that a possible solution could be that Jean is François's parent. (This example is less straightforward than the one with the turtles because even though the reasoning technique works by connecting the statements, we could encounter a situation where we say that Camille is the parent of Benoît, but Camille is not necessarily the child of Jean.)

Two other properties which are not directly linked to the example but more to the language as a whole and are also very interesting are **backtracking** and **declarative style**.
Backtracking is a property linked to the search technique used by the program. When a particular path does not lead to a solution, the system returns to the last choice point, the last substitution, and proposes the next one as a possible answer.
Declarative style is how programs are written in logic programming. It is a paradigm that allows us to describe what we want to achieve or obtain without implementing the research or how to achieve it. In fact, we declare a mathematical property and leave it to the Prolog resolution mechanism to verify whether this property is valid (and for what bindings).

3.2.1 How do these benefits fit with our solution?

After discussing the specifics and benefits of logic programming, we will now explain how we can take advantage of it to develop our tool for detecting bad practices in student source code.

Firstly, one of the key strengths of logic programming is multi-directionality. It allows us to use the same predicate for different or even opposite purposes, thereby reducing the number of predicates required. In our use of logic programming, we have tried to adhere to this principle by designing predicates that can work multi-directionally whenever possible. An example of a use case that comes to mind is the ability to use the same predicate to traverse the AST from the root to the leaves in a top-bottom manner, as well as in the opposite direction from the leaves to the root in a bottom-top manner. This would improve efficiency by allowing the search to begin at specific nodes at the bottom of the tree, which can be challenging in other languages.

The properties of unification and resolution will enable us to create requests to detect the practices we seek to identify in student programs. Unification allows us to represent the nodes of the Abstract Syntax Tree we search with a variable and, during resolution, to match these variables with the desired nodes. In the same way, by utilizing the way resolution works, we can create a set of conditions that will combine to form a more complex query for detecting bad practices.

Finally, backtracking and the declarative style inherent in logic programming will allow us to formulate queries to search for bad practices, minimising the inclusion of unnecessary elements for the search and focusing solely on the elements we are looking for. As the search is conducted by the logic program using backtracking, we are not required to specify the mechanics of the search, only the expected result.

3.3 Design of the solution

In this section, we will discuss the design of the solution we built using logic programming to detect good and bad practices in student programs and how it was envisioned without delving into the implementation details.

3.3.1 Transforming a Python program into logic facts

The initial step in using logic programming to analyse a program source code is to convert it into a set of logic facts.
To detect bad practices in the code, it is essential to recover as much information as possible about the code structure. The Abstract Syntax Tree is a common data structure representing the code as described in chapter 2. This provides the right amount of information we need to analyse a program. In addition, the Python AST is easily retrievable. The core idea for the conversion is to transform every node of the AST into a logic fact. Then, using logic programming, we'll have a

simple and effective way of navigating through it.

We represent an Abstract Syntax Tree (AST) as logic facts by assigning a unique ID to each node. The links with its descendants and ancestors are maintained in the tree by storing the IDs of its parent and children inspired by methods found in literature[29].
This version has several advantages, such as keeping the size of the logic facts modest, it also makes it easier to navigate from the bottom to the top of the tree, providing a helpful way easily implemented in logic programming to traverse the tree.

In Chapter 4, you will see another type of logical fact implemented. This version is linked to the various node types present in the abstract syntax tree and is used to retrieve information quickly, functioning like a dictionary key. It acts as a layer between the user and the logic database storing the data of the tree, avoiding the need to understand and to be fully acquainted with the structure of the logic facts.

3.3.2 Implementing essential predicates to mine information in the AST

After a Python program has been converted into logic facts, to simplify the creation of the first requests to detect good or bad practices in students' programs, we need to create a language, a standard set of predicates that can be used to reason about these logical facts. This will allow users to create queries without having perfect knowledge of how these logic facts are represented internally at a low level.
When creating queries, we identified two main tasks that often need to be performed:

1. browsing between different nodes in the AST to make links between them
2. checking the attributes of a node in the AST

To accomplish this task, we have created some basic predicates. In this and subsequent sections, when defining predicates, you will encounter a notation that uses +, -, ? prefix symbols in front of variables called argument mode indicators. It gives information about the intended direction of the predicate as explained in the Table 3.1. It is only used for documentation purposes. These symbols are not present in the implementation of the predicates.

Prefix	Meaning
+	argument only expected as **input** (bound to a variable at call-time)
-	argument only expected as **output** (not bound to a variable at call-time)
?	argument used **either as input or output**

Table 3.1: Caption

For example, the predicate

$$foo(+A, ?B)$$

always expects that the variable A is specified by the user whereas the variable B can stay unspecified (the predicate will then return the name of all variables that succeed the predicate), or can also be bound by the user (the predicate will then return if the predicate succeeds or not).
In short, variable A must have been bound to a constant at some point before this predicate was called, whereas variable B may or may not have been bound to a constant.

If predicates are used in an unintended way, the result provided by the predicate is not guaranteed.
You can find a table with the description of the basic predicates implemented at 3.3.2

Predicate	Description
id_type(?Id, ?Type)	***Type*** is the AST node type of the AST node with identifier ***Id***
info(?Id, +Name, ?Info)	***Info*** is the information for the attribute ***Name*** of the node with the given ***Id***
info(+Type, ?Id, +Name, ?Info)	The same as above but guarantee that the node is of a certain type ***Type***.
children(-Children, +Id)	List of identifiers ***Children*** is the children of the node with identifier ***Id***
parent(?Parent_Id, ?Id)	Id ***Parent_Id*** is the parent of the node with id ***Id***
descendant(?Descendant_Id, ?Ancestor_Id)	Node ***Descendant_Id*** is a (direct or indirect) descendant of node ***Ancestor_Id***
descendant(?Descendant_Id, ?Ancestor_Id, ?Child_Level)	***Descendant_Id*** is a descendant of node ***Ancestor_Id*** with a depth level ***Child_Level*** (1 for a direct parent-child relationship, 2 if there is an intermediary, 3 if there are 2 intermediaries, etc.)

Table 3.2: Basic predicates and their description

More details on the functioning and implementation of these predicates will be provided in the next chapter on implementation.

3.3.3 Combining predicates to write new queries

Now that we've transformed a Python program into logic facts and created general predicates that allow us to manipulate, traverse or retrieve information via these logic facts, the next step is to create queries to detect code that does or does not adhere to certain practices.
To serve as an example for future users and to have a few generic queries already implemented, we have a list of ready-made queries that can be run directly on student code. These queries use a combination of the predicates presented above and some specific to logic programming.
One of the queries we created retrieves the function name whose implementation contains a list of at least three constants. This query can be used to detect parts of hard-coded answers in student exercises. It is typically an example of coding

flaws that we find in an exercise that asks to find n prime numbers. For example, this kind of structure can be found at the start of some students programs:

```
1     def premiers(n):
2         list = [2,3,5,7,11,13,17,19,23]
3         ...
```

We provide you with the implementation and an explanation of the query used to detect this coding flaw.

```
1  contains_hardcoded_list(-Name) :-
2      id_type(List_Id, list),
3      info(List_Id, elts, Children_List),
4      findall(Child,
5          (member(Child, Children_List), id_type(Child,
               constant)),
6          Constant_Children),
7      length(Constant_Children, N_Constant), N_Constant >=
           3,
8      descendant(List_Id, Function_Id),
9      id_type(Function_Id, functiondef),
10     info(Function_Id, name, Name).
```

The code does the following :

1. The first line is the call to the query where the variable Name will be unified with the name of a function that corresponds to our search

2. The second line is used to find the node in the AST representing a Python list

3. The third line gets the different "elements" of the list. It is important to note that in Python AST, the List node has two types of arguments: the *'elts'*, which holds the type of elements and the *'ctx'*, which has a value Store or Load, depending on whether the container is an assignment or not. In our case, since we are interested in elements, we will only retrieve *'elts'*. As you can see, to have an exact predicate, it is important to understand the Python AST.

4. From the fourth to the sixth line, we have a Prolog *findall*, which finds all elements that are constant in the Python list.

5. The seventh line gets the length and compares it to 3, ensuring we have enough elements. It is used like an IF in our code, if *N_Constant* is less than 3 then Prolog will backtrack.

6. The eighth, ninth and tenth lines are used to go up the tree and find in which function the problematic list was created and Assign the name of the Function to the variable Name.

The predicates we introduced earlier help in navigating and retrieving information from the Abstract Syntax Tree, while the predicates used in logic programming are powerful enough to carry out the necessary search operations.

Chapter 4

Implementation

While the previous chapter presented the approach and solution design, this chapter will focus on the implementation of the solution. We will also discuss more advanced choices that have been made, either to make the tool easier to use, improve performance, and stay close to the logic programming principles.

4.1 From Python to logic facts

We followed a few steps to convert a Python program to logic facts. Firstly, we needed to transform the Python code into its AST representation. This is achieved using Python's built-in *'ast'* module[1]. The outcome is a description of the code in an Abstract Syntax Tree (AST), which has not undergone any modification. Although certain data obtained in the tree might appear unnecessary, we have opted to retain all the information so that a user with a specific requirement can use it as his preference.
Next, the AST generated in Python must be transmitted to the logic language. To do this we use the ast2json library[2]. This module converts the AST into a JSON-compatible representation. Given that SWI-Prolog has a library to interpret JSON files[3], this allows us to make the link between the python program and the logic program. This procedure provides a readable version of a program's Python AST for SWI-Prolog. The next step is to convert this AST into a series of logic facts. You can see all the steps in the conversion of the Python AST to Logic facts in figure 4.1

To create the logic facts, the program starts from the tree's root and traverses it to

[1] docs.python.org/3.11/library/ast.html
[2] pypi.org/project/ast2json/
[3] www.swi-prolog.org/pldoc/man?section=jsonsupport

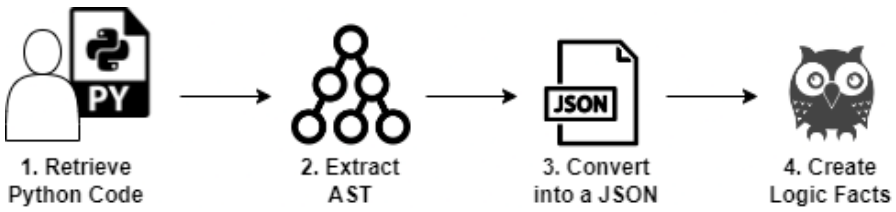

Figure 4.1: Steps in the conversion of the Python AST

generate the corresponding logic fact for each node. To keep track of the links of the tree, each node is assigned an ID, and each logic fact includes as arguments the ID of its parent node and, if it has any children, the IDs of those children. The consequence is that, while generating logic facts, we must explore all the children of a node before creating the fact that represents its parent. The tree is, therefore, traversed using a depth-first search algorithm. Once the end of the branch is reached, the algorithm goes back up the branch to generate the facts for the previous nodes.
While traversing the trees, our program generates two types of logic facts. The first one called *node_val* is the equivalent of a node of the AST and follows the following format inspired by those found in literature [29] :

$$node\_val(Type, Id, P\_Id, Children, Values)$$

Where :

- Type: The type of the node represented in the AST such as functiondef, assign, constant, name, for, ...
- Id: Each fact has an associated number, this is their id. This starts at 0 and progresses in ascending order. The node with the number 0 is the root of the program.
- P_Id: This is the ID of the node's direct parent. This makes it easy to navigate the tree representation. the root (id = 0) has a P_Id = -1
- Children: This is a list of the IDs of the node's children so that the tree can be traversed in the direction opposite to the P_Id.
- Values: These are the values and attributes associated with the node. For example, a node of type "Name" has an attribute "id" which contains usually the name of a variable.

The second type of logic facts called *node_def* is the format specification for the different types of nodes present in the AST. Its purpose is to ensure that all nodes

of the same type follow the same format when storing data in the various logic facts. Additionally, it allows retrieving a given attribute's value from all nodes of the same type. This will prove very useful later on when we create queries to simplify the task of verifying and retrieving information. These facts follow the format :

$$node\_def(Type, Children\_List, Values\_List)$$

Where :

- Type: The type of the node represented in the AST such as functiondef, assign, constant, name, for, ...
- Children_List: The list of the different attributes of the node of the given type that expect a child (points to another node)
- Values_List : The list of the different attributes of the node of the given type that expect a value (does not point to another node)

Each node in the abstract syntax tree has a specific set of attributes. For example, as shown in the figure 2.1, a node of type "Nam" has an "id" attribute linked to a value (a name), as well as a "ctx" attribute linked to another node of the AST. When an attribute is connected to another node, it is called a child. When it is linked to a term or value, we refer to it as a value. These are all the steps performed to transform a Python program into a series of logical facts. If we retake the example of figure 2.1, it is represented in Prolog via the logic facts presented in figure 4.2

```
node_def(module,[body,type_ignores],[])
node_def(assign,[value,targets,type_comment],[lineno,col_offset,
    end_col_offset,end_lineno])
node_def(binop,[op,left,right],[lineno,col_offset,end_col_offset,
    end_lineno])
node_def(add,[],[])
node_def(name,[ctx],[id,lineno,col_offset,end_col_offset,end_lineno])
node_def(load,[],[])
node_def(constant,[kind],[value,s,n,lineno,col_offset,end_col_offset,
    end_lineno])
node_def(null,[],[])
node_def(store,[],[])
node_def(expr,[value],[lineno,col_offset,end_col_offset,end_lineno])
node_def(call,[args,func,keywords],[lineno,col_offset,end_col_offset,
    end_lineno])

node_val(add,3,2,[],[])
node_val(load,5,4,[],[])
node_val(name,4,2,[5],[x,1,4,5,1])
node_val(null,7,6,[],[])
node_val(constant,6,2,[7],[1,1,1,1,8,9,1])
node_val(binop,2,1,[3,4,6],[1,4,9,1])
node_val(store,9,8,[],[])
node_val(name,8,1,[9],[x,1,0,1,1])
node_val(null,10,1,[],[])
node_val(assign,1,0,[2,[8],10],[1,0,9,1])
node_val(null,14,13,[],[])
node_val(constant,13,12,[14],[Hello World!,Hello World!,Hello World!,2,6,20,2])
node_val(load,16,15,[],[])
node_val(name,15,12,[16],[print,2,0,5,2])
node_val(call,12,11,[[13],15,[]],[2,0,21,2])
node_val(expr,11,0,[12],[2,0,21,2])
node_val(module,0,-1,[[1,11],[]],[])
```

Figure 4.2: Logic facts representation of 2.1

By taking only the *name* type facts we can see the link between the *node_def* and the *node_val* :

```
node_def(name,[ctx],[id,lineno,col_offset,end_col_offset,end_lineno])
```

```
node_val(name,4,2,[5],[x,1,4,5,1])
node_val(name,8,1,[9],[x,1,0,1,1])
node_val(name,15,12,[16],[print,2,0,5,2])
```

Here, we have three different *name* facts, the first with ID 4, the second with ID 8 and the third with ID 15. The definition of nodes of type *name* results in a single *'ctx'* node in the children_list. This means that for IDs 4, 8 and 15, respectively, they are connected to another node represented by IDs 5, 9 and 16. If we now look at the values for their 'end_col_offset' (UTF-8 byte offsets of the last tokens that generated the node), the definition shows that this is the 4th element in the value_list, and we, therefore, need to take the 4th element in the list for ID 4, 8 and 15, respectively. We can see that their value are 5, 1 and 5.
Note that while in their children_list, the numbers represent the IDs of the nodes to which the fact is linked, in the value_list, there are no nodes, and the numbers are, therefore, values assigned to the node.

We'd also like to draw your attention to the lists of children of *"assign"*, *"call"* and *"module"* nodes in the example presented in table 4.2. If we take the *"module"* example, you can see that the list of the children contains other lists: `[[1,11],[]]`. The definition gives us two attributes: `[body, type_ignores]`. So how does it work? Note that a node attribute can refer to a list of multiple children. For instance, in this case, the *"body"* argument refers to a list of two nodes. Likewise, the "type_ignores" attribute refers to an empty list. This is an example of a complexity of Python AST. For the node with id 12 of type "call", which refers to the call of a function, it has three child attributes: [args, func, keywords]. The list of children is made up of sub-lists: `[[13], 15, []]`. The first attribute, *'args'*, refers to the first element, the list `[13]`. In fact, *'args'* refers to the function's arguments. Since it is possible to have several arguments for the function, the attribute can point to a list of several elements as well as an empty list. The second attribute, *'func'*, points to the function called, which logically only points to a single element. As for the keywords, none were used in the example, so the attribute points to an empty list.

4.2 How to use the logic facts

As previously said in Chapter 3, when creating queries, two main tasks often need to be performed :

1. browsing between different logic facts to make links between them
2. checking the attributes of a logic fact

This section will present the various generic predicates we have implemented, starting with the latter.
We will continue to use the prefix notation for the arguments of the different predicates seen in the table 3.1

4.2.1 Checking the attributes of a logic fact

We have previously shown you the representation of the nodes of the AST by our logic facts :

$$node\_val(Type, Id, P\_Id, Children, Values)$$

Although the representation is relatively short, it is not necessarily easy to manipulate such primitive facts in a readable way when creating queries. What's more, for the last two attributes (Children and Values), depending on the type of the logic fact, these attributes have a different representation with different values, different categories and not the same length. A node representing the definition of a function does not have the same attributes that the one representing a variable name. So, to find the information you want in these two, you need to know how they are represented. In this section, we will show which predicates can be defined to make the manipulation of these logic facts more straightforward without having to know their low-level representation. They act as a layer of abstraction between the low-level representation and the higher-level use.

One important thing to note is that we want to limit (if not eliminate) the need for using a complete logic fact as argument to a predicate. Instead, since we have a unique ID assigned to each node in the AST, this will be used to identify which logic fact it represents. Therefore, the objective is to obtain all the relevant information from a node's ID and letting generic predicates do the recurring tasks.

As our first predicate, we link the ID with the type of fact. This gives us the following predicate:

$$id\_type(?Id, ?Type)$$

This predicate simplifies the manipulation of logic facts by avoiding the need to manipulate the entire representation. It is implemented as follows:

```
1     id_type(Id, Type) :-
2         node_val(Type, Id, _, _, _).
```

This is a simple predicate used either to identify IDs with a specific type or to check that the ID matches the desired type.

The second predicate, which will now be presented, is used to retrieve information related to an ID. In the logic representation of an AST node, the fourth and fifth elements assign data linked to the node, respectively its children (linked nodes descending from it) or the values of the node itself (a node of Type 'name' has an 'id' attribute which records the name of a variable, for example). This is where the second type of logical fact we have created becomes relevant.

$$node\_def(Type, Children\_List, Values\_List)$$

In order to access a specific attribute, the user would need to know where that attribute is stored, which is generally not the case since we want users to be able to use the tool without necessarily knowing the representation of the logic facts. This makes it difficult to access the right place in the logical fact.
To solve this issue, we have created a predicate that allows users to access attributes using only the name of the desired attribute. This works by using 'node_def' facts, which record the names of attributes and their location in the logic facts. Once you know where to look, you can go straight to the desired information. This is implemented via the following predicates.

$$info(?Id, +Name, ?Info)$$

$$info(+Type, ?Id, +Name, ?Info)$$

The first retrieves the node type and sends the information to the second. The second will find out where it needs to retrieve the information via 'node_def' and then obtain this information.
Using these predicates, we can retrieve all the information belonging to a node without the user having to manipulate the logic fact structure.
Here is their implementation :

```
1  info(Id, Name, Info) :-
2      node_val(Type, Id, _, _, _),
3      info(Type, Id, Name, Info).
4
5  info(Type, Id, Name, Info) :-
6      node_def(Type, Children_Keys, _),
7      nth0(Index, Children_Keys, Name), !,
8      node_val(Type, Id, _, Children, _),
9      nth0(Index, Children, Info).
10
11 info(Type, Id, Name, Info) :-
12     node_def(Type, _, Value_Keys),
13     nth0(Index, Value_Keys, Name), !,
```

```
14      node_val(Type, Id, _, _, Values),
15      nth0(Index, Values, Info).
```

We take advantage that in logic programming, there can be multiple implementations of the same predicate. When searching for a solution, the first predicate will be used first. If it doesn't provide a solution, the following implementation will be used, and so on. This feature makes it possible to create conditions, for instance, in this example, we have two implementations that look whether the desired information is part of the node's children or values.

4.2.2 Navigating between different logic facts

After accessing the information held in a node, we will learn about the predicates used to navigate between the nodes in this section.
To begin with, two basic predicates are used:

$$children(-Children, +Id)$$

$$parent(?Parent\_Id, ?Id)$$

```
1       children(Children, Id) :-
2           node_val(_, Id, _, Children, _).
3
4       parent(Parent_Id, Id) :-
5           node_val(_, Id, Parent_Id, _, _).
```

The first predicate retrieves all the direct children of a node, while the second retrieves the node's parent.
These two predicates form the basis for navigating between logic facts. However, they only allow direct links between nodes. For instance, if we want to link node A to node C, which are only connected through an intermediary node B, we need to use a combination of the above predicates.
The problem is that we need to be able to make simple links between nodes connected through intermediaries, and above all, we may not know or may not necessarily care about the number and the type of intermediaries that exist between two nodes. We require a new predicate to address this problem and simplify the navigation between the logic facts.

$$descendant(?Descendant\_Id, ?Ancestor\_Id)$$

$$descendant(?Descendant\_Id, ?Ancestor\_Id, ?Child\_Level)$$

The 'descendant' predicate links two IDs connected by any number of intermediaries (0, 1 or more, with Descendant_Id different from Ancestor_Id). With :

- Descendant_Id: the ID of the node which is the descendant of the other node
- Ancestor_Id: the identifier of the node which is the ancestor of the other node
- Child_Level: which is the level of depth at which the descendant should be searched for in relation to the ancestor (1 for a direct parent-child relationship, 2 if there is an intermediary, 3 if there are 2 intermediaries, etc.)

For instance, if we consider the A-B-C example mentioned earlier, descendant(C, A, 2) would be successful if A is C's "grandparent".

You may observe that this predicate can be used in any direction: it can be utilized to move up or down in the AST. This is one of the primary advantages of using the logic representation of the AST. Unlike other techniques, it enables us to directly search for the desired information at the bottom of the tree before going back up, instead of having to search through the entire tree to verify whether the desired information is present.
For the user, there is only one predicate (two if you consider the difference with-/without the Child_Level variable, but the 'without' calls on the 'with', they share the same implementation). However, there are several different implementations that allow the predicate to be performed in an optimised way depending on which variables are expected as input/output. For example, if the user only specifies the ancestor, the search for the descendant will be carried out using the 'children' predicate, whereas if the predicate receives the descendant as input, then the search will be carried out using the 'parent' function.

4.2.3 Example of the creation of a query

In the introduction of this master thesis, we provided an example of a programming exam question. Despite a minor mistake, one student received significantly fewer points than another student who had a major coding flaw in its code by partially hardcoding the answers. The latter student still received half the points for the question.

The expected "standard" solution was as follows:

```
1       sum = 0
2       for x in range(i+1):
3           sum += (-1)**x/((2*x)+1)
4       sum = 4*sum
5       return sum
```

This exercise presents a common pattern in introductory computer science questions where a variable accumulates the result of the terms of a series in a loop. This pattern is called an accumulator. By identifying programs built around an accumulator, we can detect the opposite, those that lack this structure and may contain bad practices.

We define the accumulator pattern we're looking for as follows:

- a variable gets assigned to a constant
- After this assignment, this variable needs to be updated in a loop
- The function returns the variable

A simple way to implement this query would be to search for each part of the pattern in order. It gives the following implementation :

```
1  accumulator() :-
2      id_type(Start_Assign_Id, 'assign'),
3      info(Start_Assign_Id, 'targets', List_Targets),
4      member(Target_Id, Targets),
5      id_type(Target_Id, 'name'), info(Target_Id, 'id',
           Variable_Name),
6      info(Start_Assign_Id, 'value', Right_Assign_Id),
7      id_type(Right_Assign_Id, 'constant'),
8
9      id_type(Incrementation_Id, 'augassign'),
10     info(Incrementation_Id, 'target', Inc_Target)
11     info(Inc_Target, 'id', Variable_Name),
12     descendant(Incrementation_Id, Loop_Id),
13     (id_type(Loop_Id, 'for'); id_type(Loop_Id, 'while'))
           ,
14
15     id_type(Return_Id, 'return'),
16     info(Return_Id, 'value', Value_Id),
17     id_type(Value_Id, 'name'),
18     info(Value_Id, 'id', Variable_Name)
19
20     sort([Start_Assign_Id, Incrementation_Id, Return_Id
           ],
21         [Start_Assign_Id, Incrementation_Id, Return_Id])
               .
```

.
With

1. Line 2: We retrieve an assign statement
2. Lines 3-5: We retrieve the variable name targeted in the "assign" statement (the variable A in the statement A = B).
3. Lines 6-7: We check that the value of the "assign" statement is a constant (B is a constant in the statement A = B).
4. Lines 9-11: We retrieve an "augassign" statement (ex: C += D), where the target (C) is the variable name found earlier in the 'assign" statement (A should be the same variable name as C).
5. Lines 12-13: We check that the "augassign" statement is a descendant of a loop node of the AST, meaning it is contained in the body of the loop.
6. Lines 15-18: We retrieve a "return" statement and check that the variable returned has the same name as the variable we identified previously.
7. Lines 20-21: We check that we retrieve the order we expected (first the assign and last the return statement).

This request might be complex to comprehend as it includes multiple tasks that need to be connected with low-level predicates. If in addition, the user is unfamiliar with Python AST structure, understanding or replicating the query may be challenging as the structure of the query is close to the Python syntax tree.

4.2.4 Generality versus Readability (and ease of use)

Just before, we showed you an initial query using the general predicates we've implemented. You may have had trouble understanding the query at first glance because its structure is very complex and similar to the structure of the Python abstract syntax tree.
Our basic predicates have been designed to be as general and versatile as possible, and they can work with any type of node in the AST. However, this generality comes with a downside. These predicates may not be the easiest to read or use as they are too general, and simple manipulations become rapidly complex.
To simplify the query writing process, we need to provide a set of predicates that act as a layer on top of these basic predicates. These additional predicates will provide more expressive or specific options for making queries.

The first sort of predicate we've implemented is one that links an ID to its type.

```
ast_functiondef(Id) :- id_type(Id, 'functiondef').

ast_list(Id) :- id_type(Id, 'list').

ast_assign(Id) :- id_type(Id, 'assign').
```

As demonstrated in the examples above, the implementation of these predicates is straightforward - they use the *id_type* predicate to create a more declarative version of the code that is easier for users to read. These predicates only offer syntactic sugar and do not alter the functionality of the code.

The second kind of predicate is used to retrieve information that is needed on a regular basis.

```
ast_functiondef_name(Id, Name) :-
    info('functiondef', Id, 'name', Name).

ast_list_elements(Id, Elements) :-
    info('list', Id, 'elts', Elements).

ast_assign_left(Id, Left) :-
    info('assign', Id, 'targets', Targets),
    member(Target, Targets).
```

As demonstrated in the examples above, these predicates are usually a replacement of the *info* predicate with the aim to reduce the use of certain constants, thus avoiding recurring errors and being more declarative in queries. In addition, the use of such a predicate can eliminate the need of simple *id_type* predicate in the queries as the matching between the Id and the nodes are made via the info predicate, thus reducing and simplifying the creation of queries.

The third predicate variety implements more specific cases encountered while creating queries. Unlike the first two kinds of predicates, intended to be syntactic sugar, this third type responds to more precise search needs when creating queries.

```
maybe_descendant(Id, Id).
maybe_descendant(D_Id, A_Id) :- descendant(D_Id, A_Id).

ast_loop(Id) :- ast_while(Id).
ast_loop(Id) :- ast_for(Id).
```

Given the three sorts of predicates, the example given in the previous section can be rewritten as follows :

```
1  accumulator() :-
2      ast_assign_left_name(Start_Assign_Id, Variable_Name)
           ,
3      ast_assign_right(Start_Assign_Id, Right_Assign_Id),
4      ast_constant(Right_Assign_Id),
5
6      ast_augassign_left_name(Incrementation_Id,
           Variable_Name),
7      descendant(Incrementation_Id, Loop_Id),
8      ast_loop(Loop_Id),
9
10     ast_return_value(Return_Id, Return_Value_Id),
11     ast_name_id(Return_Name_Id, Variable_Name),
12
13     order(Start_Assign_Id, Incrementation_Id, Return_Id)
           .
```

We have used specific predicates in our code to improve its readability and make it easier for new users to create queries. The generic predicates we have presented first serve as the foundation of our tool and can be used to create simple queries. However, additional predicates are (and, according to the need of the users, more still need to be) introduced to the language to enhance the declarativity of the queries and make them easier to use.

Chapter 5

Detecting coding flaws

In this chapter, we will discuss how to use our logic tool to identify coding flaws. We will provide examples to help you better understand the process. However, we will first revisit the example from the previous chapter to explain why detecting certain flaws can be complicated. This complexity arises due to the multiple possible implementations and the uniqueness of the answers provided by each student. These special cases sometimes require complex queries to detect flaws for all students accurately.

5.1 The complexity of writing accurate queries

In the introduction, we provided an example of an exam question given to students. In the following chapters, we presented an implementation of a query that could detect the presence or absence of an expected pattern for solving this question. This pattern was called an accumulator. Some may have noticed that even though the query identifies a type of accumulator, it is still possible to write a different implementation in Python that uses a variable as an accumulator but won't be detected by the above query. This section concerns a problem we encountered when testing our first queries on student source code. We realized that we hadn't anticipated all possible implementations.
Let's take a step back and look at the definition we gave of the pattern we were looking for :

- A variable gets assigned to a constant
- After this assignment, this variable needs to be updated in a loop
- The function returns the variable

All possible implementations must be considered to detect all occurrences of this pattern. For example, we have already considered that the loop could be a 'for' loop or a 'while' loop and that we need to detect those two to ensure the proper functioning of the query.
Some people may also have noticed that we only detect updates to the value of a variable when an "augassign" (A += B) is used for the accumulation. However, it is also possible to update the value of a variable using a traditional "assign" (A = A + B) for accumulation purposes. A variant of the expected "standard" solution is as follows:

```
1       sum = 0
2       for x in range(i+1):
3           sum = sum + (-1)**x/((2*x)+1)
4       sum = 4*sum
5       return sum
```

Our previous query would detect such an answer as missing an accumulator. One could argue that our implementation was poor in this particular case as they failed to anticipate a common variant and did not anticipate that the pattern could have been implemented in multiple ways.

But sometimes, the originality of the students' responses means that they use rarely used techniques that do not necessarily come to mind. One of such example is the multiple assign statement on a unique line. In Python, the following variant of the expected "standard" answer does not generate any problem.

```
1       sum = 0
2       x = 0
3       while x < i+1:
4           sum, x = sum + (-1)**x/((2*x)+1), x+1
5       sum = 4*sum
6       return sum
```

While analysing student source codes with our query, we discovered a code that used an equivalent of line 4. We expected to find only one assignment on the same line. Therefore, we needed to modify our implementation to adapt to these specific cases and detect this pattern accurately.

But sometimes, the very definition of the pattern can be seen as inaccurate. Our query has been implemented on the principle that the accumulated variable gives the result and is, therefore, returned.

```
1       sum = 0
2       for x in range(i+1):
```

```
3           sum += (-1)**x/((2*x)+1)
4       return 4*sum
```

The code is correct, but because we only detect when the variable is returned in our query, we won't detect the accumulator in this code. Therefore, we need to allow the 'return' statement to return a more complicated statement than just the variable possibly.

Yes, but what about this?

```
1       sum = 0
2       for x in range(i+1):
3           sum += (-1)**x/((2*x)+1)
4       answer = 4*sum
5       return answer
```

This code is also correct, and it may be considered as containing an accumulator and even with the fix discussed just above, our implementation would not detect it. To be able to detect the accumulator in this code, we would need to rethink the design we had in mind for an accumulator and adapt the implementation of the query to it.

In this section, we wanted by simple examples highlight the challenge of designing complex queries that can accurately identify the presence or absence of a certain practice. The creator of such queries needs to anticipate the various implementation possibilities that students may use, which can be quite numerous and unexpected.

5.2 Optimisation

In the previous sections, we implemented a query to detect the presence of an accumulator in a Python program. We followed the order of a Python program by first detecting a variable in an 'assign' statement, then whether this variable is updated in a loop, and finally, whether this variable is returned. However, just like searching in an SQL database, the choice of the first element to be detected influences the performance of the search. By searching for a 'more infrequent' element, you limit the number of branches following the first choice, thus limiting the search quantity.

If we want to optimize our search for an accumulator by limiting the options at the time of the first choice, assuming that we stay with the initial definition of the accumulator we had made, we should start by looking at which variables are returned. This approach should produce less starting branch possibilities than looking for all the variables assigned to a constant at any time in the code. This

is when the possibility of efficiently visiting the AST from top to bottom as well as from bottom to top becomes very practical because it allows us to rethink the order in which we carry out the search in order to optimise it.

5.3 A range of basic queries

In this section, we will show the initial examples of queries we have developed, provide an explanation of our approach, and highlight some of the comments and opportunities we have encountered during the creation process. We based our work on examples of poor coding practices found in student code and examples of desirable properties to be identified in Eric Arts' master thesis [24].

Hardcoded List

We start with a coding flaw we already discussed in Chapter 3. The objective is to identify codes that contain a list with at least N constants. To achieve this, we have implemented the following query:

```
1 contains_hardcoded_list(-Name, +N_Constants) :-
2     ast_list_elements(List_Id, Elements),
3     findall(Constant_Element,
4             (member(Constant_Element, Elements),
                ast_constant(Constant_Element)),
5             Constant_Elements),
6     length(Constant_Elements, N_Constant_Elements),
        N_Constant_Elements >= N_Constants,
7
8     descendant(List_Id, Function_Id),
9     ast_functiondef_name(Function_Id, Name).
```

It operates as follows:

1. Line 1: The head of the query. *Name* will be linked to the function name containing the flaw detected. *N_Constants* is the minimum number of Constants we need in the list to detect the flaw. For example, for *N_Constants* = 3, we will only detect lists containing at least 3 constants.

2. Line 2: Find a list and retrieve all its *Elements*.

3. Line 3-5: Search for all constants in the *elements* of the list.

4. Line 6: Verify there is at least *N_Elements* that are constants

5. Line 8-9: Go back up the tree to find the name of the function in which the problematic list is.

No Return Statements

A simple query to find out which functions have no 'return' statements.

```
1 function_without_return(-Name) :-
2     ast_functiondef_name(Function_Id, Name),
3     findall(Return_Id,
4             (descendant(Return_Id, Function_Id),
                ast_return(Return_Id)),
5             []).
```

It operates as follows:

1. Line 2: Find a function and its *Name*
2. Line 3-5: We use a 'findall' to retrieve all the 'return' statements belonging to this function. But by matching the result of the 'findall' with an empty list, it makes it possible to find only a function that has no 'return' statement.

The 'findall' predicate is particularly useful when creating queries because it makes it easy to find all occurrences of a property. Often, when we need to count the number of times an element is present (whether it is several times or 0), we can use the 'findall' predicate.

Multiple If

A common issue found in students' code when attempting to hardcode an answer is the excessive use of *if statements*. Previous examples have shown that it is easy to count the number of elements in a program.

```
1 functions_with_if(-Name, +N_Ifs) :-
2     ast_functiondef_name(Function_Id, Name),
3     findall(If_Id, (descendant(If_Id, Function_Id),
        ast_if(If_Id)), Ifs),
4     length(Ifs, Length_IFs), Length_IFs >= N_Ifs.
```

As in the previous examples, we count the number of *if statements* in a function and compare it to the number given as argument to the predicate.

Unexpected Constants and Magic Number

To detect flaws in students' source code, we can also examine the constants they use. Typically, students should only employ a small set of predictable constants when solving basic exercises. The query below lets us identify instances where an unexpected constant is used.

```
1 unexpected_constants(?Name, +Constants, -
     Unexpected_Constant) :-
2      ast_constant_value(Constant_Id, Unexpected_Constant)
          ,
3      not(member(Unexpected_Constant, Constants)),
4      descendant(Constant_Id, Function_Id),
5      ast_functiondef_name(Function_Id, Name).
```

It operates as follows :

1. Line 1: The head of the query. *Name* will be linked to the function name containing the flaw detected. *Constants* is a list of expected constants written by the user as input. *Unexpected_Constant* is, as its name suggests, the value of a constant used in the function and which was not expected.

2. Line 2: Find a constant and retrieve its value

3. Line 3: Check if it is not a member of the expected constant list.

4. Line 4-5: Go back up the tree to find the name of the function in which the unexpected constant is.

This query has the particularity of having '?' as the prefix for the *Name* argument of the query. Unlike the other queries, we have tested it to ensure it also works in this direction. It allows us to create a 'magic number' query. The following query detects functions that use more than N unexpected constants.

```
1 magic_number(-Name, +N_Magic_Numbers, +Constants) :-
2      ast_functiondef_name(_, Name),
3      setof(Unexpected_Constant,
4            unexpected_constants(Name, Constants,
                Unexpected_Constant),
5            Unexpected_Constants),
6      length(Unexpected_Constants, N_Values), N_Values >=
          N_Magic_Numbers.
```

It operates as follows :

1. Line 1: The head of the query. *Name* will be linked to the function name containing the flaw detected. *Constants* is a list of expected constants written by the user as input. *N_Magic_Numbers* is the number of unexpected constants the function needs to contain at least to be detected as with a flaw.

2. Line 2: Find a function

3. Line 3-5: Find all the unexpected constants present in the function. The 'setof' predicate is an equivalent of 'findall' that sort the results and drops the duplicates. This ensures that the same constant is not counted several times.

4. Line 6: Verify there is at least *N_Magic_Numbers* unexpected constant.

Nested If

Earlier, we developed a query to identify instances with too many if statements within a function. However, another potential issue related to if statements is the presence of nested if statements. A nested if statement is defined as an if statement located within the body of a previous if statement, resulting in code with conditions nested within other conditions.

```
1  functions_with_nested_if(Name, N_Nested_Ifs) :-
2      ast_functiondef_name(Function_Id, Name),
3      ast_if(If_Id),
4      descendant(If_Id, Function_Id),
5      New_N is N_Nested_Ifs - 1,
6      nested_ifs_count(If_Id, New_N).
7
8  nested_ifs_count(_, 0) :- true, !.
9
10 nested_ifs_count(If_Id, N_Nested_Ifs) :-
11     ast_if(New_If_Id),
12     ast_if_ifbody(If_Id, Body_If),
13     member(Member_Body_If, Body_If),
14     descendant(New_If_Id, Member_Body_If),
15     New_N is N_Nested_Ifs - 1,
16     nested_ifs_count(New_If_Id, New_N).
```

This query is more complicated than the preceding ones. It is separated into two main predicates :

1. functions_with_nested_if is the traditional query that searches for the name of the functions containing the flaw

2. nested_ifs_count is used as a loop to count the number of nested_if in the function

The query operates as follows :

1. Line 1: The head of the query. *Name* will be linked to the function name containing the flaw detected. *N_Nested_Ifs* is the number of nested if statements we search.
2. Line 2-5: We start the search process with a function and an if statement from within that function.
3. Line 6: Call the loop.
4. Line 8: End case, it has enough nested if to be detected as a flaw.
5. Line 10: Loop predicate. *If_Id* is the current if statement. *N_Nested_Ifs* is the number of nested if statements we still search.
6. Line 11: Take a new if statement.
7. Line 12-14: Verify if the new if statement is a node descending from the body of the previous if statement.
8. Line 15-16: Launch the next loop.

In this query, we might need to count elements, but we can't use the 'findall' predicate as we search for a structure in the AST that follows a specific order with well-defined branches.

Statements after return

In this query, we will search if there is dead code: statements unreachable after a return statement.

```
1 statements_after_return(Name) :-
2     findall(Return_Id, ast_return(Return_Id), Return_Ids
        ),
3     findall(Return_Id,
4             (   ast_return(Return_Id),
5                 start_indentation(Indentation_Id,
                    Return_Id),
6                 attributes_indentation(List_Attributes),
7                 member(Attr, List_Attributes),
8                 info(Indentation_Id, Attr, Indentation),
```

```
 9                   last(Indentation, Return_Id)),
10               Last_Return_Ids),
11       subtract(Return_Ids, Last_Return_Ids,
            Return_Ids_Without_Last),
12       member(Return_Id, Return_Ids_Without_Last),
13       descendant(Return_Id, Function_Id),
14       ast_functiondef_name(Function_Id, Name).
```

This query is divided into 3 steps:

1. Line 2: Find all the return statements
2. Line 3-10: Find all the return statements that are the last statements of the indentation they are on (no dead code)
3. Line 11-14: Take the difference between the two first steps: it is the return statements followed by dead code.

As for the previous query, in this one, we need to look for a structure in the AST, which makes the queries more complex.

What can we learn from basic queries?

We discussed a variety of basic queries. While straightforward queries such as 'no return statements' and 'multiple if' that fall under a category that we can name 'identifying and counting node type' queries are relatively easy to implement, more complex queries that require identifying a group of nodes following a specific structure such as the 'nested if' query or even the example of this chapter the 'accumulator' query, are more challenging to express in our language. Arts [24] has already recognized this issue: navigating in the AST is quite complex.

Chapter 6

Validation

In this chapter, we will discuss the different techniques we used to validate our tool, which uses meta-logic programming to detect good or bad practices in student programs.
We will cover the development of tests in the project, their benefits and limitations. Additionally, we will analyse the result of some queries on a set of student answers to exam questions. This analysis aims to demonstrate the contribution and effectiveness of our tool as a correction aid.
Finally, we will present an experiment taken by some students who were tasked to create various queries in order to detect bad practices in simple programs. This experiment will provide feedback on the tool's usability and a comparison with other tools that can be used to detect bad practices.

6.1 Testing

To ensure the validity of our tool, we need to conduct tests to identify and fix any errors in our implementation. We have established two types of tests for this purpose: tests on predicates and tests on queries.

Predicate testing consists of unit tests that use specific examples to check that the predicate behaves as intended.
In a previous chapter, we introduced predicates and the attached prefix symbols(+, -, ?) (see 3.1). During the tests, we check all possible combinations according to the prefixes to ensure that the predicates function correctly in each case. On the other hand, when a variable prefix is designated by a +, for example, this may mean that we have not tested the case - for this same variable, as this could lead to complications or that the predicate was not intended to be used in this way, and therefore has not been tested.

To test the queries, we created a Python file for each query, based on examples found in the students' programs, containing a certain number of functions with and without the coding flaws we want to detect. This allows us to test if our queries are working, generally by identifying which of the implemented functions are detected by the query. This system of testing with multiple functions in a single Python file explains why our queries usually end by finding the name of the function to which the coding flaw detected belongs. As our tool loads the AST from one program at a time, we needed multiple functions on the same file to test multiple cases easily and efficiently. However, tracing back to the problematic function at the end of the query is not very useful for student programs, as they generally consist of a single function. It is, therefore, mainly used to facilitate testing.
But, while this system allows us to test our queries, it only provides partial testing. As stated by Eric Arts in his master thesis [24]: Even "simple" properties are often a lot more complicated than they may look, as there are usually many edge cases to consider. To verify that the queries created cover all these edge cases, we need to create a test suite adapted to all these scenarios. The challenge is, therefore, the complexity of finding all these edge cases.

6.2 Experiment

As part of the Programming Paradigms course taught by Professor Kim Mens at the École Polytechnique de Louvain, students were initially introduced to logic programming. This provided them with a foundation knowledge to test our tool. Later in this course, they received an overview of program query languages, followed by a more detailed description and manual of selected program query languages. Subsequently, the students participated in an experiment using a program query language to identify coding flaws in Python source code. The goal was to familiarize them with these languages and to gather their feedback on using some of them. Among these languages there was our project using metalogic programming.

6.2.1 Method

Four different languages and techniques for detecting specific patterns in programs were selected for the experiment. These were :

1. Regular expressions by using the *'re'* built-in module in Python.
2. The Python AST traversal by using Visitor Design Pattern and more specifically, the *'NodeVisitor'* class brought by Python *'ast'* module.

3. Pyttern, a project of the Computer Engineering department at UCLouvail. It is a Python library designed to create patterns to match Python code.

4. Prolog and logic metaprogramming (our project).

The aim of the experiment was for the students to implement queries to detect a list of coding flaws :

- If instead of While: Code fragment without a while loop containing an If where instead there should have been a While statement.
- Erase argument value: Code fragments that accidentally or purposefully erase an argument variable by reassigning it to a constant.
- Loop variable not used: When a code fragment uses a loop with a loop variable not used inside that loop.
- Missplaced return: Function without a return-statement at top-level in its body, but does have one at a deeper nesting-level such as inside an If-statement or a loop.
- Too much indentation: Program that have 3 or more nesting levels by overusing if or loop statement.
- Print instead of return: Program without a return statement but with a call to the print function.

To achieve this, the experiment was divided into two two-hour sessions, during which each student was given the opportunity to work with one of the four languages. The assignment of languages and the results given by the students were done anonymously.
After each session, every student was required to complete a questionnaire to give feedback. The questions were designed to allow students to provide feedback on the ease of use of the programming language. They were asked to give their opinion, ranging from "strongly disagree" to "strongly agree," on statements such as: "I found this programming language easy to use" and "I believe most people would learn to use this programming language very quickly."

6.2.2 Results and Discussion

We had sixteen students who participated in the experiment and among them eight used our tool. As anticipated, their feedback was not really positive, highlighting issues we raised during this master thesis. You can find a detailed result of their feedback for our language in table 6.1. For a comparison of the feedback of the

different languages used in the experiment see table 6.2. In the two tables we have taken an average of their feedback where 0 is 'strongly disagree' and 4 is 'strongly agree'.

One of the main differences compared to the feedback given to the other languages tested in this experiment is that the students believe that other users will need to learn a lot of things before being able to use the language. To get the most out of using our tool, you need to have some knowledge of logic programming, and the students had a basic knowledge thanks to the introduction they had at the beginning of the course. However, using this language to create new queries would be very complex for other users without this knowledge. Additionally, you must understand how the Python Abstract Syntax Tree works as the search is carried out at the AST level. Since it is a structure that is not handled or inspected every day, it can be complicated to search for coding flaws in this structure.

Similarly, feedback on the complexity of use and the confidence these students had in using the tool was not really positive. This feeling from the students was confirmed when we tested their implemented queries. In general, they wrote simple queries that detected basic cases. This can be explained by the fact that it seemed difficult for the students to test the queries during the experiment. The codes provided were numerous and without annotations, so the participants in the experiment had to find out for themselves which codes had which flaws. One student created simple Python programs containing flaws himself, and this student was one of those who succeeded best in the experiment. Generally speaking, the students' queries are based on good ideas, but the difficulty of using the tool and understanding AST Python makes it difficult to write correct queries.

We were surprised to find that, among the four languages used in the experiment, the students saw our language as the least declarative. Our goal in using logic programming was to create a declarative tool for identifying poor coding practices in student programs. However, we acknowledge that the complexity of navigating the Abstract Syntax Tree (AST) and our approach of closely following the tree structure can make specific tasks in our tool less declarative.

Regarding the positive results, despite receiving mixed student reviews, our language was ranked as one of the most powerful with the least risk of unexpected responses. The students also observed that the different features of our language are well integrated. When creating queries, we noticed that some of them took advantage of the potential of the different predicates to match an AST node and, from this node, extract information about the tree via other predicates.

Question	Mean feedback (0-4)
I think that I would like to use this program query language frequently	1.375
I found this program query language unnecessarily complex	2.125
I thought this program query language was easy to use	1.75
I think that I would need the support of a technical person to be able to use this program query language	2.375
I found the various features of this program query language were well integrated	2.375
I thought there was too much inconsistency in this this program query language	1.625
I would imagine that most people would learn to use this program query language very quickly	0.625
I found this program query language very cumbersome to use	2.625
I felt very confident using this program query language	1.25
I needed to learn a lot of things before I could get going with this program query language	3.375
Writing queries for coding flaws in this program query language reports many false positives (incorrect responses)	1.375
Writing queries for coding flaws in this program query language yields a lot of false negatives (missed responses)	1.5
Executing queries in this program query language was sufficiently fast for the task that was assigned to us	2.125
When writing a query that didn't work, was it easy to understand how to debug your query	1.25
This program query language is sufficiently powerful to allow me to express complex patterns in sufficient level of detail	2.25
This program query language strikes the right balance between expressiveness and ease of use	1.75
Queries in this program query language are declarative	2.125
Overall, I preferred this program query language over the second one I tried	1.75

Figure 6.1: Mean feedback for the logic metaprogramming language

Question	Prolog	Pyttern	Regex	AST
Use this program query language frequently	**1.375**	1.5	**3.25**	1.666
Language unnecessarily complex	**2.125**	**1.357**	1.75	1.833
Language easy to use	1.75	**2.357**	2	**1.666**
Need the support of a technical person	**2.375**	2.357	**0.75**	1.666
Various features well integrated	2.375	**1.643**	2	**2.666**
Too much inconsistency in the language	1.625	**2.071**	1.25	**0.833**
Most people would learn to use it very quickly	**0.625**	**2.286**	0.75	1
Cumbersome to use	**2.625**	**1.375**	2.25	1.666
Confident using the language	**1.25**	1.357	**1.5**	1.333
Need to learn a lot of things to use the language	**3.375**	1.214	**1**	2.333
Queries reports many false positives (incorrect responses)	**1.375**	2.429	**2.5**	2
Queries yields a lot of false negatives (missed responses)	1.5	**2.357**	**1.25**	1.333
Executing queries in this language was sufficiently fast	2.125	**2**	**2.75**	**2**
Easy to understand how to debug the query	1.25	**0.786**	**2**	1.666
Language sufficiently powerful to allow expressing complex patterns in sufficient level of detail	2.25	**1.785**	2	**2.333**
Language strikes the right balance between expressiveness and ease of use	**1.75**	2.071	**1.75**	**2.166**
Queries in this language are declarative	**2.125**	**3**	2.75	2.5
Preferred this program query language	**1.75**	1.928	**3**	2.333

Figure 6.2: Mean feedback for the different language of the experience

6.2.3 Threat to validity

The main challenge in validating the experience is the limited number of students involved in the experiment. Additionally, even if some students have background tutoring introductory programming courses, they are not candidates for using our tool to detect coding flaws in student programs beyond the experiment.

It's important to note that the definition of flaws made for the experiment seems to be understood differently by some students, with details in the implementation that change between students. It could also be attributed to the wide range of Python programming possibilities, making detecting specific issues challenging.
To enhance the experience, providing clearer Python code examples of the flaws we wanted to detect could have helped participants test their queries more effectively and obtain higher-quality results.

It's also worth mentioning that one student encountered an issue when running the Prolog code, as they were using an older version of SWI-Prolog. It revealed that our tool only worked with a version higher than 9.0. Consequently, this student spent time resolving the problem, likely impacting its overall experience.

Chapter 7

Final discussion

This thesis focused on developing a method for identifying problematic coding practices in student programs. This method would not only help teachers provide better feedback to students but also assist automated correction tools in teaching students good programming practices. Our approach involved using logical metaprogramming to transform the abstract syntax tree of a Python program into a set of logic facts.

To navigate between different nodes and extract relevant information, we created basic predicates that could be combined to form more complex queries for identifying good or bad coding practices. However, we encountered challenges with this approach. We found that our attempt to closely mirror the structure of the Python AST resulted in complex predicates, making it difficult to extract useful information. In an effort to simplify our approach, we focused on a more declarative writing style, resulting in less generic but more user-friendly predicates. Unfortunately, our experiments with students revealed that this did not effectively solve the problem.

Another issue we encountered was the complexity of creating queries. Detecting single nodes or counting nodes in the program's AST is relatively straightforward to implement. However, identifying flaws represented by a more complex structure of nodes linked together by imprecise paths is more challenging. The main challenge we encounter is the navigation in the AST as in Arts' master thesis [24]. It also became apparent that in order to detect a specific coding flaw, one would need to anticipate all the different cases in which that flaw could occur. This made the tool cumbersome to use, as it required the query developer to foresee all possible scenarios, which proved challenging for students without clear examples.

In conclusion, our method of using logic metaprogramming shows promise for detecting coding flaws in students' source code. It could successfully detect simple

properties by taking advantage of the functionalities and specific characteristics of logic programming. But some problems remain :

Edge-cases and AST Representation

In this master thesis, we aimed to detect coding flaws in Python by using its Abstract Syntax Tree as an intermediary code representation. However, we faced difficulties when attempting to create queries. We had to go through an intermediate stage, which involved discovering the AST representation of the flaw we were trying to identify. We believe that before using our tool to detect this representation, we need to theorize about the possible structures in the tree that generate the flaw we wish to detect. As a result, we will divide the task into two steps: the first step is to identify the problematic structures in the AST, and the second is to develop a tool that can detect these structures.

AST Navigation

Creating queries can be challenging due to the complexity of accurately navigating the AST. Retrieving specific information requires sometimes following a particular branch of the tree, and understanding the workings of the AST can be complicated at times. Improving our tool by developing predicates to simplify AST navigation and recurring practices in it would be beneficial to be able to implement new queries more easily.

Declarative Language and user-friendliness

This project aimed to develop a declarative language that could be easily manipulated by a skilled user in logic programming to detect bad practices in the students' programs. However, our experience with the students revealed that sticking closely to the structure of the AST of Python makes it challenging for new users to use our tool. We believe that creating specific predicates that deviate from Python abstract syntax tree structure and align more closely with how Python code is actually written would improve the tool's usability.

Use of generative AI tool

Generative AI tools were utilized in the writing of this master's thesis. They were employed to enhance and refine the written content. It is important to note that these tools were not used to generate text without the author's verification. Their role in the writing process for a text section was as follows:

1. A first version of the text was written
2. The text is then given to a generative AI tool to improve it using prompts such as : "Improve the text", "Adapts the text to an academic level" or "Use synonyms to suggest a variation of the following text"
3. Depending on the results of the prompts, the first version of the text is adapted.

Appendix

A clean version of the code is available as an appendix to this master thesis.

Bibliography

[1] G. Derval, A. Gego, P. Reinbold, B. Frantzen, and P. Van Roy, "Automatic grading of programming exercises in a mooc using the inginious platform," *European Stakeholder Summit on experiences and best practices in and around MOOCs (EMOOCS'15)*, pp. 86–91, 2015.

[2] K. Mens, S. Nijssen, and H. Pham, "The good, the bad, and the ugly: mining for patterns in student source code," in *EASEAI 2021: Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence, Athens, Greece, 23 August 2021* (A. Vescan, C. Serban, J. Henry, and U. Praphamontripong, eds.), pp. 1–8, ACM, 2021.

[3] S. Combéfis, "Automated code assessment for education: Review, classification and perspectives on techniques and tools," *Software*, vol. 1, no. 1, pp. 3–30, 2022.

[4] R. Saikkonen, L. Malmi, and A. Korhonen, "Fully automatic assessment of programming exercises," in *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pp. 133–136, 2001.

[5] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2017, Bologna, Italy, July 3-5, 2017* (R. Davoli, M. Goldweber, G. Rößling, and I. Polycarpou, eds.), pp. 110–115, ACM, 2017.

[6] D. M. Breuker, J. Derriks, and J. Brunekreef, "Measuring static quality of student code," in *Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29, 2011* (G. Rößling, T. L. Naps, and C. Spannagel, eds.), pp. 13–17, ACM, 2011.

[7] L. Östlund, N. Wicklund, and R. Glassey, "It's never too early to learn about code quality: A longitudinal study of code quality in first-year computer science

students," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 1, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023* (M. Doyle, B. Stephenson, B. Dorn, L. Soh, and L. Battestilli, eds.), pp. 792–798, ACM, 2023.

[8] M. D. Ernst, "Invited talk static and dynamic analysis: synergy and duality," in *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004* (C. Flanagan and A. Zeller, eds.), p. 35, ACM, 2004.

[9] T. Ball, "The concept of dynamic analysis," in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings* (O. Nierstrasz and M. Lemoine, eds.), vol. 1687 of *Lecture Notes in Computer Science*, pp. 216–234, Springer, 1999.

[10] V. Gupta and J. K. Chhabra, "Measurement of dynamic metrics using dynamic analysis of programs," in *Proceedings of the WSEAS International Conference on Applied Computing Conference*, pp. 81–86, 2008.

[11] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[12] A. Gosain and G. Sharma, "A survey of dynamic program analysis techniques and tools," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014 - Volume 1, Bhubaneswar, Odisa, India, 14-15 November 2014* (S. C. Satapathy, B. N. Biswal, S. K. Udgata, and J. K. Mandal, eds.), vol. 327 of *Advances in Intelligent Systems and Computing*, pp. 113–122, Springer, 2014.

[13] S. C. Johnson, *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[14] I. Albluwi and J. Salter, "Using static analysis tools for analyzing student behavior in an introductory programming course," *Jordanian Journal of Computers and Information Technology (JJCIT)*, vol. 6, no. 3, pp. 215–233, 2020.

[15] P. KASZAB and M. CSERÉP, "Detecting programming flaws in student submissions with static source code analysis.,"

[16] S. H. Edwards, N. Kandru, and M. B. Rajagopal, "Investigating static analysis errors in student java programs," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pp. 65–73, 2017.

[17] S. A. Mengel and V. Yerramilli, "A case study of the static analysis of the quality of novice student programs," in *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pp. 78–82, 1999.

[18] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' java programs," in *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pp. 317–325, 2004.

[19] M. Striewe and M. Goedicke, "A review of static analysis approaches for programming exercises," in *International Computer Assisted Assessment Conference*, pp. 100–113, Springer, 2014.

[20] P. Freeman, I. Watson, and P. Denny, "Inferring student coding goals using abstract syntax trees," in *Case-Based Reasoning Research and Development: 24th International Conference, ICCBR 2016, Atlanta, GA, USA, October 31-November 2, 2016, Proceedings 24*, pp. 139–153, Springer, 2016.

[21] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, IEEE, 1998.

[22] M. Zheng, X. Pan, and D. Lillis, "Codex: Source code plagiarism detection based on abstract syntax tree.," in *AICS*, pp. 362–373, 2018.

[23] M. Klinik, P. Koopman, and R. van der Wal, "Personal prof: Automatic code review for java assignments," in *Proceedings of the 10th Computer Science Education Research Conference*, pp. 31–38, 2021.

[24] E. Arts, *Towards Querying Abstract Syntax Trees for Python Programs*. PhD thesis, Master Thesis, Department of Computer Science, Eindhoven University of . . . , 2022.

[25] Y. Lilis and A. Savidis, "A survey of metaprogramming languages," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 113:1–113:39, 2020.

[26] R. Kowalski, "Logic programming," in *Handbook of the History of Logic*, vol. 9, pp. 523–569, Elsevier, 2014.

[27] K. D. Volder, "Aspect-oriented logic meta programming," in *Object-Oriented Technology, ECOOP'98 Workshop Reader, ECOOP'98 Workshops, Demos,*

and Posters, Brussels, Belgium, July 20-24, 1998, Proceedings (S. Demeyer and J. Bosch, eds.), vol. 1543 of *Lecture Notes in Computer Science*, pp. 414–417, Springer, 1998.

[28] N. Amin, W. E. Byrd, and T. Rompf, "Lightweight functional logic meta-programming," in *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings* (A. W. Lin, ed.), vol. 11893 of *Lecture Notes in Computer Science*, pp. 225–243, Springer, 2019.

[29] F. Flederer, L. Ostermayer, D. Seipel, and S. Montenegro, "Source code verification for embedded systems using prolog," in *Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, WLP 2015 / WLP 2016 / WFLP 2016, Dresden and Leipzig, Germany, 22nd September 2015 and 12-14th September 2016* (S. Schwarz and J. Voigtländer, eds.), vol. 234 of *EPTCS*, pp. 88–103, 2017.

[30] E. Hajiyev, M. Verbaere, and O. De Moor, "Codequest: Scalable source code queries with datalog," in *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*, pp. 2–27, Springer, 2006.

[31] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The soul tool suite for querying programs in symbiosis with eclipse," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pp. 71–80, 2011.

[32] P. Klint, T. Van Der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 168–177, IEEE, 2009.

[33] P. Klint, T. Van Der Storm, and J. Vinju, "Easy meta-programming with rascal," *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, pp. 222–289, 2011.

[34] S. Ceri, G. Gottlob, L. Tanca, *et al.*, "What you always wanted to know about datalog(and never dared to ask)," *IEEE transactions on knowledge and data engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[35] K. De Volder, "Jquery: A generic code browser with a declarative configuration language," in *Practical Aspects of Declarative Languages: 8th International*

Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006. Proceedings 8, pp. 88–102, Springer, 2006.

[36] R. F. Crew, "ASTLOG: A language for examining abstract syntax trees," in *Proceedings of the Conference on Domain-Specific Languages, DSL'97, Santa Barbara, California, USA, October 15-17, 1997* (C. Ramming, ed.), USENIX, 1997.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**