UCLouvain

epl

École polytechnique de Louvain

Jattern, a program query language for detecting coding idioms in Java programs

Author: **Lucie Leurquin**
Supervisors: **Kim Mens, Julien Liénard**
Reader: **Bastien Wiaux**
Academic year 2025–2026
Master [120] in Computer Science

Abstract

Code pattern matching can help detect common design patterns or coding flaws. One possible usage scenario is for analysing code of students who are learning how to program. Pyttern, however, is an existing pattern query language for analysing Python programs which aims to have a beginner-friendly learning curve.

This thesis explores the possible adaptation of Pyttern to other programming languages than Python, using Java as case study. The problem is two-fold. Firstly, we implement Jattern, a language for querying Java code inspired by and largely reusing the existing implementation of Pyttern. Secondly, we merge all the common parts between Pyttern and Jattern so that any implementation extending the framework to a future programming language will have an easier time.

We conclude the thesis with a guide explaining how to add support for other programming languages to the refactored version of Pyttern. Finally, we validate the results with a series of unit tests that have been developed using a small framework.

Contents

Chapter 1

Acknowledgments

Firstly, I would like to thank Professor Kim Mens and Julien Liénard for being my supervisors and helping me throughout this year-long journey. Thank you for being there every week answering my questions, listening to my ideas and getting me unstuck more times than I can count.

Secondly, I would like to thank my family and all of the friends I made in both my bachelor's and master's degrees. Getting this far was only possible thanks to my dream team's support and the many occasions for the rainbow to shine proudly.

Thirdly, I would like to thank my D&D tables with whom we leave our fate in the hands of a twenty-sided die.

Finally, thank you, dear reader, for taking the time to read this thesis.

~ Lucie Leurquin

Disclaimer

No artificial intelligence tools (such as ChatGPT) were used as part of this thesis.

Chapter 2

Introduction

2.1 Context

Pattern matching is widely used to automatically analyse code files to find patterns and flaws [1]. One use case is for educational practices, such as helping students understand weaknesses in their code through personalised feedback [2]. Yet, pattern matching tools often require specific knowledge and expertise which are significant barriers to entry.

Pyttern is an existing program querying language for Python code [3] which aims to be much easier to use by proposing a syntax that remains close to Python. Patterns in Pyttern are expressed as Python-like code, with special wildcards that can be used to abstract away parts of the program under analysis, such as the variable names or irrelevant blocks of code. However, Pyttern is made specifically for Python code, and does not yet support other programming languages such as Java.

2.2 Problem

The problem tackled in this thesis is *"How to refactor Pyttern to make it easily adaptable to analyse programs in another programming language?"*, using Java as a case study. In particular, we will also tackle the subquestion *"Can Pyttern be adapted to Java, and if so, what are the strengths and weaknesses of such an adaptation?"*

2.3 Motivation

Pattern matching tools, and more specifically program querying languages (PQL) tend to be complex to learn to use. They often have their own exclusive syntax, thus requiring a steep learning curve before being able to use them properly. That is especially true if the patterns are themselves complex, making them harder to express.

Pyttern was created as a PQL with a purposefully much more beginner-friendly learning curve. By directly using Python's syntax, anyone with knowledge of Python can dive quickly into Pyttern without needing to read much documentation. However, Pyttern is currently made only for Python code. It would be better if it could support other popular programming languages, such as Java, which is often used to teach Object-Oriented Programming.

2.4 Objectives

The first objective of this thesis is to implement Jattern, an extension of Pyttern made to support querying Java programs. We start by using Java's ANTLR grammar, then we extend it to support wildcards.

The second objective is to refactor Pyttern and Jattern by merging common parts that do not depend on the specific target programming language at all. This will make future adaptations to other programming languages easier.

The third objective is to provide a guide for future adaptations so that developers can easily follow it, without needing to understand everything that happens behind-the-scenes in Pyttern or Jattern.

This thesis will detail the implementation of Jattern, highlighting the differences between a pattern language for Java and for Python.

2.5 Contributions

This thesis contributes to program querying languages by:

- Implementing Jattern, a PQL designed to detect patterns in Java programs.
- Refactoring Pyttern (and Jattern) to provide a more modular approach, separating target language independent parts from those that are specific to each target language.
- Identifying and documenting how to generalise Pyttern to another programming language.
- Validating Jattern's implementation and Pyttern's refactoring through a series of automated tests.

2.6 Roadmap

In the next chapter, we explain how Pyttern's current implementation works and we explain the various obstacles we expect to face due to the differences between Python and Java.

In chapter 4, we report on related work and compare the different approaches to Pyttern. The implementation journey of Jattern is explained throughout chapter 5. In chapter 6, we refactor Pyttern and Jattern then include a guide for porting new languages to Pyttern. Afterwards, in chapter 7 we conduct a validation of our solution through unit tests. Finally, we conclude by summarising the contributions of this research, and presenting some possibilities for future work.

Chapter 3

Background Material

In this chapter, we cover how pattern matching works in Pyttern. We begin by describing ANTLR, the tool responsible for parsing, and its role within Pyttern. We then explain the syntax of Pyttern, followed by the different steps Pyttern goes through to compare a Pyttern pattern with a Python program and determine if they match. We conclude with the problem statement, explaining some of the obstacles we expect to face in our adaptation of Pyttern.

3.1 ANTLR

ANTLR (ANother Tool for Language Recognition) is used to generate the Pyttern lexer, parser and visitor. ANTLR is a powerful open-source parser generator [4] [5]. Given a context-free grammar, ANTLR generates a parser that can build and traverse parse trees. In ANTLR, ambiguity is solved by going through the different rules from top to bottom. In cases of ambiguity caused by left-recursion, the tool can verify the priority of operators.

This tool greatly simplifies the work of developers building a parser, as they only need to create a grammar file. This thesis ports Pyttern by also using ANTLR to create the Jattern lexer, parser, and visitor.

3.2 Pyttern

Pyttern is an open-source program querying language designed to analyse Python code [3]. Patterns defined by the user are compared to Python code files. The patterns are written in a syntax that is a superset of Python, with additional wildcards which function as "holes" that can be filled in during the matching. There are several kinds of wildcards, each of which can match a specific part of Python code. The different Pyttern wildcards are summarised in Table 3.1 and explained in greater detail in chapter 5.

Wildcard type	Syntax	Semantics
Simple wildcard	?	Match any element against the wildcard
Named wildcard	?var	Match and bind one element
List wildcard	?*	Match any number of elements
Number wildcard	?{n,m}	Match between n and m elements (inclusive)
Simple compound wildcard	?:	Match one level of indentation
Multiple compound wildcard	?*:	Match any level of indentation
Contains wildcard	?< >	Match a sub-part of an expression

Table 3.1: Pyttern Wildcards

In Listing 3.1 is an example of a Pyttern pattern and in Listing 3.2 is a program that would match this pattern. The question mark represents the simple wildcard, which matches any element. For example, if used as function name, it will match any function name (such as "foo"). This allows us to abstract away unimportant details when writing a pattern.

Listing 3.1: Example Pyttern pattern

```
def ?():
    ? = 0
    return ?
```

Listing 3.2: Example Python program

```
def foo():
    x = 0
    return "bar"
```

Pyttern's querying logic works in several steps.

- **Pattern definition.** The user writes a pattern using Pyttern's syntax, potentially using wildcards if they desire to abstract some parts of the code to match against. This pattern will be what we are trying to match.
- **Pattern parsing.** The pattern file is parsed into an AST (Abstract Syntax Tree). For that, we use the lexer and parser previously generated by ANTLR.
- **Pattern pruning.** We prune the AST to remove syntactic elements which carry no important information beyond the parsing phase and may get in the way of matching, such as parentheses or commas. Without the pruning, cases such as "print(a); print(b)" and "print(a) \n print(b)" would be considered different by the matcher, even though they are structurally identical, and we would like them to match.
- **Creation of the PDA.** We visit the AST to create a non-deterministic pushdown automaton (PDA). This PDA represents the pattern and determines the possible AST traversals that will be done during the matching phase. The visitor used to

create the PDA inherits from the one implemented by ANTLR. Some methods are overridden to add new kinds of transitions to the PDA. This is where wildcards are implemented, as we add special transitions.

- **Code parsing & pruning.** The Python program we want to compare the pattern to is parsed into an AST. We use the same lexer and parser generated by ANTLR, then prune it.
- **Matching.** The PDA initiates the matching at the topmost node of the code's AST. The PDA then applies the various transitions until it either reaches its final node (acceptance state) or it reaches the end of the AST before that (refusal state).

A flowchart of the process can be seen in Figure 3.1.

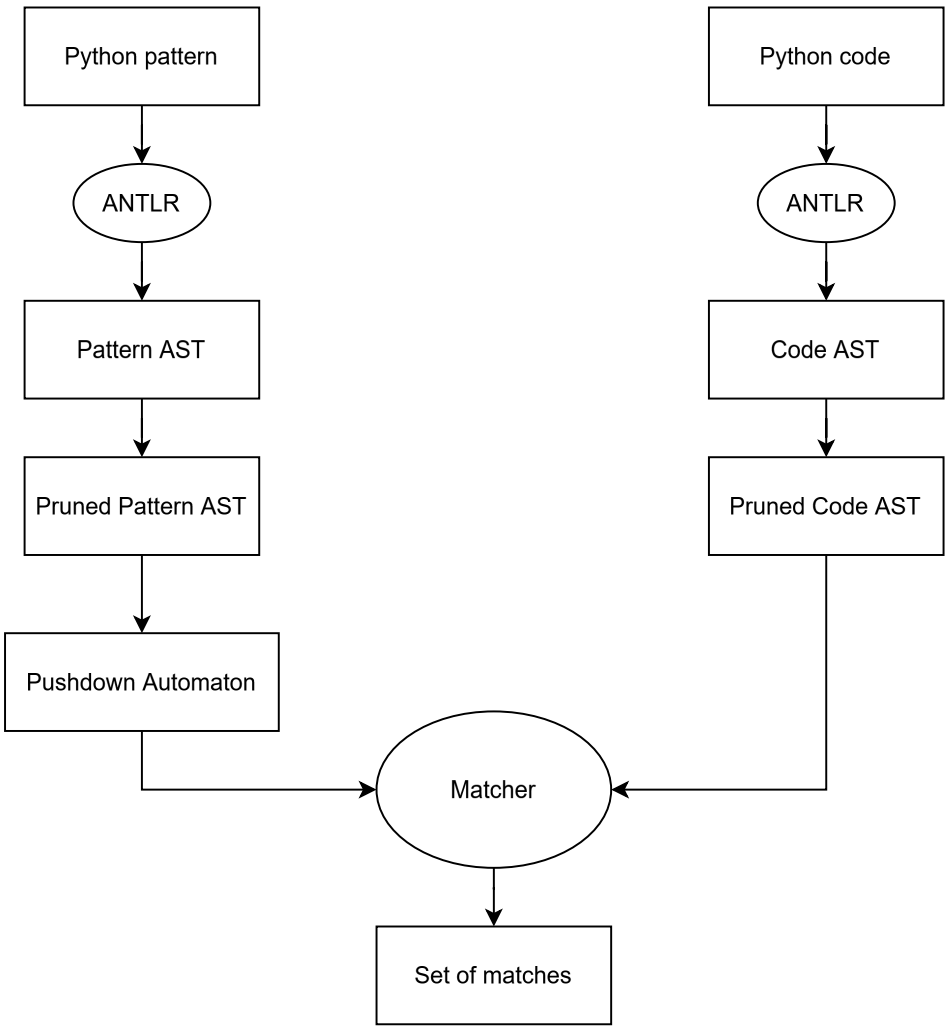

Figure 3.1: Flowchart of Pyttern

During the matching, Pyttern's AST traversal is a prefix traversal. First, we visit the parent node, then its child nodes from leftmost to rightmost. Wildcards are exceptions to this rule, and make the matcher move around the tree differently.

To visualise the steps taken by the Pyttern matcher, a web visualisation tool was made by Julien Liénard. This tool proved crucial in understanding how to implement the wildcards and debug when needed. The user can provide a pattern, which the tool will then parse and create the associated PDA. The user can then look at either the AST of the pattern or the PDA. The user can also provide code to match, which the tool will again parse. The user can then do a step-by-step visualisation of the PDA's traversal through the code's AST.

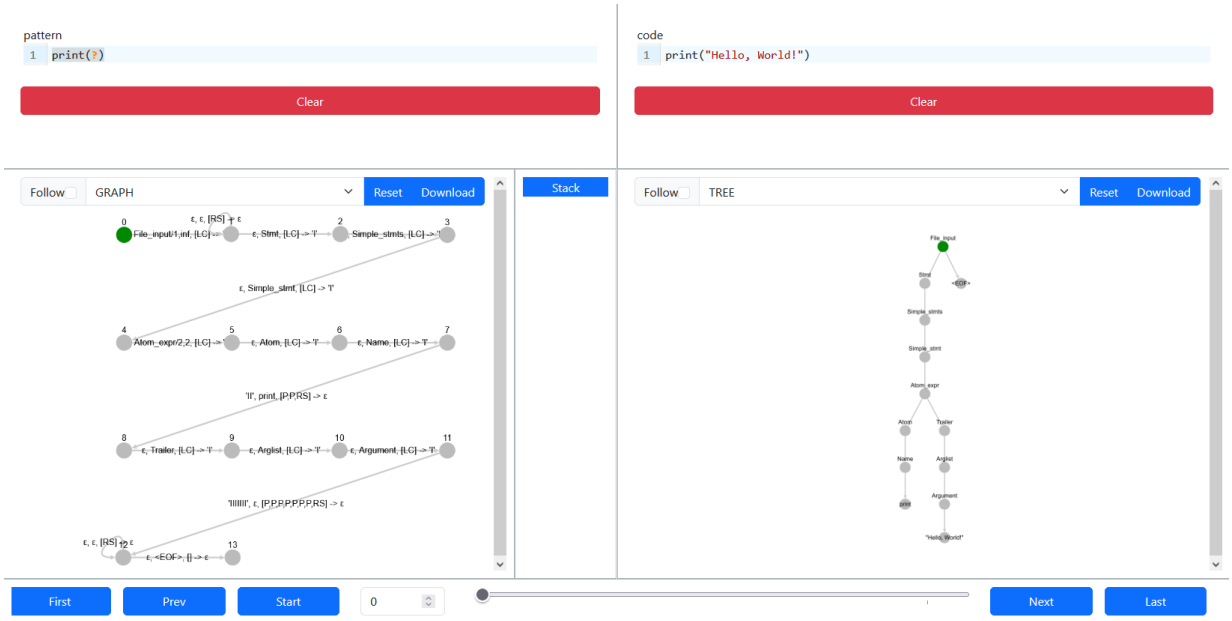

Figure 3.2: Pyttern web visualiser

An example of the visualisation tool can be found in Figure 3.2. We see on the top left the pattern code and on the bottom left the corresponding pushdown automaton. In the middle, we can see the current status of the stack of the pushdown automaton, which starts empty. On the top right, we see the code we will match against and on the bottom right is the code's AST. The user can interact with the various buttons on the bottom to see how the matching algorithm runs.

Transitions in the PDA have the following characteristics:

- The current PDA state and the new state.
- The stack symbol(s) that will be popped (if any) and the stack symbol(s) that will be pushed (if any).

- The input symbol, also called the transition condition. It is an optional constraint on the node that will be matched. The constraint can be on the type of node and on the number of children (later referred to as boundaries).
- The AST's movements. It is a list of any combination of these symbols:
  - *Move to Left Child* (abbreviated as LC)
  - *Move to Right Sibling* (abbreviated as RS)
  - *Move to Parent* (abbreviated as P)

When several transitions are possible, the PDA does not use any heuristic to determine the order in which they will be attempted. Instead, they will all be attempted and the resulting states will be added to the list of states. There is also an option to stop at the first occurrence of an accepting state if the user is only interested in knowing whether there is a match or not but not the number of matches.

Pyttern is thus a versatile tool that enables developers to easily create patterns, without barriers to entry. Its beginner-friendly syntax ensures that anyone who knows Python can quickly get into it. The wildcards allow for "holes" in the pattern to be filled during the matching phase, which the visualiser helps understand.

3.3 Problem Statement

Now that we have explained how Pyttern works, we can better understand the difficulty of porting Java to Pyttern. Indeed, Python and Java are two very popular, but different languages.

- **Typing.** Java uses statically strong typing: all variables need to have a single clearly defined type. On the other hand, Python is dynamically typed: any variable can have any type at any point.
- **Structure.** Java code blocks are wrapped by curly brackets "{}", while Python uses colons ":" and indentation.
- **Explicit rule naming.** Python's ANTLR grammar often has explicit rule names, such as "if_stmt". Java does the opposite by often inlining the rules. For example, there is no one rule for all if statements, and instead they are directly contained within the Statement rule. The opposite situation also occurs several times, although more rarely.

When porting to Pyttern, we need to follow these steps:

- Create the lexer rules and parser grammar which will then be used by ANTLR to generate the lexer and parser.

- Create the tree pruner.
- Implement the creation of a PDA.

The matching algorithm itself is language-independent. Indeed, once the PDA is complete, the matcher can just explore any AST regardless of what language it originally came from.

A previous attempt at creating Jattern was made, although back then Pyttern used a Finite State Machine instead of a pushdown automaton [6]. This previous thesis laid the groundwork for a Java adaptation, notably defining "processors" and determining the Jattern wildcard symbol to be the hash symbol ("#"). However, much of it had to be remade from scratch due to the updates of Pyttern as well as issues in the original work, which will be discussed in chapter 5.

Chapter 4

Related Work

In this chapter, we list other types of program query languages and compare their approaches to Pyttern. We group them by their type of matching algorithm and discuss their learning curve.

4.1 Regex

Regular Expressions are sequences of characters which define a pattern [7]. This pattern is then matched against strings. Many programming languages have implemented RegEx through modules. RegEx has many special sequences, which each have their own semantics. One such example is the dot (".") which matches any single character (except new line in some circumstances). Another example is the asterisk ("*"), also called a *Kleene star*, which matches the previous item any number of times.

RegEx is powerful and its underlying principles are easy to understand as it is the plain text of the program being matched, unlike Pyttern whose pattern matching requires a transformation to an AST. However, regular expressions quickly become difficult for humans to understand what they do. Some tools exist to help developers understand regular expressions [8], but it remains a non-trivial task. For example, Listing 4.1 shows a commonly used regular expression to verify whether a given string is a valid email address [9]. As a result, detecting structural patterns is easier to do in Pyttern.

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{
    |}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x
    5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0
    -9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z
    0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)
    {3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a
    -z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f
    ]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Listing 4.1: RegEx for email addresses

4.2 AST matching

In this section, we group together and discuss program query languages which use abstract syntax trees as part of their matching process.

4.2.1 SCRUPLE

SCRUPLE is a framework for searching patterns in C programs [10]. Patterns are defined using C syntax, with the addition of wildcards. There are several kinds of wildcards, depending on what the user wants to abstract. As such, there is no one simple wildcard but rather several types of "entity" wildcards. SCRUPLE matching first translates the pattern into a Finite State Machine, then feeds the code's AST to that FSM.

Pyttern and SCRUPLE are very similar as Pyttern's first implementation was closely inspired by SCRUPLE. Jattern's principles are also similar to SCRUPLE. As a result, SCRUPLE's learning curve is as beginner-friendly as Pyttern and Jattern.

4.2.2 Semgrep

Semgrep is an open-source pattern matcher [11]. Semgrep works with several programming languages, such as Python, Java and JavaScript. Patterns are defined in a syntax similar to the programming language chosen, which is then translated to a simplified AST. There are also simple wildcards with the dollar operator ("$") while the ellipsis operator ("...") allows matching any number of elements, just like a Kleene star.

The most interesting aspect of Semgrep when compared to Pyttern is that it already supports many programming languages as part of its design. It manages to combine both the simplicity of using the target programming language's syntax and the versatility of being able to cover many programming languages. However, Pyttern shines when it comes to the versatility of wildcards as Semgrep lacks wildcards such as the compound wildcards, therefore making Pyttern the most expressive of the two.

4.2.3 PMD

PMD is a static analyser tool [12]. Users can define (or import) rules to detect patterns such as bad practices, suboptimal code or security flaws. Rules can be written in XPATH or as Java classes. Rules can be matched against many programming languages such as Java and JavaScript. The matching is done by parsing code files into an AST using ANTLR.

While Pyttern and PMD share the idea of analysing the AST, PMD requires writing the pattern in an AST-like structure. PMD thus requires knowledge of how the language we want to match against is structured, which makes for a significant barrier to entry for beginners, unlike Pyttern.

4.2.4 Checkstyle

Checkstyle is a development tool that helps programmers adhere to coding standards [13]. Users can enable "checks", which function as rules the programmers should follow. While it does use the AST, Checkstyle is made for styling and consistency, but not for detecting structural issues or more advanced patterns, unlike Pyttern.

It is easy for beginners to use existing "checks", which cover most use cases. However, writing new rules is a complex task that requires manually implementing a visitor design pattern [14]. Therefore, while it is easy to learn, it is hard to master. This contrasts with Pyttern, which aims to be both easy to learn and to master.

4.3 Advanced matching

In this section, we group together and discuss program query languages which have their own advanced and unique method of matching.

4.3.1 PQL

PQL is a program query language focused on static and dynamic analysis of Java programs [15]. It is powerful enough to abstract object names and also implements a wildcard symbol. There are also actions that can be defined to take place once a match is found, such as replacing the match with another snippet of code.

To find matches, PQL translates the query into a Finite State Machine. Then, it instructs the target application to produce an abstract execution trace. The trace is then interpreted by the FSM. Different optimisations are performed at various steps to make the process more efficient.

However, PQL uses its own syntax, as shown in Listing 4.2, thus making it harder for new developers to get into, unlike Pyttern.

Listing 4.2: SQL injection query in PQL

```
query simpleSQLInjection()
uses
    object HttpServletRequest r;
    object Connection c;
    object String p;
matches { p = r.getParameter(_); }
replaces c.execute(p)
    with Util.CheckedSQL(c, p);
```

4.3.2 FindBugs/SpotBugs

SpotBugs is an open-source program using static analysis to look for bugs in Java code [16]. It is the successor of FindBugs. SpotBugs works by analysing Java bytecode and comparing it to a list of predefined standard bug patterns. These patterns include bad practices, performance issues, security flaws and more.

Unlike Pyttern which uses the code itself, SpotBugs uses bytecode and therefore requires compiling. SpotBugs is therefore not an option for programs whose bugs prevent a compilation in the first place. Additionally, adding new custom bug patterns to SpotBugs is a complex endeavour, as it requires reading and using SpotBugs's API.

Chapter 5

Implementation of Jattern

In this chapter, we explain how we implemented Jattern. We start with a toy case, which is trivial matching where we only match if the pattern and program are identical. We continue Jattern's implementation by iteratively porting more elements of Pyttern, such as relaxed mode and tree pruning. We discuss how each wildcard has been implemented, and the obstacles that appear due to the differences between Java and Python. The full code resulting from the implementation of Jattern as discussed in this chapter can be found in Appendix A.1.

5.1 Implementing trivial matching

Trivial patterns simply follow the Java grammar, as shown in Listing 5.1. Such a pattern would only match against a program that is fully identical.

Listing 5.1: Example trivial pattern

```
class HelloWorld {
    public static void hello_world() {
        System.out.println("Hello, World!");
    }
}
```

Trivial matching is straightforward to implement. For Jattern, we started by copying the Pyttern implementation, replacing the inheritance from the Python Parser Visitor with inheriting from the Java Parser Visitor. We removed all references to wildcards as they are not the current concern. We are left with a very bare-bones implementation: it can only match two identical files.

It is important to explain how trivial matching works before explaining the wildcards, which differ from the trivial match. We start at the topmost node of the code's AST. The PDA applies the transition that corresponds to that node, which means moving through the

AST. If the PDA expects the node to have children, the movement will therefore be *Move to Left Child*. If the node is a leaf node, the movement will therefore be *Move to Parent* as many times as needed (potentially zero), followed by a *Move to Right Sibling* to continue our prefix traversal.

The matcher finishes by respecting one of the following:

- The PDA reaches its final state. The final state is accepting, therefore the code matches the pattern.
- The PDA reaches the end of the AST without being in its final state. Only the final state is accepting, therefore the code does not match the pattern.
- The PDA is stuck before reaching the end of the AST or its final state because none of the transitions can be taken from the current position of the AST. This is actually the same case as the previous one, therefore the code does not match the pattern.

5.2 Defining boundaries

While traversing the AST, when the matcher reaches a node, it also verifies the "boundaries". Boundaries are the minimum and maximum number of children the node in the AST can have. Most of the time, these boundaries are equal to the number of children in the pattern.

For example, if a method's argument list has two children in the pattern (because it has two arguments), we expect to have two children in the code too. If the boundaries are not respected (in this example, because there would be only one child or more than two), the matching stops before we explore the children.

However, many times, the number of children is not as clear and therefore the boundaries should be adapted accordingly. One such example in Python is "if" statements which can have any number of children because there is no limit on the number of "elif" follow-ups. Java also has similar cases, albeit not the same. For example, "if" statements in Java can have one or two children, but not more.

Unfortunately, one of the lessons learned is that there is no clear-cut method to find all of the cases that need consideration. It requires knowledge of how the language is structured. To find these cases, we implemented many unit tests (which are described in chapter 7). Another lesson was that the usual suspects which require special care are rules for which the number of children may vary and whose child in the pattern AST is a wildcard. Finally, the special cases of boundaries relative to the list wildcard (boundaries between 0 and infinity) and number wildcard (boundaries between the two given numbers) are unchanged from Pyttern.

5.3 Implementing relaxed matching

Relaxed matching introduces the possibility of skipping statements of the program. This means that if the program contains the same statements as the pattern, with more statements in between, the matcher should still yield a positive result. For example, in Listing 5.2 is a simple pattern, and in Listing 5.3 is a program that contains the pattern and statements in between. We would like the two to match, but for that we have to allow the PDA to skip statements.

Listing 5.2: Example relaxed pattern

```
class HelloWorld {
    public static void hello_world() {
        System.out.println("Hello, World!");
    }
}
```

Listing 5.3: Example relaxed code

```
class HelloWorld {
    public static void hello_world() {
        System.out.println("Hi!");
        System.out.println("Hello, World!");
        System.out.println("Hello again!");
    }
}
```

Pyttern makes a distinction between **strict mode** and **non-strict mode** (relaxed mode). In strict mode, as the name implies, the pattern has to be *exactly* as it is in the code; we cannot skip parts, therefore the example above should not match. Relaxed mode, on the other hand, allows skipping over parts of the AST as long as the general pattern is still followed. Because of the educational applications of Pyttern and Jattern, we want to make matches more often than not, to ensure the students can get feedback. Since we prefer to have false positives rather than false negatives, relaxed mode is the default.

In practice, to implement relaxed mode, we do the following:

- Look into the Java ANTLR grammar to identify which parts should be allowed to be skipped.
- Put the corresponding grammar nodes in a list.
- Inside the *VisitChildren* method, if the current node is one of those in the list of skippable nodes, we add a self-transition to the PDA. This transition will not move the PDA to another state nor pop/push on the stack, but it will do a *Move to Right Sibling* to skip over one node.

By defining the Statement rule as one of the nodes that can be skipped, we solve our earlier problem. Following the example from Listing 5.3, the PDA can use the self-transition to move from the first println to the second. Since the PDA expects no siblings after the "Hello, World!", it directly goes back to the parent without visiting the third println statement.

To identify the parts that should be skippable, a good place to start is the statement grammar rule. The different nodes that are types of statements should be skippable. For Pyttern, there is an intermediate statement rule which is the same one used everywhere throughout the Pyttern grammar; therefore Pyttern only needs to be able to skip the statement node. For Jattern, it is more complex, as there is no intermediate statement grammar rule. Instead, we have to look into the nodes that can be children of the file input rule (called *CompilationUnitContext* in Java). Additionally, unlike Pyttern, we would like to be able to skip class method nodes, class attribute nodes and the like in Jattern.

The list of nodes that can be skipped in relaxed mode and the reasons why are as follows:

- *BlockStatementContext*: This is the node for statements inside class methods. It is the closest node to the Python equivalent.
- *PackageDeclarationContext*, *ImportDeclarationContext* and *TypeDeclarationContext*: These are the possible children of the file input rule. They are a sort of statement outside class methods and as such should be skippable.
- *ClassBodyDeclarationContext*, *InterfaceBodyDeclarationContext*, *RecordDeclarationContext* and *CompactConstructorDeclarationContext*: These are the possible declarations of methods, fields, etc. inside classes, interfaces and records. Unlike Pyttern, we would like to be able to skip them to gain the full intended benefits of relaxed mode, since adding more fields or methods does not go against the structure of patterns.

Finally, there is one last thing required: managing the boundaries. Since there can be more children nodes than expected (because they are parents of nodes that can be skipped), we have to set the upper bound to infinity for nodes that are parents to skippable nodes. This applies to the following nodes: the starting node (*CompilationUnitContext*), class nodes (*ClassBodyContext*), interface nodes (*InterfaceBodyContext*), record nodes (*RecordBodyContext*), and block nodes (*BlockContext*).

5.4 Tree pruning

After parsing the code into an AST and before sending it to the matcher, we want to simplify the AST by pruning parts of it. The main reason for the need to prune is that the tree created by ANTLR is closer to a concrete syntax tree (CST) than an abstract syntax tree (AST). It contains all syntactic elements and intermediate nodes, even if they are not all needed to represent the structure of the code or pattern. There are two kinds of cases we want to prune:

- Useless syntactic details. For example, "(x+1)" and "x+1" are identical expressions. The only importance of parentheses is during the parsing phase, but they carry no information afterwards, since that information is already relayed by the shape of the AST itself. We can systematically remove such leaf nodes to avoid them getting in the way of the matcher. If we did not, a pattern containing "x+1" would not match against "(x+1)" because of the lack of parentheses, even though they are structurally identical.

- Long chains of nodes that always have only a single child. For example, the Expression rule always ends with a *Primary* node which expands into a terminal node. The *Primary* node itself carries no information. Removing it makes the algorithm a bit faster and, more importantly, helps with depth issues which are explained in section 5.5.2.

Determining what to prune was surprisingly difficult. In Java, the semicolon ";" is by itself a Statement. At first, it seemed best to avoid pruning it because it might cause difficulties while matching the simple wildcard (which matches a single element) against a Statement. However, this caused ambiguity problems.

5.4.1 Ambiguity

Let us first consider the following case: a wildcard alone is considered a valid statement in Jattern.

Listing 5.4: Example of ambiguous pattern

```
class Calculator {
    void hello_world() {
        # #;
    }
}
```

In Listing 5.4, the pattern could either mean we want to match two Statements of any kind followed by a semicolon (shown in Listing 5.5), or a local variable definition of any kind (shown in Listing 5.6).

Listing 5.5: Should the pattern match this code?

```
class Calculator {
    void hello_world() {
        System.out.println("A"); // first statement
        System.out.println("B"); // second statement
        ;
    }
}
```

Listing 5.6: Or should the pattern match this code?

```
class Calculator {
    void hello_world() {
        int a; // local variable definition
    }
}
```

We cannot have such ambiguity, because those two different cases would result in two different PDAs and we must only create one. Therefore, the Statement rule cannot be expanded into a wildcard alone. The clear solution is that a Statement rule should be expanded to a wildcard followed by a semicolon. The ambiguous pattern is therefore a local variable definition. However, this solution introduces another problem: not all Statements have semicolons at the end.

Indeed, compound statements such as loops do not need a semicolon. Naively trying to match the simple wildcard against a loop would therefore fail. This is due to the fact that the simple wildcard will match the loop, but there is nothing to match against the semicolon. There are two possible solutions here: either we add an exception, or we remove all semicolons from the AST. Since semicolon nodes carry no important information, it is better to prune them from the AST.

This ambiguity issue taught us that languages that use semicolons should prune them, even if they are Statements on their own. This is likely applicable to many other languages, such as C.

5.4.2 Pruned elements

In the end, the tree pruner removes the following elements:

- Terminal nodes made of parentheses ("(" and ")"), colons from switch statements (":"), commas (","), dots ("."), curly braces ("{" and "}") and semicolons (";").
- Intermediate nodes (called *Primary*) between the Expression node and the leaf node.

Listing 5.7 shows a simple pattern that will be used to compare the AST before and after the pruning. In Figure 5.1 is a subtree of the original AST. Terminal nodes that were removed are highlighted in red boxes, whereas intermediate nodes that were removed are highlighted in yellow boxes. The result can be seen in Figure 5.2.

Listing 5.7: Pattern that will be pruned

```
class Calculator {
  public static int add(int a, int b) {
    return (a+b);
  }
}
```

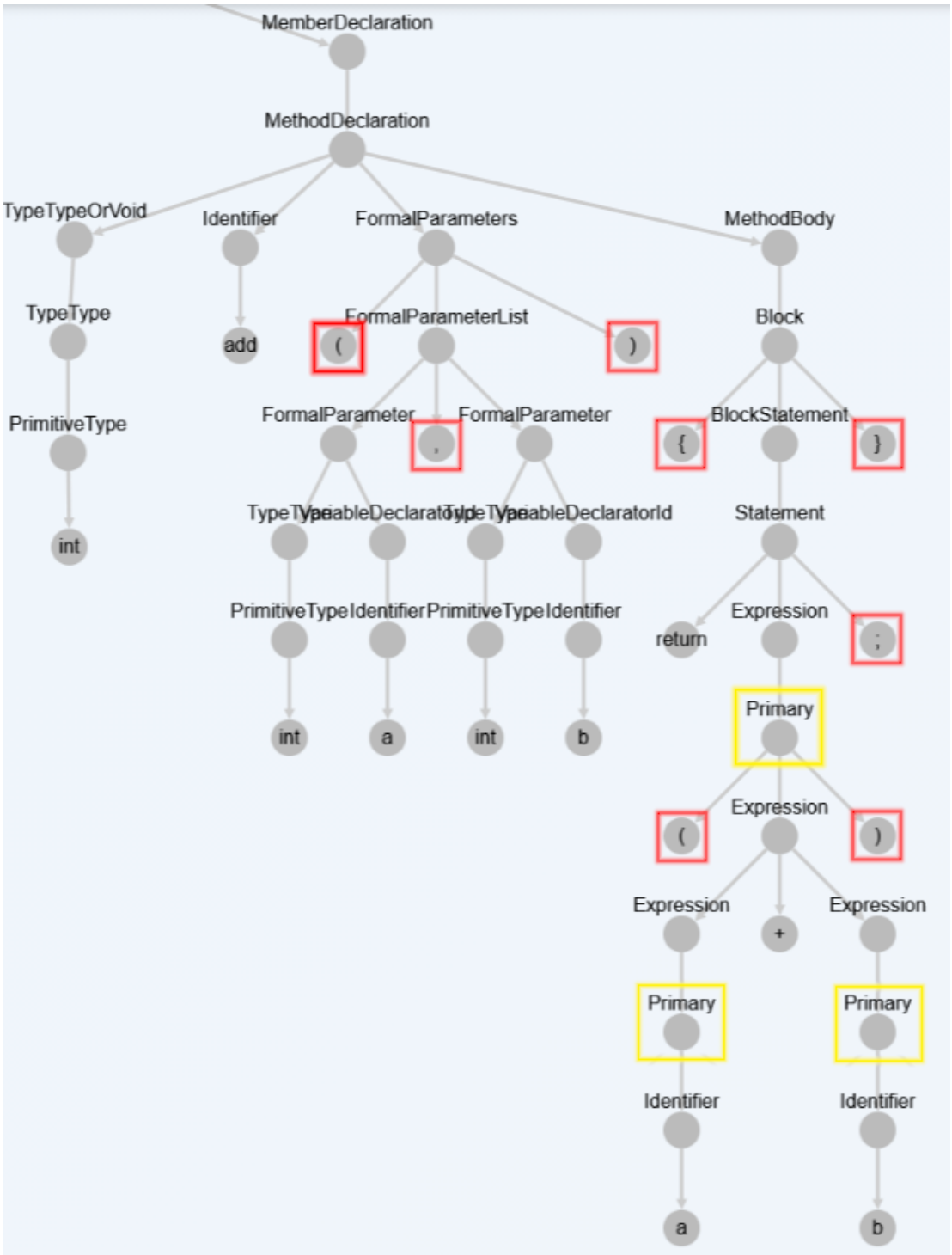

Figure 5.1: Before pruning

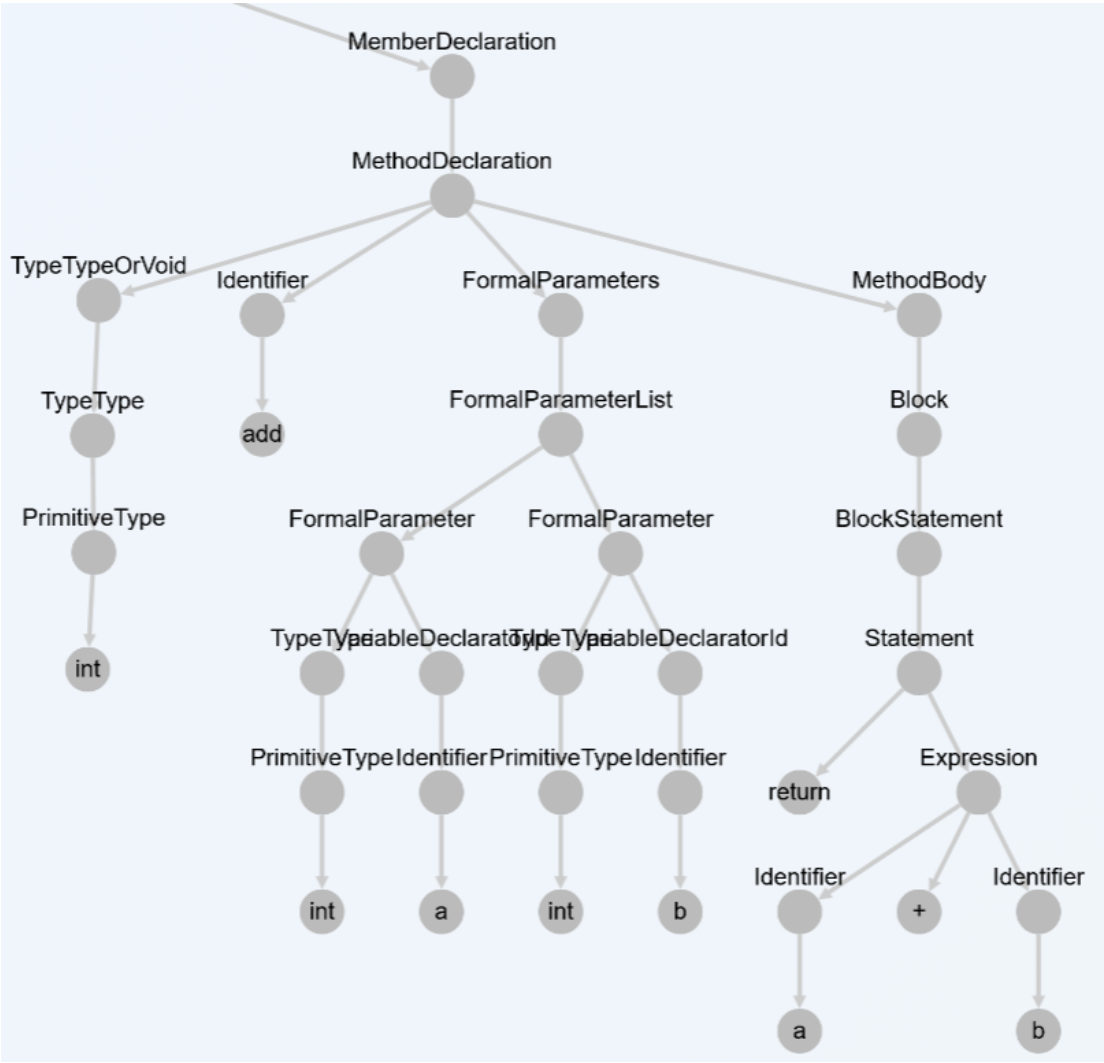

Figure 5.2: After pruning

5.5 Implementing Wildcards

It is now time to move to the most interesting aspect of Pyttern: wildcards. We have adapted them to Java's syntax, as shown on Table 5.1. We use the hash as the wildcard symbol instead of the question mark because the question mark already exists in Java's grammar [6]. Each wildcard is explained in more detail in its corresponding section.

Wildcard type	Syntax	Semantics
Simple wildcard	#	Match any element against the wildcard
Named wildcard	#var	Match and bind one element
List wildcard	#*	Match any number of elements
Number wildcard	#(n,m)	Match between n and m elements (inclusive)
Simple compound wildcard	#{}	Match one level of indentation
Multiple compound wildcard	#*{}	Match any level of indentation
Contains wildcard	#< >	Match a sub-part of an expression

Table 5.1: Jattern Wildcards

5.5.1 Simple Wildcard

Semantics

The simple wildcard matches an element in the tree without regard to the content. It is represented as only the wildcard symbol by itself (in Jattern, "#"). It can replace an identifier, a type, or an expression.

PDA behaviour

In Listing 5.8 is an example of the simple wildcard being used to abstract away the class name. In this case, the pattern should match the code given in Listing 5.9. The simple wildcard acts as a "hole" that can be filled by anything.

Listing 5.8: Example of simple wildcard

```
class # {
}
```

Listing 5.9: Example of code

```
class Calculator {
}
```

In practice, this means that, in the AST traversal, we ignore the entire subtree that matches this wildcard. If the subtree is the last child of its parent, we add an "up" transition by doing *Move to Parent* a number of times equal to the depth, followed by a single *Move to Right Sibling*. Otherwise, it is only a single *Move to Right Sibling*.

Implementation choices

One significant consequence of Java's grammar is that the simple wildcard cannot match the *void* keyword. This is due to the fact that keyword is not under the same ANTLR grammar rule (which defines the AST structure during the parsing) as other types. Indeed, it is under the *typeTypeOrVoid* rule, which then expands into *void* or *typeType*. While we

could have changed the *typeTypeOrVoid* grammar rule, we chose to leave it as is because it is more than we would like to abstract with the simple wildcard. If we had added it to the *typeTypeOrVoid* grammar rule, there would have been no way to create a pattern expressing "any return type except void". On the other hand, even without adding it to the *typeTypeOrVoid* grammar rule, there is still a possibility to express "any return type, including void" by using two different patterns. Maximising the expressiveness of Jattern therefore requires to not add the simple wildcard to *typeTypeOrVoid*.

We also made it possible to specify the dimensions of the type being abstracted. For example, putting "#[]" as type in the pattern will match any one-dimensional list but not two-dimensional lists nor primitive types. This is because we placed the wildcard in two different places in the *typeType* rule:

- The first location is at the top of the grammar rule, which will therefore be attempted first. If there is no "[]", the parsing using this rule succeeds and the simple wildcard will thus abstract the entire type. For example, the following excerpt of pattern "# x;" will allow any type for x.
- The second location is deeper in the grammar rules. In this location, the simple wildcard only replaces the primitive (or class) type, but not the entire type. This second grammar rule will only be used if the parsing using the first rule failed due to there being a "[]" after it. For example, "#[][] x;" will only allow a 2-dimensional matrix of any primitive type for x.

Java Grammar Consequences

Due to Java's typing, we can use the simple wildcard to perform Jattern queries that would otherwise be impossible with Pyttern. For example, it is impossible to distinguish between a generic statement and a variable declaration. However, it is possible to make this distinction in Jattern because we can use the simple wildcard at different levels. Alone, it would match all kinds of statements, but with two consecutive wildcards ("# #;") it can only match a local variable definition.

Similarly, we can also distinguish between a variable definition and a variable assignment. Indeed, the pattern "# = #;" would have to be a variable assignment while the pattern "# # = #;" is definition and assignment together. None of these patterns can be expressed in Pyttern.

Lessons learned

The main difficulty of the simple wildcard is determining where to place it in the ANTLR grammar, as that corresponds to which parts will be abstracted. There is otherwise no difference with how it is handled in Pyttern, since ignoring the subtree is necessarily not grammar-dependent. This wildcard is therefore likely generalisable.

5.5.2 Named Wildcard

Semantics

The named wildcard is represented as the wildcard symbol followed by an identifier. Its semantics are identical to the simple wildcard with one additional constraint: all named wildcards with the same name must match identical parts of the AST. The first time a wildcard with a given name is encountered, it *binds* the subtree it matches to its name. If a named wildcard with the same name is encountered later, it should match the exact same subtree. Named wildcards can be used anywhere simple wildcards can be used.

PDA behaviour

Listing 5.10: Example of named wildcard

```
class Calculator {
    int inc(int #argument) {
        return #argument + 1;
    }
}
```

In the example shown in Listing 5.10, upon encountering the "#argument" named wildcard inside the arguments of the *inc* method, the matcher will store the corresponding subtree. Then, when descending into the expression, it will see the same named wildcard and compare it to the stored subtree. If they are identical, as in the case of Listing 5.11, the matching algorithm continues. If they are not, as in the case of Listing 5.12, the matching algorithm stops and does not match.

Listing 5.11: Example of matching program

```
class Calculator {
    int inc(int x) {
        return x + 1;
    }
}
```

Listing 5.12: Example of non-matching program

```
class Calculator {
    int inc(int x) {
        return 1 + 1;
    }
}
```

Named wildcards can be used anywhere simple wildcards can be used.

Ambiguity

Named wildcards introduced another problem of ambiguity. Usually, spaces are removed during the lexer phase and do not impact the parser at all. However, spaces distinguish between the simple wildcard followed by an identifier and a named wildcard, as shown in Listing 5.13.

Listing 5.13: Named wildcard ambiguity

```
class Calculator {
    void hello_world() {
            # a; // this should match a local variable definition
            #a; // this should be a named wildcard and match a statement
        ;
    }
}
```

To solve this issue, an earlier attempt at creating Jattern made a distinction in the lexer between wildcards that were followed by whitespaces, and wildcards that were not [6]. Then, in the parser, the simple wildcard could be either of those wildcards, whereas the named wildcard could only be a wildcard not followed by a white space combined with an identifier. While functional, this solution did not solve all ambiguity problems.

Listing 5.14: Class definition with named wildcard

```
class #name {
    #name () { }
}
```

In the example from Listing 5.14, we would like to match a class and its constructor. However, because the method definition rule comes before the constructor definition rule, ANTLR will first try to parse *#name*(){} as a method definition. Under these circumstances, the "#" will be the simple wildcard and "name" will be the name of the method. Since ANTLR tries rules from top to bottom, it will consider this to be a valid method definition and will not consider it to be a constructor.

There are several possible solutions to this problem.

- Put the constructor rule before the method rule. While this would work for this specific instance, it may cause other parsing problems. Moreover, there may be other instances of this problem. Therefore, it is not a good solution.
- Force the simple wildcard to always require having a space after it. This solution is quite effective and simple, but more annoying to the user. Taking this solution one step further, we could make the space required in some places, but not all. For example, in Statements, we could allow not having a space after it. However, this would need careful consideration in every location the simple wildcard can be used.

- Solve the problem during the lexing phase instead of the parsing phase. Because lexers use the *longest first match* principle, we could have a token that is defined as the wildcard symbol directly followed by an identifier (no space between). During the parsing phase, this token would then become the named wildcard.

We opted for the latter solution because it is simple and elegant, as it can work for any language, which is the end goal of this thesis. The lexer modifications for this solution can be seen in Listing 5.15.

Listing 5.15: New Jattern lexer rules

```
// Rules for wildcards
fragment HASH: '#';

WILDCARD_SPACE: HASH WS+;
VAR_WILDCARD: HASH IDENTIFIER;
WILDCARD : HASH;
```

Matching issues

Named wildcards introduced a second problem: not all occurrences have the same depth in the AST. In Listing 5.16 is a pattern that we would like to match with the code in Listing 5.17.

Listing 5.16: Named wildcard pattern example

```
class TestType {
    public static int main() {
      int #x = 1; // identifier
      return #x; // expression
    }
}
```

Listing 5.17: Named wildcard code example

```
class TestType {
    public static int main() {
      int foo = 1; // identifier
      return foo; // expression
    }
}
```

In the AST, the variable definition of foo will be considered as an identifier whose child node is the named wildcard. The return statement will consider foo as an expression whose child node is the named wildcard. There is a mismatch between the two cases, because the expression expands to *Primary*, to *Identifier*, and then to the value itself.

The problem can be visualised in Figure 5.3. The two purple subtrees on the left are identical, and therefore the two nodes on the right should be identical too. However, they are not, since the red subtree contains an extra node.

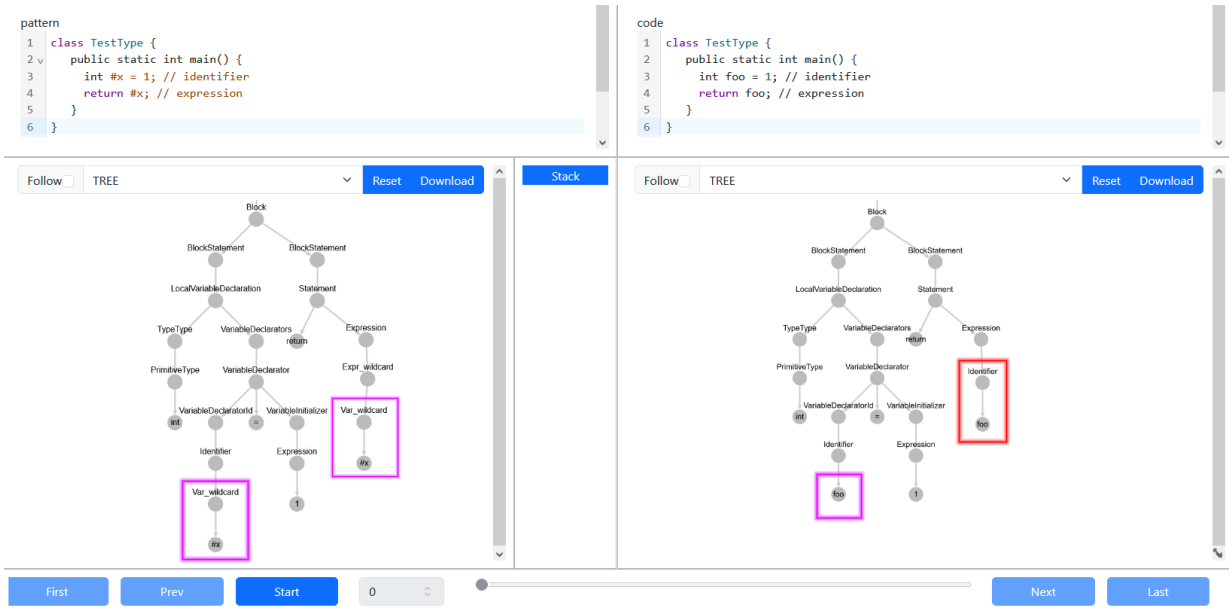

Figure 5.3: Named wildcard mismatch example

There are several possible solutions to this problem.

- Put the wildcard rules less deep in the grammar rules. For example, use $(identifier|simple\_wildcard|var\_wildcard)$ as in Pyttern instead of putting the wildcards inside the identifier rule. While this works, it requires more work from the developer to verify that all modifications do not break other parts of the grammar. It requires knowledge of the language and thorough testing. This is not compatible with our goal to make adapting Pyttern easy.
- Prune the identifier node. Since the identifier node would no longer exist, it would no longer get in the way of the matching. However, the identifier node carries information, and removing it could lead to other issues.
- Prune the identifier node only if its child is a wildcard. This is similar to the previous solution, but without losing information in the code AST. The resulting pattern AST will be the same as if using the first solution, but because the matching is done deeper and as the last rule, there will be no false positives.

Since the last solution combines the best of both worlds, with neither of the drawbacks, it is what we implemented.

Lessons learned

The named wildcards had two main difficulties. The first was the ambiguity, but we found a solution through the lexer that can solve the issue for all languages. The second was the varying depth, for which we found a solution that works but requires more manual intervention in the tree pruner. Neither issues modifies the PDA behaviour, though, so it is generalisable.

5.5.3 List Wildcard

Semantics

The list wildcard can match any number of elements, from 0 to infinity. It is represented as the wildcard symbol followed by an asterisk ("#*"). It can be used to replace the arguments of a method, whether in the definition of the method itself or when calling such a method. It can also be used to replace statement lists, although it is redundant with relaxed mode most of the time.

PDA behaviour

Listing 5.18: Example of list wildcard

```
class Calculator {
    void compute( #* ) {
    }
}
```

In the example pattern from Listing 5.18, the *compute* method can contain any number of arguments. Therefore, the example program from Listing 5.19 will match the pattern.

Listing 5.19: Example of matching program

```
class Calculator {
    void compute( int a, float b, String c ) {
    }
}
```

The PDA handles this wildcard by having a self-transition do the *Move to Right Sibling* movement.

Java Grammar Consequences

In Python, all bodies should have at least one statement in them. This is not the case in Java. This caused the need for Pyttern to have a different list wildcard which matches at least 1 element and up to infinity. It is called the double wildcard, and it only exists in Pyttern. Since Java has no such restriction, we do not need it.

However, it leads to an interesting consequence: if, in a pattern, there is a method that has no statements in it, the method node will be terminal. This is because we prune the "{" and "}" nodes, therefore the node has no children. Because the node is terminal, the boundaries are by definition $(0, 0)$; we expect no children. If we want to expect children, we can place the list wildcard in the pattern. This allows us to distinguish between "a method that has no statements" and "a method that has any number of statements".

Lessons learned

The list wildcard presents no special difficulty caused by the language differences since it is only a series of *Move to Right Sibling* movements. The only addition required is to call the *handle_empty_list* method when visiting the nodes parent to the list wildcard. This wildcard is therefore easily generalisable.

5.5.4 Number Wildcard

Semantics

The number wildcard matches a given number of elements. It is represented as the wildcard symbol followed by parentheses containing one or two numbers. There are three possible cases:

- $\#(n, m)$ will match between n and m elements (inclusive).
- $\#(n)$ will match exactly n elements.
- $\#(n,)$ will match at least n elements, up to infinity.

The number wildcard can be used in the same places as the list wildcard.

PDA behaviour

Listing 5.20: Example of number wildcard

```
class Calculator {
    void add( #(2,4) ) {
    }
}
```

Listing 5.21: Example of matching program

```
class Calculator {
    void add( int a, int b, int c ) {
    }
}
```

In the example from Listing 5.20, the *add* method should have between 2 and 4 parameters. Since the code from Listing 5.21 has 3 parameters (each type and identifier pair count as one argument together), which is within the acceptable range, the pattern will match the program.

The matching algorithm creates n intermediate states with a *Move to Right Sibling* transition. Then, the *nth* state has an empty transition to the next part of the AST. To allow up to m elements, we add $m - n$ additional nodes in the PDA that can be reached to perform a *Move to Right Sibling* transition, and all these $m - n$ nodes also have the empty transition to the rest of the AST.

If we are in the case where only n is given with no comma, then there are exactly n intermediate states with no additional nodes. If we are in the case where only n is given with a comma, then after the n intermediate states, the PDA behaves the same as with the list wildcard.

Lessons learned

The number wildcard was implemented exactly the same way as in Pyttern as it does not depend on the language at all. It is one of the most promising wildcards in terms of generalisation.

5.5.5 Simple Compound Wildcard

Semantics

The simple compound wildcard corresponds to an increase in indentation. It is represented as the wildcard symbol followed by curly braces ("#{ }") and may contain statements inside, as shown in Listing 5.22. This allows us to abstract away whether the indentation is from a for loop, a while loop, an if statement, etc. A program matching the example from Listing 5.22 can be seen in Listing 5.23.

Listing 5.22: Example of simple compound wildcard

```
class Calculator {
    void hello_world() {
        #{
            System.out.println("Hello, World!");
        }
    }
}
```

Listing 5.23: Example of matching program

```
class Calculator {
    void hello_world() {
```

```
        for (int i = 0; i < 5; i++) {
            System.out.println("Hello, World!");
        }
    }
}
```

Naive approach

To implement this wildcard, we split the Statement rule in two: regular statements were left as is, but all statements that increase in indentation were grouped together in a new *compound_stmt* rule. Then, the simple compound wildcard was added to that new rule. This does not change the behaviour of the parser, because all of the rules that were moved are LL(1), i.e., the first parsed element is different for each of the rules.

In this example, the matcher will first enter the compound statement. Then, it will *Move to Right Sibling* in a self-loop until it can match a *Statement* node.

Java Grammar Consequences

Unlike in Python, this approach does not hold for all types of compound statements. In Java, not all indentation directly have a *Statement* node. For example, try/catch statements instead have a *catchClause* node which then has a *Block* inside of which are *Statement* nodes.

To solve this issue, we have to consider all cases that do not fit the usual pattern. For each of these cases, we create a different valid path throughout the PDA.

This brings another issue to light: the depth is not always the same. In Pyttern, we can assume that the simple compound wildcard always increases the depth by one, but that does not hold in Jattern. A miscalculated depth will cause issues when trying to move to the next part of the match.

To solve this second issue, we need to change our approach.

PDA behaviour

Before entering the compound statement, we add a transition that pushes a "body" element onto the stack. Then, after matching the compound statement, we create an intermediate node. This node has a self-transition which will *Move to Parent* as many times as we have performed a *Move to Left Child* transition after entering the compound statement. Once the stack is reduced to only the "body" element, we pop it and move on to the rest of the match. This approach allows us to handle the depth in a more dynamic way.

Lessons learned

The simple compound wildcard was surprisingly the most difficult wildcard to adapt to Jattern. Using the stack to solve the depth issue was a simple approach to solve a complex problem, but it is likely that this wildcard will not be generalisable at all.

5.5.6 Multiple Compound Wildcard

Semantics

The multiple compound wildcard corresponds to any number of indentation increases (including 0). It is represented as the wildcard symbol followed by an asterisk, then curly braces ("#*{ }") which **must** contain at least one statement inside. The reason we do not allow an empty body inside the multiple compound wildcard is that we can match 0 indentation levels, and an empty body would therefore match everywhere and have no meaning.

PDA behaviour

Listing 5.24: Example of multiple compound wildcard

```
class Calculator {
    void hello_world() {
        #*{
            System.out.println("Hello, World!");
        }
    }
}
```

Listing 5.25: Example of matching program

```
class Calculator {
    void hello_world() {
        for (int i = 0; i < 5; i++) {
            if (i % 2 == 0) {
                System.out.println("Hello, World!");
            }
        }
    }
}
```

In the example from Listing 5.24 matching against Listing 5.25, the matcher will first enter the compound statement. This is when we put a "body" element onto the stack. Then, it will *Move to Left Child* and *Move to Right Sibling* in a self-loop until it can match a *Statement* node. Finally, once the matching is complete, we move back up with *Move to Parent* movements until we reach the "body" element again, at which point we pop it.

Lessons learned

The multiple compound wildcard was very easy to adapt. The only difficulty was in fetching the statements inside the wildcard. Apart from that, it is unchanged from Pyttern, making it likely generalisable.

5.5.7 Contains Wildcard

Semantics

The contains wildcard is used to verify whether a subsequence is contained within an expression or not. It is represented as the wildcard symbol followed by angle brackets ("#< >") containing an expression.

PDA behaviour

Listing 5.26: Example of contains wildcard

```
class Calculator {
    int compute(int x) {
        return #<x> ;
    }
}
```

Listing 5.27: Example of matching program

```
class Calculator {
    int compute(int x) {
        return 3*x + 1 ;
    }
}
```

In this example from Listing 5.26 matching against Listing 5.27, the matcher will first enter the expression statement. This is when we put a "body" element onto the stack. Then, it will *Move to Left Child* and *Move to Right Sibling* in a self-loop until it can match the subsequence x. Finally, once the matching is complete, we move back up with *Move to Parent* movements until we reach the "body" element again, at which point we pop it.

As one can observe from the above explanation, the contains wildcard is very similar to the multiple compound wildcard. The only difference is that the multiple compound looks for a *Statement* node, whereas the contains wildcard looks for a given subsequence inside an expression.

Lessons learned

The contains wildcard was implemented exactly the same way as in Pyttern, as it does not depend on the language at all. It is one of the most promising wildcards in terms of generalisation.

Chapter 6

Generalising Pyttern to other languages

Now that we have successfully implemented Jattern, the next goal is to refactor both Pyttern and Jattern to make it easier to add other languages in the future. Once the refactoring is complete, we explain the steps required to port Pyttern to other languages. The full code resulting from the refactoring of Pyttern and Jattern discussed in this chapter can be found in Appendix A.2.

6.1 Refactoring Pyttern and Jattern

The main aspect of Pyttern and Jattern is the creation of the PDA; therefore we would like to create a superclass of the PDA and pull up methods to this superclass. This superclass will be called *Generic_to_PDA*, as generalisation of *Java_to_PDA* and *Python_to_PDA*. Similarly, we would like to make a *GenericTreePruner* as an abstraction of the Pyttern and Jattern tree pruners. Finally, the last language-specific component we would like to pull into a superclass is the processor, which is responsible for calling all the other components.

6.1.1 Multiple inheritance

The first challenge we face is the multiple inheritance problem, also called the diamond problem. We would need the *Java_to_PDA* to inherit from ANTLR's *JavaParserVisitor* and from our newly created superclass. The ideal situation would be that, when calling a method, the method resolution order is to first look into *Java_to_PDA*, followed by *Generic_to_PDA*, and finally *JavaParserVisitor*.

Python can easily solve this issue because it supports multiple inheritance, and the order in which the classes are inherited determines the method resolution order. Therefore, all we need to do is make *Java_to_PDA* inherit from *Generic_to_PDA* first and *JavaParserVisitor* second. The tree pruner also needs multiple inheritance, which is

solved in the same way: the tree pruner inherits from *GenericTreePruner* first and *JavaParserVisitor* second.

6.1.2 Subclass attributes

Our next challenge is to generalise as much as possible. While some methods can already be trivially pulled up (such as *lookahead* and *lookbehind*), this is not the case for most of them. A common problem is that methods need to verify the type of a node in the AST, and this type will depend on the grammar being used. An example of the problem is shown in Listing 6.1, where we verify if the lookahead is the list wildcard.

Listing 6.1: Excerpt from Jattern's _handle_empty_list

```
def _handle_empty_list(self, ctx):
    list_wildcard = self.lookahead(ctx, JavaParser.List_wildcardContext)
    if list_wildcard is not None:
        ...
    return self.visitChildren(ctx)
```

To solve this issue, we set the grammar being used as a subclass attribute. Because the wildcards have the same name in both Pyttern and Jattern, we can simply compare the node to *self.grammar*[<*name of the wildcard*>]. The code from Listing 6.1 is therefore replaced by the one in Listing 6.2.

Listing 6.2: Excerpt from Jattern's refactored _handle_empty_list

```
def _handle_empty_list(self, ctx):
    list_wildcard = self.lookahead(ctx, self.grammar.List_wildcardContext)
    if list_wildcard is not None:
        ...
    return self.visitChildren(ctx)
```

For other nodes that do not have the same name in both grammars, we place them inside specific lists. The nodes which are statements, and therefore should be allowed to be skipped (due to relaxed matching), are all placed inside the list attribute *skippable_nodes*. The list wildcards (only one in Jattern, but two in Pyttern as explained in chapter 5.5.3) are all placed inside the list attribute *remove_double_wildcard*. Finally, we also add the grammar-specific tree pruner to the subclass attribute *tree_pruner*.

The same logic is used for the generic tree pruner to determine the nodes that should be pruned or kept, which are respectively stored in the *TO_REMOVE* and *TO_KEEP* attributes.

6.1.3 What cannot be pulled up

We were able to pull up most of the methods, but some are too grammar-specific to be pulled up. The following methods remain:

- *define_boundaries*: The minimum and maximum number of children possible is fully dependent on the grammar structure of the language. It is also dependent on what nodes can be skipped as part of the relaxed mode. There is no way to generalise it, and it has therefore been left as an abstract method in the superclass.
- *visitSimple_compound_wildcard*: The structure of compound statements is highly dependent on the grammar. It is easy in Python, because all compound statements directly expand into the *block* rule. However, that is not the case in Java and we cannot assume it will be the case in other languages. It has also been left as an abstract method in the superclass.
- Parents of list wildcard: The grammar rules that can be expanded into the list wildcard are the parents of the list wildcards. All parents of the list wildcards should have a corresponding visit method responsible for calling the method *handle_empty_list*. In Jattern, the possible parent nodes of the list wildcards are the $FormalParameters$, $ExpressionList$ and $Arguments$. In Pyttern, they are $Parameters$, $Varargslist$, $Argument$. In another language, they are bound to be different, therefore there is no way to generalise it.

6.1.4 Class diagrams

Processors

The class diagram for the processors can be seen in Figure 6.1. The Python processor (for Pyttern) overrides more methods than the Java processor (for Jattern), as it uses caching which the Java processor does not. To add a new processor, only the three methods shown in the Java processor are needed.

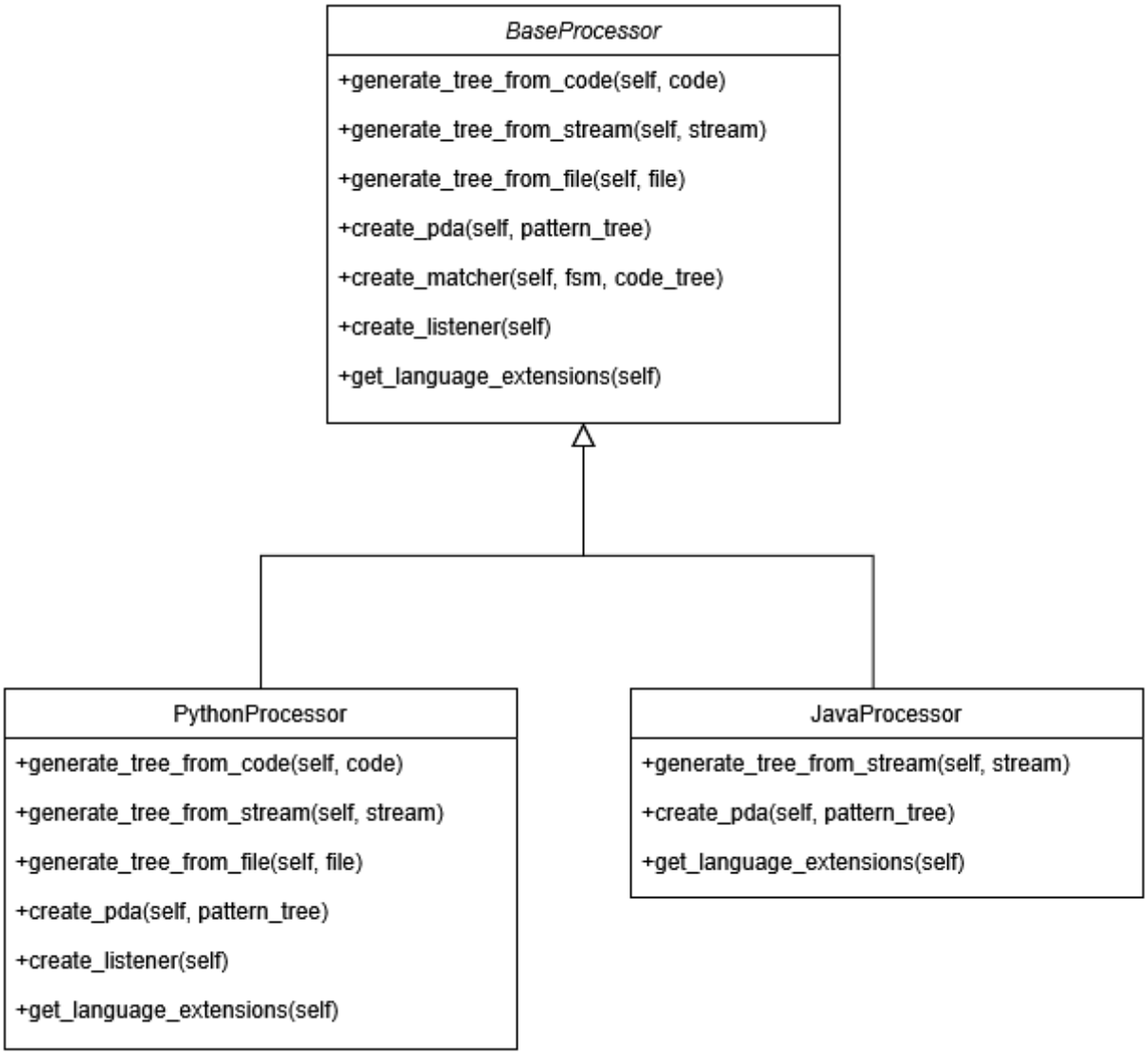

Figure 6.1: Processor class diagram after refactoring

Tree pruners

The class diagram for the tree pruners can be seen in Figure 6.2. As explained previously, the tree pruners inherit from both the ANTLR parser visitor for the corresponding language and the generic tree pruner. The generic tree pruner contains most of the logic, while the overridden methods correspond to special cases that should be pruned differently.

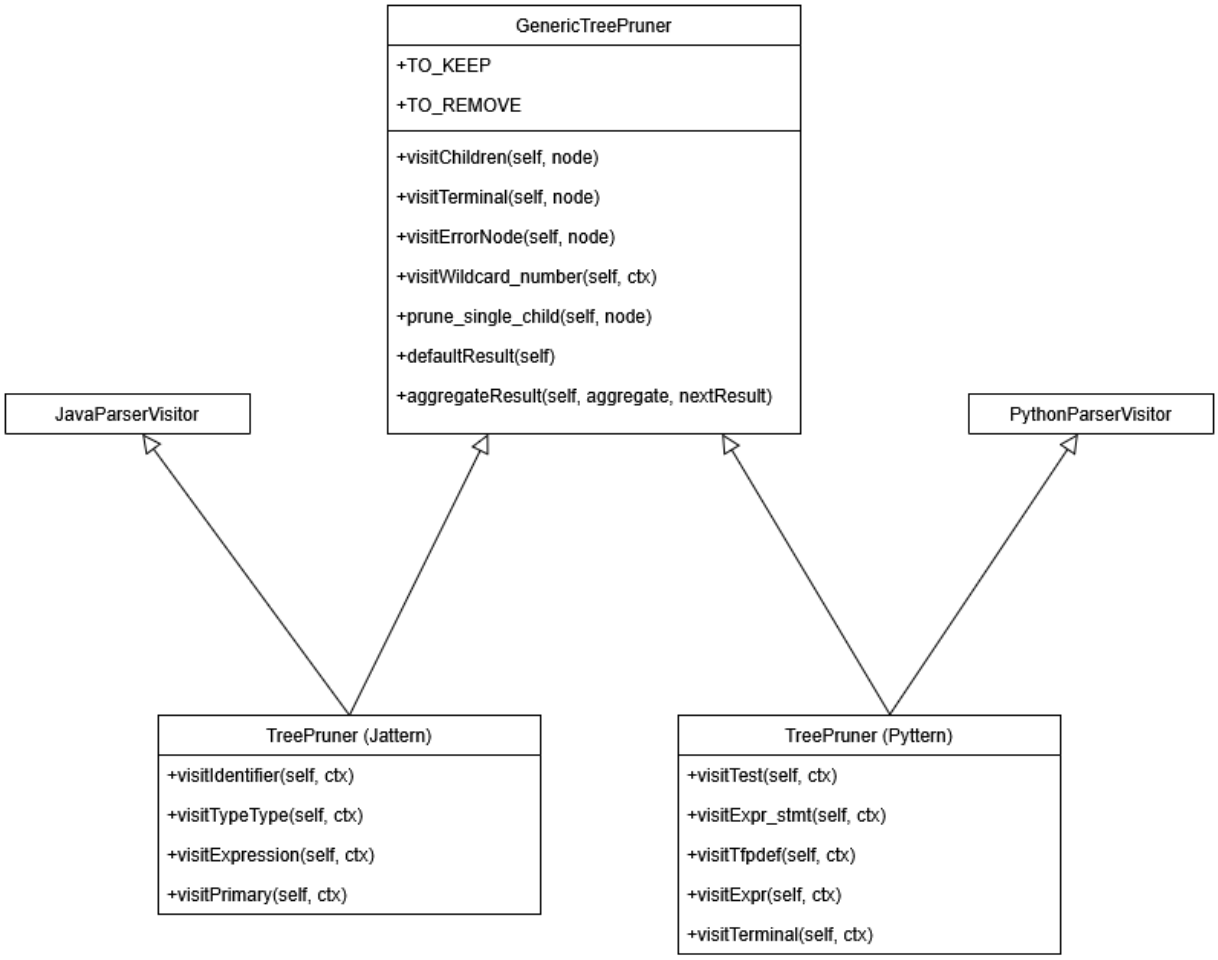

Figure 6.2: Tree pruner class diagram after refactoring

PDA

The class diagram for the classes creating the PDA can be seen in Figure 6.3, where colours have been added to make it more easily understandable.

- The purple attributes are the language-dependent attributes. They are determined during the initialisation and provided by the subclass.
- The green methods are the methods responsible for the wildcards which are not language-dependent. We can see that most of the wildcards do not depend on the language.
- The blue methods are the methods that visit nodes that are parent to the list wildcard and only call *handle_empty_list*.

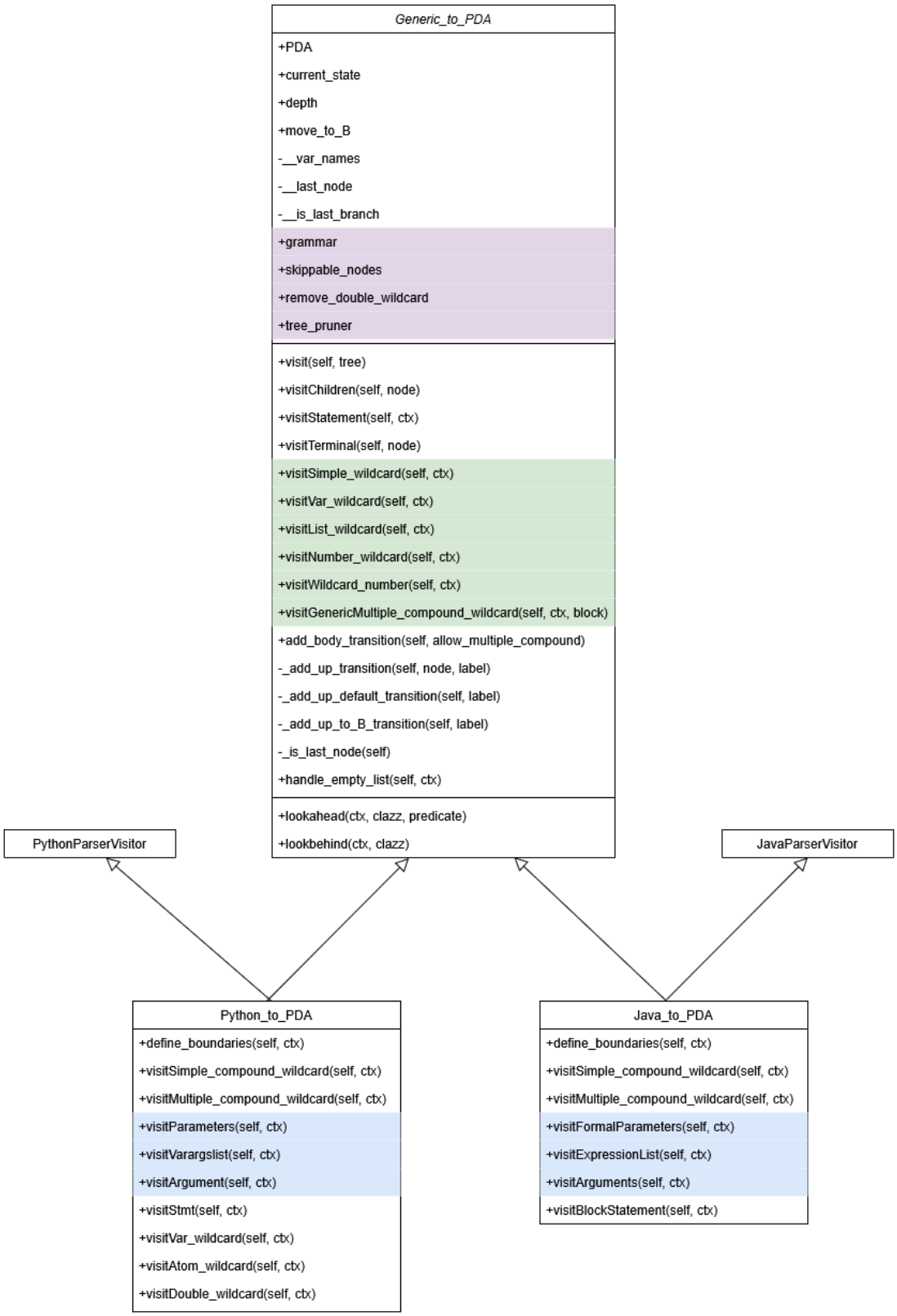

Figure 6.3: PDA class diagram after refactoring

Overview

The class diagram showing the interactions between the different components can be seen in Figure 6.4, although $JavaParserVisitor$ and $PythonParserVisitor$ have been omitted from it to keep the class diagram easily understandable.

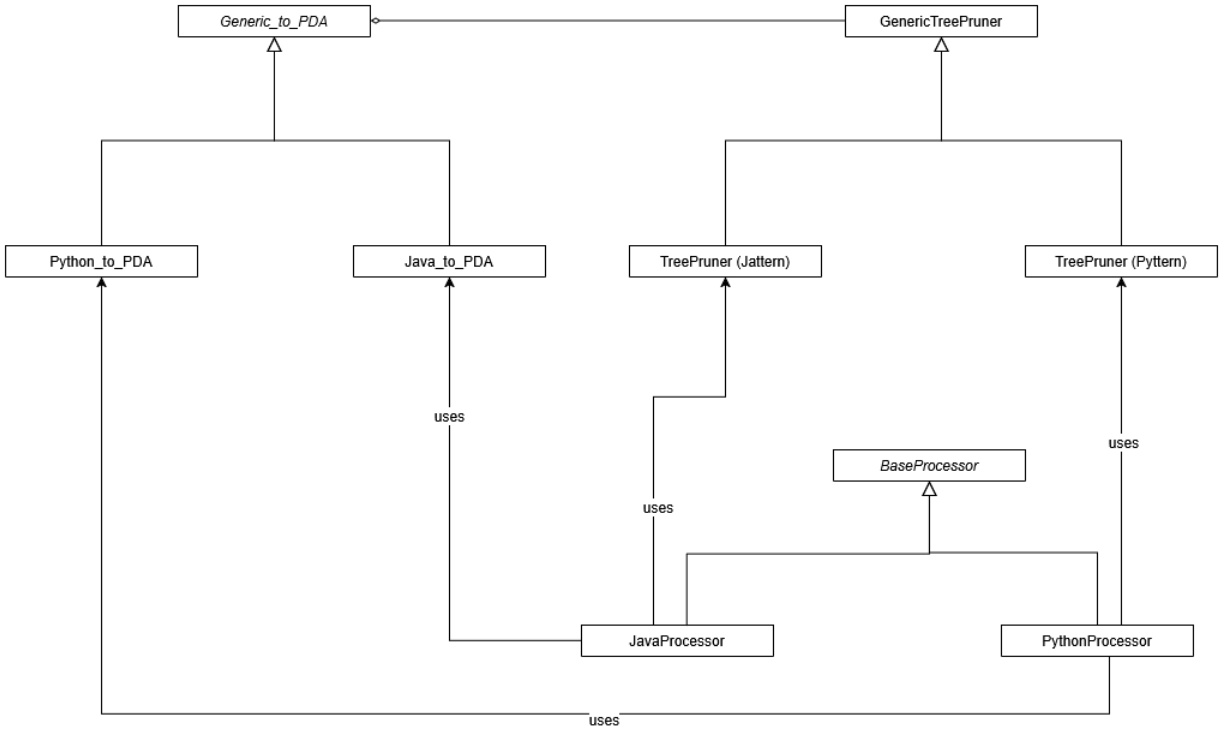

Figure 6.4: Class diagram overview after refactoring

6.1.5 Refactoring conclusion

After refactoring, the tree pruners files over two times smaller and the PDA generators are two times smaller for Pyttern and three times smaller for Jattern. This shows that the refactoring described in this chapter was efficient at reducing the number of language-dependent components. Since the number of language-dependent attributes and methods has been greatly reduced, adding other languages to Pyttern will be easier.

6.2 Guide to add a new language

Taking inspiration from how we ported Pyttern from Python to Java, in this section, we explain how we could adapt our new generalised version of Pyttern to new languages.

6.2.1 Setting up ANTLR

The first step would be to set up ANTLR's lexer and parser for that language. Next, we need to add all the wildcard rules to that lexer and parser.

- The wildcard symbol in the lexer should be a symbol that is not already in use in the grammar of the language. The lexer rules should include the wildcard followed by a space, the named wildcard (referred to as $VAR\_WILDCARD$ in the grammar), and the wildcard symbol alone, in that order specifically, as shown in Listing 6.3. As long as the relative position of the rules of Listing 6.3 is respected, they can be added anywhere in the lexer rules since the wildcard symbol cannot be recognised by any other lexer rule.
- The simple and named wildcards should be added as alternatives in the identifier rule of the grammar of the new language. If there are types in that language, the simple and named wildcards should be added to the corresponding type grammar rule.
- The simple wildcard, the named wildcards, and the contains wildcard should be added together as alternatives in the expression rule of the new language's grammar.
- The list and number wildcards should be added to the rules that define arrays and method parameters (both in their definition and when calling them) as well as other constructs that have lists of values or variables.
- Statements that add an indentation level should be grouped together as a new compound statement rule. The simple compound and multiple compound wildcards should be added as alternatives to this new rule.
- The composite statement wildcard should be added as an alternative to the statement rule. The statement wildcard is a composite which corresponds to wildcards that are statements by themselves. It is expanded as either the simple wildcard, the named wildcard, the list wildcard, or the contains wildcard.

The parser grammar rules for Jattern can be found in Appendix A.3, with the rules defining the wildcards being at the end. It is recommended to copy the notation and adapt it as needed to better fit the language that is being ported.

Listing 6.3: Lexer rules

```
// Rules for wildcards
fragment WILDCARD_SYMBOL: '#'; // replace the # with the chosen symbol

WILDCARD_SPACE: WILDCARD_SYMBOL WS+;
VAR_WILDCARD: WILDCARD_SYMBOL IDENTIFIER;
WILDCARD : WILDCARD_SYMBOL;
```

6.2.2 Creating the PDA

To create the PDA, after inheriting from *Generic_to_PDA* and the ANTLR visitor, the following methods and attributes need to be defined:

- **define_boundaries**: This is the trickiest part of the PDA. Anything that can contain an unlimited number of children (such as block nodes, list wildcard nodes, etc.) should have its higher boundary set to infinity. It is recommended to look into the special cases of Pyttern and Jattern to see if they hold in this new language.
- **visitSimple_compound_wildcard**: If all compound statements use the same *block* rule, it should be defined similarly to what was done in Pyttern; otherwise it should be done like in Jattern.
- **visitMultiple_compound_wildcard**: The body of the wildcard, i.e., the node containing all statements that are inside the wildcard, should be retrieved and stored in the variable *body*. The superclass then handles the rest after being called through *return super().visitGenericMultiple_compound_wildcard(ctx, body)*
- Nodes that can have the list wildcard as a child (such as a method parameters node or an array node) should have a visit method which executes *return self.handle_empty_list(ctx)*
- The attribute **grammar** of the PDA should be defined as the ANTLR parser for that language.
- The attribute **skippable_nodes** of the PDA should be defined as a list of statement nodes that can be skipped as part of the relaxed matching.
- The attribute **remove_double_wildcard** of the PDA should be defined as a list of the list wildcards that exist in this language. By default, it is only the list wildcard itself. Depending on the language, there may be a need to create and add another list wildcard to the list, which matches at least one element, as in Pyttern.
- The attribute **tree_pruner** of the PDA should be defined as the tree pruner for that language; see section below for more details.

6.2.3 Creating the Tree Pruner

To create the tree pruner, after inheriting from *GenericTreePruner* and the ANTLR visitor, the following methods and attributes need to be defined:

- **TO_REMOVE** should be a string that contains the characters that should be pruned from the AST. The string "() :, .}{;" is recommended as default as the characters contained within are likely to be relevant only during the parsing phase. If one of those characters is important in the new language and should not be pruned, it can be removed from the string.

- **TO_KEEP** should be a list that contains all the types of nodes that should stop the pruning. The terminal nodes, identifier node, and the composite expression wildcard are recommended as default.
- The method to visit an identifier should be implemented to prune the identifier node if and only if its only child is the simple or named wildcard. The same applies to the type node, if it exists in that language. This is to ensure the depth can be properly calculated by the named wildcard, as explained in the *matching issues* section of chapter 5.5.2.

Depending on the language, it may be beneficial to prune other nodes. There is no universal solution to determine what should be pruned. This is something that should be analysed with care. For example, for Java we had to prune the comma, but in another language (such as JavaScript) it might be an operator.

6.2.4 Creating the Processor

The processor is the class that brings it all together. It is responsible for calling the parser, calling the tree pruner, and starting the conversion to a PDA. After inheriting from *BaseProcessor*, the following methods need to be defined:

- **generate_tree_from_stream**: This method should call the lexer, parser and pruner for that language. An example can be seen in Listing 6.4.
- **create_pda**: This method should call the class responsible for the conversion from AST to a pushdown automaton. As shown in Listing 6.4, it consists only of calling the visit method of the corresponding PDA converter.
- **get_language_extensions**: This method should return a list corresponding to the file extensions that are supported for this language. As shown in Listing 6.4, in Jattern this would be "java", "jav" and "jat", with the last one being the file extension for Jattern patterns.

Listing 6.4: Java Processor

```
class JavaProcessor(BaseProcessor):
    def generate_tree_from_stream(self, stream):
        lexer = JavaLexer(stream)
        stream = CommonTokenStream(lexer)
        java_parser = JavaParser(stream)

        error = io.StringIO()
        java_parser.removeErrorListeners()
        error_listener = Python3ErrorListener(error)
        java_parser.addErrorListener(error_listener)
```

```
        tree = java_parser.compilationUnit()

        pruned_tree = TreePruner().visit(tree)
        return pruned_tree

    def create_pda(self, pattern_tree):
        return Java_to_PDA().visit(pattern_tree)

    def get_language_extensions(self):
        return ["java", "jav", "jat"]
```

6.2.5 Guide conclusion

Throughout this guide, we saw the simplified process for porting a new language to Pyttern. The main hotspots are the abstract methods of the converter to PDA and the ANTLR parser grammar. The tree pruner requires careful consideration to determine its attributes, but creating it is much simpler than it was previously.

Chapter 7

Validation

In this chapter, we validate the implementation of Jattern and the refactoring of both Pyttern and Jattern. We verify that Jattern follows the specifications described in the previous chapters and that Pyttern's behaviour was not modified as a result of the refactoring.

7.1 Method

To ensure none of the modifications made when implementing Jattern caused any ambiguity and to verify that all wildcards were properly implemented, we created many unit tests for Jattern. Those tests were implemented in parallel with our implementation of Jattern, often using test-driven development.

Due to the large number of tests required to ensure sufficient coverage, we implemented a system that detects test files representing programs, and queries and their corresponding expected matching result automatically, as opposed to having to write specific unit tests for each case. It starts in a given folder, then recursively visits each subfolder using *os.walk()*.

For each subfolder it visits, it verifies whether a *pattern.jat* file is present. If there is one, then it looks for test cases. All java files whose name start with "ok_" are code files that should match the pattern. All java files whose name start with "ko_" are code files that should **not** match the pattern.

For example, in Figure 7.1, the file *pattern.jat* must match with *ok_expression.java* and *ok_expression_2.java*, and must not match with *ko_expression.java*, *ko_expression_2.java* or *ko_expression_3.java*.

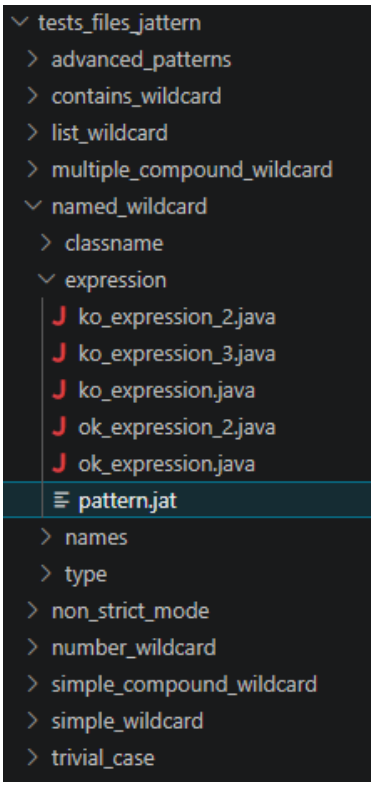

Figure 7.1: Test folder example, highlighting pattern's location

The pattern of the file *pattern.jat* can be seen in Listing 7.1, which describes a pattern where the *compute* method returns the addition of the two given parameters.

Listing 7.1: Simple addition pattern

```
class Calculator {
    public static int compute(int #var1, int #var2) {
        return #var1 + #var2 ;
    }
}
```

The content of the file *ok_expression.java* can be seen in Listing 7.2. Since the program matches the pattern and the file name starts with "ok_", the test passes successfully.

Listing 7.2: Matching addition program

```
class Calculator {
    public static int compute(int a, int b) {
        return a + b;
    }
}
```

The content of the file *ko_expression_2.java* can be seen in Listing 7.3. The program does **not** match the pattern because the constraints of the wildcard named *var2* are not

respected. The lack of a match was expected because the file name starts by "ko_", so the test passes successfully.

Listing 7.3: Non matching addition program

```
class Calculator {
    public static int compute(int a, int b) {
        int c = 0;
        return a + c; // should be a+b
    }
}
```

This small but efficient detection system allows us to easily add more test cases without needing to manually add more *assert* statements. This testing approach is very generic and can therefore straightforwardly be applied to other languages as well.

7.2 Writing tests

To determine what tests to write, we started by considering all the basic use cases. For example, for the simple wildcard, we have tests where the simple wildcard replaces an identifier, tests where it replaces an expression, and tests where it replaces a type. When relevant, we also added negative test cases.

When we uncovered a bug through the use of tests, we added additional tests to better understand the origin of the bug. When we uncovered a bug without the use of tests, we also added tests that covered it, to ensure it would not go undetected again.

Finally, once all wildcards had been implemented, we created more advanced test cases that combined several of them. Some were directly ported from Pyttern, such as the "misplaced return" case (detecting a return statement inside a compound statement). Others were written as examples of how real users might use Jattern to detect coding flaws (such as a hard-coded list) or structural patterns (such as the accumulator pattern).

7.3 Advanced tests

In this section, we showcase and explain some of the advanced test cases mentioned previously.

7.3.1 Accumulator

In Listing 7.4, we show the pattern for an accumulator variable, which will be matched against Listing 7.5. The pattern is heavily abstracted, as it will first look for a class with any name, then a method with any name, any return type, and any number of arguments.

This already showcases how patterns can easily be very generic. In the case of Listing 7.5, we will enter the class *Calculator* and the method *pi*.

Once inside the method definition, we look for a variable definition and bind the identifier to the *accumulator* named wildcard. In the case of Listing 7.5, it will bind the identifier x. Then, we look for a compound statement (such as a for or while loop) in which the accumulator variable is updated based on an expression that uses the accumulator variable. Finally, we look for a return statement in which the accumulator variable is used as part of the return expression. Since Listing 7.5 respects all of the aforementioned steps, it results in a match.

Listing 7.4: Accumulator pattern

```
// any class & method that initialise a variable,
// update it in an indent then use it in the return statement
class # {
    # #(#*) {
        # #accumulator = #;
        #{
            #accumulator = #< #accumulator >;
        }
        return #< #accumulator >;
    }
}
```

Listing 7.5: Accumulator program

```
import java.lang.Math;

class Calculator {
    double pi(int n) {
        double x = 0;
        for (int i = 0; i < n+1; i++) {
            x = x + (Math.pow((-1),i))/(2*i+1);
        }
        return 4*x;
    }
}
```

On the other hand, for Listing 7.6, the last part of the pattern is not respected. While we do return a value, x is not contained within the return expression. Since the contains wildcard is not respected, the matching fails.

Listing 7.6: Invalid accumulator program

```
import java.lang.Math;
```

```
class Calculator {
    double pi(int n) {
        double x = 0;
        for (int i = 0; i < n+1; i++) {
            x = x + (Math.pow((-1),i))/(2*i+1);
        }
        return Math.PI; // x isn't used in the return
    }
}
```

This advanced test case is a good example of how the wildcards can be combined to detect structural patterns.

7.3.2 Hard-coded list

Listing 7.7 shows the pattern corresponding to the design flaw of a hard-coded list. The pattern is again heavily abstracted as we use the wildcards to abstract the class name, method name, and method arguments.

Listing 7.7: Hard-coded list pattern

```
// any class & method which defines
// a variable containing at least 3 elements
class # {
    # #(#*) {
        # #[] = new #[]{ #(3,) };
    }
}
```

When matched against Listing 7.8, the PDA will enter the class, then the method and its body. Once inside, it will look for a local variable definition in which an array with at least three values is initialised. Since we define a hard-coded list of primes, the PDA will match.

Listing 7.8: Flawed isPrime program

```
class IsPrime {
    boolean isPrime(int n) {
        // hardcoded list of primes
        Integer primes[] = new Integer[]{2,3,5,7};

        // return true if hardcoded list contains n, false otherwise
        for (int i = 0; i < primes.length; i++) {
            if (primes[i] == n) {
                return true;
            }
        }
```

```
        return false;
    }
}
```

Let us consider another similar program in Listing 7.9. In this program, the developer created a list to cover edge cases, and the rest of the function is correct. Here, the PDA will not match because the list being defined contains less than three elements.

Listing 7.9: Correct isPrime program

```
// implementation of isPrime without hardcoded prime list
class IsPrime {
    boolean isPrime(int n) {
        Integer edge_cases[] = new Integer[]{0,1}; // definition of list,
        // but no match because less than 3 elements
        if (edge_cases[0] == n || edge_cases[1] == n) return false;

        for (int i = 2; i < n; i++) {
            if (n % i == 0) {
                return false;
            }
        }

        return true;
    }
}
```

This advanced test case is a good example of how the wildcards can be combined together to detect coding flaws.

7.3.3 Observer design pattern

Listing 7.10 shows the pattern corresponding to the observer design pattern [17]. It is a complex pattern that requires a class to have a field attribute that is an array list initialised in the constructor method. Then, it requires an *attach* method that updates the list and a *notify* method that calls the *update* method (which is assumed to be the method of the subscribers).

Listing 7.10: Observer design pattern

```
class #name {
    # #observerList;

    // constructor method
    #name(#*) {
        #observerList = new ArrayList<>();
```

```
    }

    void attach(# #observer) {
        #observerList.add(#observer);
    }

    void notify(#*) {
        #{
            #<#observerList>.update(#*);
        }
    }
}
```

A program implementing the observer design pattern is included in Listing 7.11 and successfully matches the pattern from Listing 7.10.

Listing 7.11: Observer program

```
import java.util.ArrayList;
import java.util.Observer;

class EventManager {
    ArrayList<Observer> observerList;

    // constructor method
    EventManager() {
        observerList = new ArrayList<>();
    }

    void attach(Observer observer) {
        observerList.add(observer);
    }

    void notify(String message) {
        for (int i = 0; i < observerList.size(); i++) {
            observerList.get(i).update(null, message);
        }
    }
}
```

This advanced test case is a good example of how the wildcards can be combined to detect design patterns.

7.4 Results

A total of 202 tests were implemented for Jattern. They are split into the following categories:

- **Trivial cases:** when the pattern is exactly the same as the code file. There are 10 test cases in this category.
- **Non-strict (relaxed) cases:** when the pattern is contained within the code file, but the code file contains more statements than just the pattern. There are 27 test cases in this category.
- **Wildcards:** each wildcard has its own attached tests in a corresponding subfolder. Each possible use has been covered. This category is the main one, with a total of 139 test cases.
- **Advanced:** wildcards combined to create more complex and expressive patterns, illustrating how real users might use Jattern. There are 26 test cases in this category.

All unit tests implemented pass successfully. The Pyttern tests were left untouched and all pass as well.

Coverage report: 96%

Files | Functions | Classes

coverage.py v7.14.1, created at 2026-05-30 13:34 +0200

File ▲	statements	missing	excluded	coverage
pyttern\pytternfsm__init__.py	0	0	0	100%
pyttern\pytternfsm\generic_to_pda.py	248	9	0	96%
pyttern\pytternfsm\generic_tree_pruner.py	38	1	0	97%
pyttern\pytternfsm\java__init__.py	0	0	0	100%
pyttern\pytternfsm\java\java_to_pda.py	95	0	0	100%
pyttern\pytternfsm\java\java_visitor.py	0	0	0	100%
pyttern\pytternfsm\java\tree_pruner.py	21	0	0	100%
pyttern\pytternfsm\python__init__.py	0	0	0	100%
pyttern\pytternfsm\python\match_set.py	19	1	0	95%
pyttern\pytternfsm\python\python_to_pda.py	146	10	0	93%
pyttern\pytternfsm\python\tree_pruner.py	21	0	0	100%
Total	588	21	0	96%

Figure 7.2: Coverage report

Using pytest's coverage tool [18], we created the coverage report shown in Figure 7.2. The coverage reaches a respectable 96%. Most of the missing lines correspond to errors, such as the multiple compound wildcard with an empty body.

7.5 Discussion

The objective of the units tests of Jattern was to cover the most common usages of wildcards. All relevant tests from Pyttern have been adapted for Jattern in addition to the original tests. Since they all pass successfully and cover nearly all of the lines of code of the various components of Jattern (with the exception of errors), it seems safe to conclude that Jattern has been implemented in a semantically valid way.

Additionally, since the results of the Pyttern tests remained successful throughout the refactoring, it is reasonable to conclude that the refactoring correctly preserved the behaviour of Pyttern.

Chapter 8

Conclusion

Throughout this thesis, we have explained how Jattern, a Java extension of Pyttern, was implemented. We discussed the new obstacles unique to Java, and highlighted expressiveness that was impossible in Pyttern. This successfully completed the first objective of this thesis.

We refactored both Pyttern and Jattern, causing the language-dependent parts to be significantly reduced in size. This more modular approach will make it easier to extend Pyttern to future languages as less code and understanding will be required, thereby completing the second objective of this thesis.

Additionally, we included a guide explaining all the steps required to extend Pyttern to a new language. This concludes the third objective of this thesis.

A thorough series of tests has validated the Jattern implementation by ensuring the implementation was correct and that the refactoring preserved the behaviour. All of the objectives of the thesis have thus been met.

Future work on the matter could extend Jattern to include macros, a form of sub-patterns recently added to Pyttern. Additionally, other wildcards could be implemented, such as a wildcard unique to Java to match only primitive types.

Another interesting avenue for future work would be to extend Pyttern to support a third language. This third language could be one close to Python and Java to leverage the modular approach, or instead a very different one to verify if the generalisation upholds.

Appendix A

Jattern code

A.1 Jattern repository before refactoring

https://github.com/Lucie4914/Pyttern/tree/de96eae923cb84604da7cf5b5d2a54ecafbafa9d

A.2 Jattern repository after refactoring

https://github.com/Lucie4914/Pyttern/tree/40ab7a853e6a6e7d86372e91d0ca1340b0e7f22f

A.3 Jattern Grammar

```
/*
 [The "BSD licence"]
 Copyright (c) 2013 Terence Parr, Sam Harwell
 Copyright (c) 2017 Ivan Kochurkin (upgrade to Java 8)
 Copyright (c) 2021 Michał Lorek (upgrade to Java 11)
 Copyright (c) 2022 Michał Lorek (upgrade to Java 17)
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:
 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote products
    derived from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ''AS IS'' AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
```

```
 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

// $antlr-format alignTrailingComments true, columnLimit 150, minEmptyLines
    1, maxEmptyLinesToKeep 1, reflowComments false, useTab false
// $antlr-format allowShortRulesOnASingleLine false,
    allowShortBlocksOnASingleLine true, alignSemicolons hanging, alignColons
    hanging

parser grammar JavaParser;

options {
    tokenVocab = JavaLexer;
}

compilationUnit
    : packageDeclaration? (importDeclaration | ';')* (typeDeclaration | ';')
      *
    | moduleDeclaration EOF
    ;

packageDeclaration
    : annotation* PACKAGE qualifiedName ';'
    ;

importDeclaration
    : IMPORT STATIC? qualifiedName ('.' '*')? ';'
    ;

typeDeclaration
    : classOrInterfaceModifier* (
        classDeclaration
        | enumDeclaration
        | interfaceDeclaration
        | annotationTypeDeclaration
        | recordDeclaration
    )
    | multiple_compound_wildcard
    ;

modifier
    : classOrInterfaceModifier
    | NATIVE
    | SYNCHRONIZED
    | TRANSIENT
```

```
    | VOLATILE
    ;

classOrInterfaceModifier
    : annotation
    | PUBLIC
    | PROTECTED
    | PRIVATE
    | STATIC
    | ABSTRACT
    | FINAL // FINAL for class only -- does not apply to interfaces
    | STRICTFP
    | SEALED     // Java17
    | NON_SEALED // Java17
    ;

variableModifier
    : FINAL
    | annotation
    ;

classDeclaration
    : CLASS identifier typeParameters? (EXTENDS typeType)? (IMPLEMENTS
      typeList)? (
       PERMITS typeList
    )? // Java17
    classBody
    ;

typeParameters
    : '<' typeParameter (',' typeParameter)* '>'
    ;

typeParameter
    : annotation* identifier (EXTENDS annotation* typeBound)?
    ;

typeBound
    : typeType ('&' typeType)*
    ;

enumDeclaration
    : ENUM identifier (IMPLEMENTS typeList)? '{' enumConstants? ','?
      enumBodyDeclarations? '}'
    ;

enumConstants
    : enumConstant (',' enumConstant)*
    ;

enumConstant
```

```
    : annotation* identifier arguments? classBody?
    ;

enumBodyDeclarations
    : ';' classBodyDeclaration*
    ;

interfaceDeclaration
    : INTERFACE identifier typeParameters? (EXTENDS typeList)? (PERMITS
      typeList)? interfaceBody
    ;

classBody
    : '{' classBodyDeclaration* '}'
    ;

interfaceBody
    : '{' interfaceBodyDeclaration* '}'
    ;

classBodyDeclaration
    : ';'
    | STATIC? block
    | modifier* memberDeclaration
    ;

memberDeclaration
    : recordDeclaration //Java17
    | methodDeclaration
    | genericMethodDeclaration
    | fieldDeclaration
    | constructorDeclaration
    | genericConstructorDeclaration
    | interfaceDeclaration
    | annotationTypeDeclaration
    | classDeclaration
    | enumDeclaration
    ;

/* We use rule this even for void methods which cannot have [] after
   parameters.
   This simplifies grammar and we can consider void to be a type, which
   renders the [] matching as a context-sensitive issue or a semantic check
   for invalid return type after parsing.
 */
methodDeclaration
    : typeTypeOrVoid identifier formalParameters ('[' ']')* (THROWS
      qualifiedNameList)? methodBody
    ;

methodBody
```

```
    : block
    | ';'
    ;

typeTypeOrVoid
    : typeType
    | VOID
    ;

genericMethodDeclaration
    : typeParameters methodDeclaration
    ;

genericConstructorDeclaration
    : typeParameters constructorDeclaration
    ;

constructorDeclaration
    : identifier formalParameters (THROWS qualifiedNameList)?
      constructorBody = block
    ;

compactConstructorDeclaration
    : modifier* identifier constructorBody = block
    ;

fieldDeclaration
    : typeType variableDeclarators ';'
    ;

interfaceBodyDeclaration
    : modifier* interfaceMemberDeclaration
    | ';'
    ;

interfaceMemberDeclaration
    : recordDeclaration // Java17
    | constDeclaration
    | interfaceMethodDeclaration
    | genericInterfaceMethodDeclaration
    | interfaceDeclaration
    | annotationTypeDeclaration
    | classDeclaration
    | enumDeclaration
    ;

constDeclaration
    : typeType constantDeclarator (',' constantDeclarator)* ';'
    ;

constantDeclarator
```

```
    : identifier ('[' ']')* '=' variableInitializer
    ;

// Early versions of Java allows brackets after the method name, eg.
// public int[] return2DArray() [] { ... }
// is the same as
// public int[][] return2DArray() { ... }
interfaceMethodDeclaration
    : interfaceMethodModifier* interfaceCommonBodyDeclaration
    ;

// Java8
interfaceMethodModifier
    : annotation
    | PUBLIC
    | ABSTRACT
    | DEFAULT
    | STATIC
    | STRICTFP
    ;

genericInterfaceMethodDeclaration
    : interfaceMethodModifier* typeParameters interfaceCommonBodyDeclaration
    ;

interfaceCommonBodyDeclaration
    : annotation* typeTypeOrVoid identifier formalParameters ('[' ']')* (
       THROWS qualifiedNameList)? methodBody
    ;

variableDeclarators
    : variableDeclarator (',' variableDeclarator)*
    ;

variableDeclarator
    : variableDeclaratorId ('=' variableInitializer)?
    ;

variableDeclaratorId
    : simple_wildcard
    | var_wildcard
    | identifier ('[' ']')*
    ;

variableInitializer
    : arrayInitializer
    | expression
    ;

arrayInitializer
```

```
    : '{' ((list_wildcard | number_wildcard | variableInitializer) (',' (
      list_wildcard | number_wildcard | variableInitializer))* ','?)? '}'
    ;

classOrInterfaceType
    : (identifier typeArguments? '.')* typeIdentifier typeArguments?
    ;

typeArgument
    : typeType
    | annotation* '?' ((EXTENDS | SUPER) typeType)?
    ;

qualifiedNameList
    : qualifiedName (',' qualifiedName)*
    ;

formalParameters
    : '(' (
        receiverParameter?
        | receiverParameter (',' formalParameterList)?
        | formalParameterList?
    ) ')'
    ;

receiverParameter
    : typeType (identifier '.')* THIS
    ;

formalParameterList
    : (list_wildcard | number_wildcard | formalParameter) (',' (
      list_wildcard | number_wildcard | formalParameter))* (','
      lastFormalParameter)?
    | lastFormalParameter
    ;

formalParameter
    : variableModifier* typeType variableDeclaratorId
    ;

lastFormalParameter
    : variableModifier* typeType annotation* '...' variableDeclaratorId
    ;

// local variable type inference
lambdaLVTIList
    : lambdaLVTIParameter (',' lambdaLVTIParameter)*
    ;

lambdaLVTIParameter
    : variableModifier* VAR identifier
```

```
    ;

qualifiedName
    : identifier ('.' identifier)*
    ;

literal
    : integerLiteral
    | floatLiteral
    | CHAR_LITERAL
    | STRING_LITERAL
    | BOOL_LITERAL
    | NULL_LITERAL
    | TEXT_BLOCK // Java17
    ;

integerLiteral
    : DECIMAL_LITERAL
    | HEX_LITERAL
    | OCT_LITERAL
    | BINARY_LITERAL
    ;

floatLiteral
    : FLOAT_LITERAL
    | HEX_FLOAT_LITERAL
    ;

// ANNOTATIONS
altAnnotationQualifiedName
    : (identifier DOT)* '@' identifier
    ;

annotation
    : ('@' qualifiedName | altAnnotationQualifiedName) (
        '(' ( elementValuePairs | elementValue)? ')'
    )?
    ;

elementValuePairs
    : elementValuePair (',' elementValuePair)*
    ;

elementValuePair
    : identifier '=' elementValue
    ;

elementValue
    : expression
    | annotation
    | elementValueArrayInitializer
```

```
    ;

elementValueArrayInitializer
    : '{' (elementValue (',' elementValue)*)? ','? '}'
    ;

annotationTypeDeclaration
    : '@' INTERFACE identifier annotationTypeBody
    ;

annotationTypeBody
    : '{' annotationTypeElementDeclaration* '}'
    ;

annotationTypeElementDeclaration
    : modifier* annotationTypeElementRest
    | ';' // this is not allowed by the grammar, but apparently allowed by
      the actual compiler
    ;

annotationTypeElementRest
    : typeType annotationMethodOrConstantRest ';'
    | classDeclaration ';'?
    | interfaceDeclaration ';'?
    | enumDeclaration ';'?
    | annotationTypeDeclaration ';'?
    | recordDeclaration ';'? // Java17
    ;

annotationMethodOrConstantRest
    : annotationMethodRest
    | annotationConstantRest
    ;

annotationMethodRest
    : identifier '(' ')' defaultValue?
    ;

annotationConstantRest
    : variableDeclarators
    ;

defaultValue
    : DEFAULT elementValue
    ;

// MODULES - Java9

moduleDeclaration
    : OPEN? MODULE qualifiedName moduleBody
    ;
```

```
moduleBody
    : '{' moduleDirective* '}'
    ;

moduleDirective
    : REQUIRES requiresModifier* qualifiedName ';'
    | EXPORTS qualifiedName (TO qualifiedName)? ';'
    | OPENS qualifiedName (TO qualifiedName)? ';'
    | USES qualifiedName ';'
    | PROVIDES qualifiedName WITH qualifiedName ';'
    ;

requiresModifier
    : TRANSITIVE
    | STATIC
    ;

// RECORDS - Java 17

recordDeclaration
    : RECORD identifier typeParameters? recordHeader (IMPLEMENTS typeList)?
      recordBody
    ;

recordHeader
    : '(' recordComponentList? ')'
    ;

recordComponentList
    : recordComponent (',' recordComponent)*
    ;

recordComponent
    : typeType identifier
    ;

recordBody
    : '{' (classBodyDeclaration | compactConstructorDeclaration)* '}'
    ;

// STATEMENTS / BLOCKS

block
    : '{' blockStatement* '}'
    ;

blockStatement
    : localVariableDeclaration ';'
    | localTypeDeclaration
    | statement
```

```
    ;

localVariableDeclaration
    : variableModifier* (VAR identifier '=' expression | typeType
      variableDeclarators)
    ;

identifier
    : IDENTIFIER
    | MODULE
    | OPEN
    | REQUIRES
    | EXPORTS
    | OPENS
    | TO
    | USES
    | PROVIDES
    | WITH
    | TRANSITIVE
    | YIELD
    | SEALED
    | PERMITS
    | RECORD
    | VAR
    | simple_wildcard
    | var_wildcard
    ;

typeIdentifier // Identifiers that are not restricted for type declarations
    : IDENTIFIER
    | MODULE
    | OPEN
    | REQUIRES
    | EXPORTS
    | OPENS
    | TO
    | USES
    | PROVIDES
    | WITH
    | TRANSITIVE
    | SEALED
    | PERMITS
    | RECORD
    ;

localTypeDeclaration
    : classOrInterfaceModifier* (classDeclaration | interfaceDeclaration |
      recordDeclaration)
    ;

statement
```

```
    : compound_stmt
    | ASSERT expression (':' expression)? ';'
    | RETURN expression? ';'
    | THROW expression ';'
    | BREAK identifier? ';'
    | CONTINUE identifier? ';'
    | YIELD expression ';' // Java17
    | SEMI
    | statementExpression = expression ';'
    | switchExpression ';'? // Java17
    | stmt_wildcard ';'
    ;

compound_stmt
    : compound_wildcard
    | blockLabel = block
    | IF parExpression statement (ELSE statement)?
    | FOR '(' forControl ')' statement
    | WHILE parExpression statement
    | DO statement WHILE parExpression ';'
    | TRY block (catchClause+ finallyBlock? | finallyBlock)
    | TRY resourceSpecification block catchClause* finallyBlock?
    | SWITCH parExpression '{' switchBlockStatementGroup* switchLabel* '}'
    | SYNCHRONIZED parExpression block
    | identifierLabel = identifier ':' statement
    ;

catchClause
    : CATCH '(' variableModifier* catchType identifier ')' block
    ;

catchType
    : qualifiedName ('|' qualifiedName)*
    ;

finallyBlock
    : FINALLY block
    ;

resourceSpecification
    : '(' resources ';'? ')'
    ;

resources
    : resource (';' resource)*
    ;

resource
    : variableModifier* (classOrInterfaceType variableDeclaratorId | VAR
      identifier) '=' expression
    | qualifiedName
```

```
    ;

/** Matches cases then statements, both of which are mandatory.
 *  To handle empty cases at the end, we add switchLabel* to statement.
 */
switchBlockStatementGroup
    : switchLabel+ blockStatement+
    ;

switchLabel
    : CASE (
        constantExpression = expression
        | enumConstantName = IDENTIFIER
        | typeType varName = identifier
    ) ':'
    | DEFAULT ':'
    ;

forControl
    : enhancedForControl
    | forInit? ';' expression? ';' forUpdate = expressionList?
    ;

forInit
    : localVariableDeclaration
    | expressionList
    ;

enhancedForControl
    : variableModifier* (typeType | VAR) variableDeclaratorId ':' expression
    ;

// EXPRESSIONS

parExpression
    : '(' expression ')'
    ;

expressionList
    : (list_wildcard | number_wildcard | expression) (',' (list_wildcard |
      number_wildcard | expression))*
    ;

methodCall
    : (identifier | THIS | SUPER) arguments // TODO: test
    ;

expression
    : expr_wildcard
    | number_wildcard
```

```
// Expression order in accordance with https://introcs.cs.princeton.edu/
    java/11precedence/
// Level 16, Primary, array and member access
| primary
| expression '[' expression ']'
| expression bop = '.' (
    identifier
    | methodCall
    | THIS
    | NEW nonWildcardTypeArguments? innerCreator
    | SUPER superSuffix
    | explicitGenericInvocation
)
// Method calls and method references are part of primary, and hence
    level 16 precedence
| methodCall
| expression '::' typeArguments? identifier
| typeType '::' (typeArguments? identifier | NEW)
| classType '::' typeArguments? NEW
| switchExpression // Java17

// Level 15 Post-increment/decrement operators
| expression postfix = ('++' | '--')

// Level 14, Unary operators
| prefix = ('+' | '-' | '++' | '--' | '~' | '!') expression

// Level 13 Cast and object creation
| '(' annotation* typeType ('&' typeType)* ')' expression
| NEW creator

// Level 12 to 1, Remaining operators
| expression bop = ('*' | '/' | '%') expression          // Level 12,
    Multiplicative operators
| expression bop = ('+' | '-') expression                // Level 11,
    Additive operators
| expression ('<' '<' | '>' '>' '>' | '>' '>') expression // Level 10,
    Shift operators
| expression bop = ('<=' | '>=' | '>' | '<') expression   // Level 9,
    Relational operators
| expression bop = INSTANCEOF (typeType | pattern)
| expression bop = ('==' | '!=') expression                    //
    Level 8, Equality Operators
| expression bop = '&' expression                             //
    Level 7, Bitwise AND
| expression bop = '^' expression                             //
    Level 6, Bitwise XOR
| expression bop = '|' expression                             //
    Level 5, Bitwise OR
| expression bop = '&&' expression                            //
    Level 4, Logic AND
```

```
    | expression bop = '||' expression                                  //
      Level 3, Logic OR
    | <assoc = right> expression bop = '?' expression ':' expression //
      Level 2, Ternary
    // Level 1, Assignment
    | <assoc = right> expression bop = (
        '='
        | '+='
        | '-='
        | '*='
        | '/='
        | '&='
        | '|='
        | '^='
        | '>>='
        | '>>>='
        | '<<='
        | '%='
    ) expression

    // Level 0, Lambda Expression
    | lambdaExpression // Java8
    ;

// Java17
pattern
    : variableModifier* typeType annotation* identifier
    ;

// Java8
lambdaExpression
    : lambdaParameters '->' lambdaBody
    ;

// Java8
lambdaParameters
    : identifier
    | '(' formalParameterList? ')'
    | '(' identifier (',' identifier)* ')'
    | '(' lambdaLVTIList? ')'
    ;

// Java8
lambdaBody
    : expression
    | block
    ;

primary
    : '(' expression ')'
    | THIS
```

```
    | SUPER
    | literal
    | identifier
    | typeTypeOrVoid '.' CLASS
    | nonWildcardTypeArguments (explicitGenericInvocationSuffix | THIS
      arguments)
    ;

// Java17
switchExpression
    : SWITCH parExpression '{' switchLabeledRule* '}'
    ;

// Java17
switchLabeledRule
    : CASE (expressionList | NULL_LITERAL | guardedPattern) (ARROW | COLON)
      switchRuleOutcome
    | DEFAULT (ARROW | COLON) switchRuleOutcome
    ;

// Java17
guardedPattern
    : '(' guardedPattern ')'
    | variableModifier* typeType annotation* identifier ('&&' expression)*
    | guardedPattern '&&' expression
    ;

// Java17
switchRuleOutcome
    : block
    | blockStatement*
    ;

classType
    : (classOrInterfaceType '.')? annotation* identifier typeArguments?
    ;

creator
    : nonWildcardTypeArguments? createdName classCreatorRest
    | createdName arrayCreatorRest
    ;

createdName
    : identifier typeArgumentsOrDiamond? ('.' identifier
      typeArgumentsOrDiamond?)*
    | primitiveType
    ;

innerCreator
    : identifier nonWildcardTypeArgumentsOrDiamond? classCreatorRest
    ;
```

```
arrayCreatorRest
    : ('[' ']')+ arrayInitializer
    | ('[' expression ']')+ ('[' ']')*
    ;

classCreatorRest
    : arguments classBody?
    ;

explicitGenericInvocation
    : nonWildcardTypeArguments explicitGenericInvocationSuffix
    ;

typeArgumentsOrDiamond
    : '<' '>'
    | typeArguments
    ;

nonWildcardTypeArgumentsOrDiamond
    : '<' '>'
    | nonWildcardTypeArguments
    ;

nonWildcardTypeArguments
    : '<' typeList '>'
    ;

typeList
    : typeType (',' typeType)*
    ;

typeType
    : simple_wildcard
    | var_wildcard
    | annotation* (classOrInterfaceType | primitiveType | simple_wildcard |
      var_wildcard) (annotation* '[' ']')*
    ;

primitiveType
    : BOOLEAN
    | CHAR
    | BYTE
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    ;

typeArguments
```

```
    : '<' typeArgument (',' typeArgument)* '>'
    ;

superSuffix
    : arguments
    | '.' typeArguments? identifier arguments?
    ;

explicitGenericInvocationSuffix
    : SUPER superSuffix
    | identifier arguments
    ;

arguments
    : '(' (expressionList? | list_wildcard | number_wildcard) ')'
    ;

// Syntax of wildcards
simple_wildcard: WILDCARD | WILDCARD_SPACE;
var_wildcard: VAR_WILDCARD;
list_wildcard: WILDCARD '*';
contains_wildcard: WILDCARD '<' (simple_wildcard | var_wildcard |
   contains_wildcard | expression) '>';
simple_compound_wildcard: WILDCARD wildcard_number? block;
multiple_compound_wildcard: WILDCARD '*' block;
number_wildcard: WILDCARD wildcard_number;
wildcard_number: '(' DECIMAL_LITERAL (',' | ',' DECIMAL_LITERAL)? ')';

// Composite wildcards
stmt_wildcard: (simple_wildcard | var_wildcard | list_wildcard |
   contains_wildcard);
expr_wildcard:  var_wildcard | contains_wildcard | simple_wildcard;
compound_wildcard: simple_compound_wildcard | multiple_compound_wildcard;
```

References

[1] Quentin Colla. "A Comparison of Program Query Languages to Detect Python Programming Misconceptions". eng. In: (2025). URL: https://thesis.dial.uclouvain.be/bitstreams/fbc4db4a-8f36-4dc2-abb5-e9f862cc23a0/download (visited on 21/05/2026).

[2] Quentin Colla, Kim Mens and Julien Liénard. "A Comparison of Three Program Query Languages to Detect Python Programming Misconceptions." eng. In: *Open Access Series in Informatics*. Vol. 134. 2025. DOI: 10.4230/OASIcs.Programming.2025.21. URL: https://hdl.handle.net/2078.5/256577 (visited on 21/05/2026).

[3] *Pyttern/README.md at main · JulienLie/Pyttern*. en. URL: https://github.com/JulienLie/Pyttern/blob/main/README.md (visited on 18/05/2026).

[4] *ANTLR*. URL: https://www.antlr.org/ (visited on 18/05/2026).

[5] *GitHub - antlr/antlr4 at master*. en. URL: https://github.com/antlr/antlr4/tree/master (visited on 18/05/2026).

[6] Nicolas Dedoyard. "Jattern: A program query language for Java". eng. In: (2025). URL: https://hdl.handle.net/2078.2/43874 (visited on 18/05/2026).

[7] *Regular expression HOWTO*. en. URL: https://docs.python.org/3/howto/regex.html (visited on 17/05/2026).

[8] Firas Dib. *regex101: build, test, and debug regex*. en. URL: https://regex101.com/ (visited on 18/05/2026).

[9] *Email Address Regular Expression That 99.99% Works*. en-US. URL: https://emailregex.com/index.html (visited on 18/05/2026).

[10] S. Paul and A. Prakash. "A framework for source code search using program patterns". In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 463–475. ISSN: 1939-3520. DOI: 10.1109/32.295894. URL: https://ieeexplore.ieee.org/document/295894 (visited on 18/05/2026).

[11] *Docs home | Semgrep*. en. May 2026. URL: https://semgrep.dev/docs/ (visited on 18/05/2026).

[12] *Documentation Index | PMD Source Code Analyzer*. URL: https://docs.pmd-code.org/latest/index.html (visited on 18/05/2026).

[13] *Checkstyle 13.4.2 – checkstyle*. URL: https://checkstyle.org/index.html (visited on 18/05/2026).

[14] *Writing Checks – checkstyle*. URL: https://checkstyle.org/writingchecks.html (visited on 22/05/2026).

[15] Michael Martin, Benjamin Livshits and Monica S. Lam. "Finding application errors and security flaws using PQL: a program query language". en. In: *ACM SIGPLAN Notices* 40.10 (Oct. 2005), pp. 365–383. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1103845.1094840. URL: https://dl.acm.org/doi/10.1145/1103845.1094840 (visited on 17/05/2026).

[16] *SpotBugs*. URL: https://spotbugs.github.io/ (visited on 18/05/2026).

[17] *Observer*. en. URL: https://refactoring.guru/design-patterns/observer (visited on 29/05/2026).

[18] *Reporting - pytest-cov 7.1.0 documentation*. URL: https://pytest-cov.readthedocs.io/en/latest/reporting.html (visited on 29/05/2026).

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