UCLouvain

epl

École polytechnique de Louvain

Reimplementing, Benchmarking and Optimising Pyttern in Java for Faster Pattern Matching

Author: **Quentin Lépine**
Supervisor: **Kim Mens**
Reader: **Julien Liénard**
Academic year 2025–2026
Master [120] in Computer Science and Engineering

Abstract

Pyttern is a source-code pattern-matching tool introduced by Liénard, Mens, and Nijssen [2]. Patterns are written in a syntax similar to the target language extended with wildcards. Users can thus describe the code fragments they look for without learning a separate query language or manipulating syntax trees directly. Given a program, Pyttern reports occurrences of such patterns. The current implementation targets Python and is primarily intended for the large-scale analysis of student submissions.

The current Python implementation is slow enough to disrupt the user's workflow, which limits its use in teaching contexts. This thesis improves Pyttern's performance on two bottlenecks: parsing and patterns whose search space grows combinatorially. Pyttern is first reimplemented in Java, with the original architecture kept unchanged. The matching algorithm is then modified in three ways: a fix removes a transition that produced duplicate matches, a pruning step eliminates redundant wildcards before automaton construction, and a new mechanism shares paths that converge on the same configuration to avoid duplicate work.

The Java port alone reduces parsing time by a factor of 2.5 on individual files and by close to 20 on large batches; an interpreted-only configuration shows that this gain comes mostly from HotSpot's just-in-time compilation rather than from the language change itself. The three matching optimisations further cut the cost of combinatorial patterns by several orders of magnitude on the most extreme cases, bringing them in line with the rest of the benchmark.

Acknowledgements

I would like to thank Professor Kim Mens and Julien Liénard for their continuous support and guidance throughout this work, their availability for our meetings, and their feedback on my work.

Use of AI tools. AI-based tools were used at several points during this thesis. ChatGPT and Claude were used to generate editable XML files for the styling of some of the diagrams and figures, which are individually flagged with a footnote. Claude was used as a linguistic assistant throughout the writing of the thesis, for grammar checking, translation, reformulation suggestions, and consistency review. All outputs were reviewed and validated by the author.

Contents

1 Introduction **6**

2 Background Material **9**
2.1 Pyttern . . . 9
2.1.1 Patterns and Wildcards . . . 9
2.1.2 How Pyttern Works . . . 11
2.2 Hyperfine . . . 13
2.2.1 Overview . . . 13
2.2.2 Rationale for Selection . . . 13
2.2.3 Features Employed . . . 14
2.2.4 Cold-Start Overhead . . . 14
2.3 Pyperf . . . 14
2.3.1 Overview . . . 15
2.3.2 Rationale for Selection . . . 15
2.3.3 Features Employed . . . 15
2.3.4 Worker Process Isolation . . . 16
2.3.5 Stage Isolation . . . 16
2.4 JMH . . . 16
2.4.1 Overview . . . 16
2.4.2 Rationale for Selection . . . 16
2.4.3 Features Employed . . . 16
2.4.4 JIT Warm-Up . . . 17
2.4.5 Stage Isolation . . . 17

3 Problem Statement **18**

4 Port Pyttern to Java **20**
4.1 Benchmark . . . 20
4.1.1 Benchmark Setup . . . 20
4.1.2 Experimental Environment . . . 22
4.1.3 Results . . . 22
4.1.4 Threats to Validity . . . 23
4.2 Solution . . . 23

4.2.1 ANTLR Preservation . . . 24
4.3 Conclusion . . . 24

5 Parsing in Java: Validation **25**
5.1 Benchmark . . . 25
5.1.1 Methodology . . . 25
5.1.2 Inputs . . . 26
5.1.3 Experimental Environment . . . 26
5.2 Results . . . 26
5.3 Analysis and Discussion . . . 31
5.4 Threats to Validity . . . 32
5.5 Conclusion . . . 32

6 Matching in Java: Validation **34**
6.1 Benchmark . . . 34
6.1.1 Methodology . . . 34
6.1.2 Inputs . . . 35
6.1.3 Experimental Environment . . . 35
6.2 Results . . . 35
6.3 Analysis and Discussion . . . 37
6.4 Threats to Validity . . . 38
6.5 Conclusion . . . 38

7 Optimisation **39**
7.1 Benchmark . . . 39
7.1.1 Methodology . . . 39
7.1.2 Inputs . . . 40
7.1.3 Experimental Environment . . . 43
7.2 PDA Optimisation . . . 43
7.2.1 PDA Representation of `?:*` . . . 44
7.2.2 Problem . . . 44
7.2.3 Fix . . . 45
7.2.4 Performance . . . 46
7.3 Pruning . . . 47
7.3.1 Integration . . . 47
7.3.2 Rules . . . 47
7.3.3 Performance . . . 48
7.4 Reducing redundant exploration during matching . . . 50
7.4.1 Naive Memoisation . . . 50
7.4.2 Dead-end Detection . . . 50
7.4.3 Counting Convergent Paths . . . 51
7.4.4 Performance . . . 52
7.5 Optimisation Interference . . . 53

7.6 Performance 55
7.7 Conclusion 57

8 Conclusion **58**

9 Lessons learned **60**

10 Future Work **61**
10.1 JVM Startup Optimisation 61
10.2 Replacing the Parser 61

Chapter 1

Introduction

Context

Searching source code by structure rather than by text is a recurring need in software engineering and in education. It enables queries that a textual search cannot express, such as locating every recursive call in a program, every function whose body has a specific control-flow shape, or every hardcoded solution to an exercise. Pyttern is a tool designed for this task: the user writes a pattern that extends the target language with wildcards for the parts left unspecified, and Pyttern reports every match in a given program. The current implementation targets Python, with a primary application in the large-scale analysis of student submissions.

Problem

In its current Python implementation, Pyttern is slow enough to disrupt the user's workflow. A profiling study of six representative patterns shows that parsing the input program dominates the runtime in five cases, accounting for over 97% of the total. On the sixth pattern, matching itself becomes the bottleneck, because the wildcards involved force the matcher to explore a combinatorially growing number of decompositions. Both bottlenecks can be addressed without altering Pyttern's matching semantics.

Motivation

The slowness is the main practical obstacle to using Pyttern. The intended usage, applying a pattern to a whole cohort of submissions at once, is precisely the regime most affected. Bringing the runtime down to a few seconds on realistic batches is necessary for routine use in this context.

Objectives

The goal of this thesis is to make Pyttern fast enough that it no longer disrupts the user workflow, without altering its matching semantics. Two objectives follow from the bottleneck analysis: reducing the parsing cost and reducing the matching cost in the cases where it explodes. No strict numerical target is fixed; the qualitative goal is that typical workloads complete within a few seconds.

Approach

The work proceeds in two parts. The first part reimplements Pyttern in Java, preserving the ANTLR grammar, the pushdown automaton construction, and the matching semantics. The second part targets the matching cost through three algorithmic changes: fixing a transition in the automaton that produced duplicate matches in the encoding of `?:*`, pruning redundant wildcards before the automaton is built, and introducing a matcher that merges paths converging on the same configuration to avoid duplicate work.

Contributions

- A profiling study of the existing Python implementation, identifying parsing as the dominant cost in the common case and the matcher as the bottleneck on patterns with specific wildcards.
- A complete reimplementation of Pyttern in Java, preserving the original ANTLR grammar, pushdown automaton construction, and matching semantics.
- An empirical comparison of the Java and Python implementations on file-size and batch-size workloads, including an interpreted-only configuration that isolates the contribution of the just-in-time (JIT) compiler of Java.
- An empirical validation of the Java reimplementation against the Python original on the full implementation, confirming that the gain observed on parsing is preserved when matching is included.
- A correction in the pushdown automaton encoding of the `?:*` wildcard, removing an exponential duplication of matches caused by an under-constrained transition.
- Three additional pruning rules in the existing tree pruner, collapsing consecutive `?:*` and `?*` wildcards before the automaton is built.
- A matcher variant that shares equivalent configuration explorations while preserving the exact match count.
- A combined evaluation of the three optimisations on dedicated benchmark suites, including an analysis of their interference.

Roadmap

The remainder of the thesis is organised as follows. Chapter 2 introduces the material needed to follow the rest of the thesis: the Pyttern language, its matching pipeline based on a pushdown automaton, and the three benchmarking frameworks used in this work. Chapter 3 states the research problem and positions Pyttern with respect to alternative pattern-matching tools. Chapter 4 reports the profiling experiment that identifies parsing as the dominant cost, then motivates the choice of a full Java port over a hybrid solution. Chapter 5 validates the first stage of the port by comparing the Java and Python parsers across file sizes and batch sizes. Chapter 6 validates the second stage by evaluating the full Java pipeline against the original Python implementation. Chapter 7 develops and evaluates the three matching optimisations and analyses their interference. Chapter 8 synthesises the contributions and revisits the objectives. Chapter 9 collects practical lessons learned during the port. Chapter 10 outlines two directions for further improvement: reducing the JVM startup overhead and replacing the parser by a faster alternative.

Chapter 2

Background Material

2.1 Pyttern

Pyttern is a dedicated pattern language introduced by Liénard, Mens, and Nijssen [2] for querying Python source code. It was designed to make structural source-code queries easier to express, since existing tools often require users to master complex query languages or to reason directly over syntax trees. To address this, Pyttern keeps its syntax close to regular Python and adds a small set of regex-inspired wildcards. The current Python implementation is publicly available on GitHub[1] and is still under active development: a Java grammar is currently being added to extend support beyond Python, and the pattern language is being extended with mechanisms such as sub-pattern calls and logical operators [1].

2.1.1 Patterns and Wildcards

The Pyttern pattern language can be viewed as an extension of the target programming language (Python) enriched with various wildcards. Pyttern provides seven kinds of wildcards, each corresponding to a distinct matching behaviour, such as matching a single syntactic element, a sequence of elements, or an indented body. For instance, the wildcard `?name` matches any element on its first occurrence. Subsequent occurrences of the same named wildcard must match that same element, as illustrated in Table 2.1 where the pattern `?x` will be matched twice against the same literal `a`.
Table 2.2 presents a more advanced example. The pattern matches code containing a hardcoded list which can be used for example to detect students who store prime numbers in a list instead of computing them dynamically. The function header on line 1 uses `?` to match any function name and `?*` to match any parameter list, while the `?:*` on line 2 allows the assignment below to appear at any nesting level within the function body. The assignment itself, on line 3, matches some variable (`?`) assigned to a list literal whose length is constrained to at least three elements by `?{3,}`. The pattern therefore captures any

[1] `https://github.com/JulienLie/Pyttern`

function that, somewhere in its body, assigns a list of three or more elements to a variable, regardless of the function signature.
A brief description of all these wildcards is provided in Table 2.3.

Code	Pattern

`def foo():`
`  a = 0`
`  return a`
	`def ?():`
	`  ?x = 0`
	`  return ?x`

Table 2.1: Wildcard matching example.

Code	Pattern

`def factors(n):`
`  l = [2,3,5,7,11,13]`
`  t = []`
`  for i in range(len(l)):`
`  if n%l[i] == 0:`
`  t.append(l[i])`
`  i -= 1`
`  nbr = len(t)`
`  return (nbr)`
	`def ?(?*):`
	`  ?:*`
	`  ? = [?{3,}]`

Table 2.2: Hardcoded list matching example.

Wildcard	Description
`?`	Matches one element[1]
`?{n, m}`	Matches between `n` and `m` elements
`?*`	Matches zero or more consecutive elements at the same level of indentation
`?name`	Matches one element and binds it to `name`
`?:`	Matches one element with an associated body
`?:*`	Matches zero or more consecutive indented blocks
`?<...>`	Matches a node that contains the pattern written inside the wildcard

[1] In Pyttern, an element corresponds to a syntactic node in the parse tree, such as a variable, an expression, or a statement.

Table 2.3: Wildcards supported by Pyttern [1].

2.1.2 How Pyttern Works

A Pyttern expression is matched against Python code in three main steps [2].

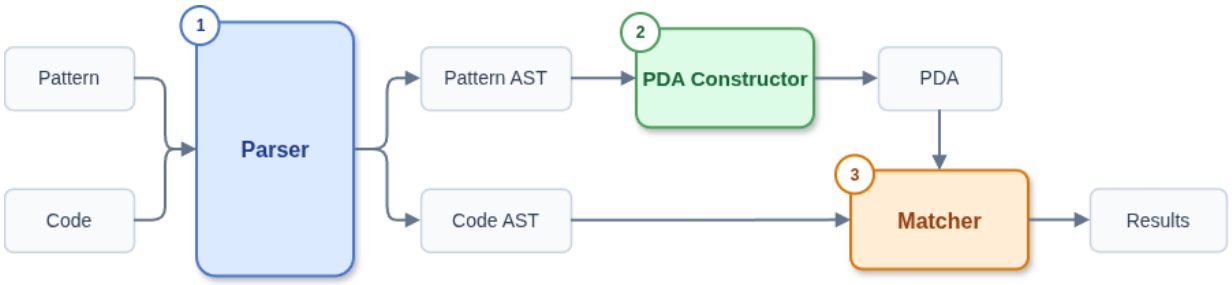


Figure 2.1: High-level workflow of Pyttern[2].

1. Parsing First, both the source code and the patterns are parsed to obtain a tree representation. Pyttern relies on ANTLR, a parser generator, to build a parse tree from a given grammar. The grammar depends on the targeted language: in practice, one grammar is used for the source language (e.g., Python) and one extended grammar is used for the pattern language (i.e., the same language extended with wildcard constructs). The raw parse trees obtained are then refined by a dedicated visitor that discards non-semantic tokens and flattens grammatical structures that are more detailed than the matching process requires. The resulting tree, in either case, is composed of typed nodes defined by the grammar (Block node for indented body, Stmt node for statement in Python, etc.); these types are used later to condition the transitions of the PDA. Although the current implementation of Pyttern only supports Python, in principle, any language for which a grammar is available could be supported, provided that the grammar and the resulting tree representation are adapted to integrate wildcards and remain compatible with Pyttern.

2. Pushdown automaton construction Second, based on the pattern's tree, Pyttern builds a pushdown automaton (PDA), which serves as an executable representation of the pattern used by the matcher to navigate and match the source code tree. A PDA is a finite-state machine augmented with a stack, which allows it to handle the nested, hierarchical structure of a parse tree. The PDA builder translates each node of the pattern tree into PDA transitions, with each type of wildcard producing its own specific transition pattern; a list wildcard, for example, generates a self-loop to accept arbitrarily many siblings. In Pyttern, the stack uses two symbols: `I` is pushed each time the matcher descends into a child node and popped when it returns to the parent, while `B` is pushed on entry to a wildcard body (`?:*` or `?<...>`) and popped on exit, marking the boundary of the region matched by the wildcard. The stack therefore reflects both the chain of descents between the

[2]XML layout produced with the assistance of Claude/ChatGPT.

root of the source tree and the current position and the boundaries of regions matched by wildcard bodies, allowing the automaton to take decisions that depend on the surrounding context. Once the full pattern tree has been visited, the resulting PDA provides a structured way to match the code tree against the pattern.

Each transition from a state q_i to a state q_j in the PDA carries five pieces of information, summarised in Figure 2.2: a condition on the top of the stack (a symbol to read, for example a `B` symbol at the top of the stack), a condition on the current node of the source code (its type, or its text in the case of a terminal; in the figure, the node must be of type `If_stmt`), an instruction to navigate in the source tree (such as moving to the left child, the right sibling, or the parent; `LEFT_CHILD` in the figure), a destination state (q_j in the figure), and a new symbol to push onto the stack (`BI` in the figure, meaning `B` and `I` are pushed onto the stack). A transition can only be taken when both the stack and the current node match its conditions. This combination of stack and tree conditions is what makes the PDA capable of expressing the structural constraints encoded in a pattern. Figure 2.3 represents the format of the PDA transitions used later in this thesis.

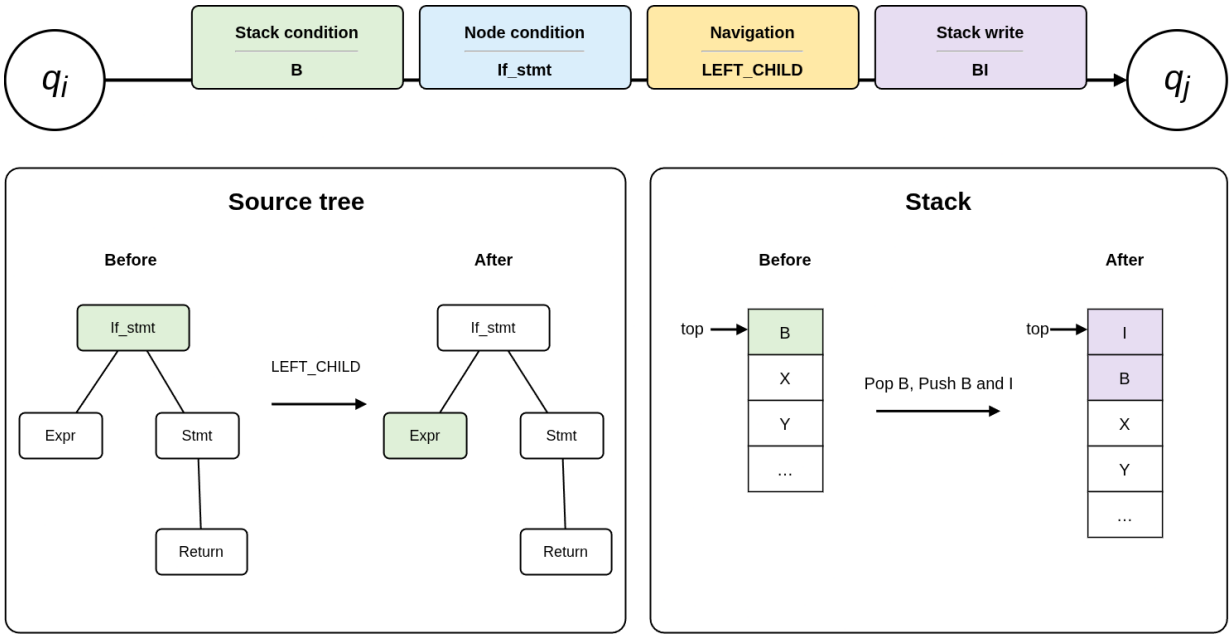


Figure 2.2: Anatomy of a PDA transition in Pyttern. The labelled fields describe the conditions that must be satisfied and the actions performed when the transition is taken[4].

[4]XML layout produced with the assistance of Claude/ChatGPT.

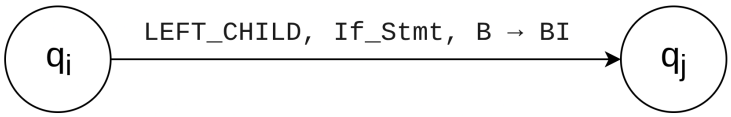


Figure 2.3: Format used later in this thesis to represent PDA transitions.

3. Matching During the matching process, Pyttern traverses the source code tree guided by the transitions of the pushdown automaton. Because wildcards allow several transitions to apply from the same point, the exploration is not limited to a single path, and the matcher therefore maintains a list of configurations, each representing a snapshot of one ongoing exploration of the pattern against the source code: a state, a position in the source tree, a stack, and a set of variable bindings. At each step, the matcher takes one configuration from the list, computes every transition that is applicable from it, and adds the resulting new configurations back to the list; the process continues until no configuration remains. Several transitions can be applicable from the same configuration, in which case the exploration splits into as many branches, all carried out in parallel. Whenever a configuration reaches the acceptance state, it is recorded as a match. The output is a match set that contains all occurrences found. Because each configuration can produce several others at every step, the number of configurations to process can grow rapidly with the structure of the pattern. This growth is one of the bottlenecks tackled by the optimisations presented in Chapter 7.

2.2 Hyperfine

A significant part of the empirical work in this thesis consists of comparing the execution time of the Python and Java implementations of Pyttern across workloads of varying file and batch sizes. Hyperfine is the tool used to collect these measurements whenever the two implementations are run as complete programs.

2.2.1 Overview

Hyperfine is an open-source command-line benchmarking tool written in Rust [7]. It measures the total elapsed execution time of shell commands externally, that is, independently of the language or runtime being evaluated. This black-box approach requires no modification to the source code under test and imposes no instrumentation overhead on the measured process.

2.2.2 Rationale for Selection

Comparing the performance of Python and Java introduces a fundamental measurement challenge: no common internal timer exists across both runtimes; dedicated microbenchmarking frameworks such as JMH and pyperf operate within different runtime contexts

and cannot guarantee that they measure the same quantity. Delegating measurement to the shell provides a single, neutral observation point, namely the total elapsed time of the process as seen by the operating system, ensuring a fair and more consistent comparison between the two implementations.

2.2.3 Features Employed

The following Hyperfine options were used in this study [7]:

- `-warmup`: a configurable number of warm-up runs is performed prior to measurement, allowing file-system and CPU caches to warm-up and reducing variability between runs. In this study, 5 warm-up runs were performed.
- `-runs`: explicitly controls the number of timed iterations retained for the final statistics. Hyperfine defaults to at least 10 runs, a value retained in this study. Hyperfine automatically computes the mean, standard deviation (σ), median, minimum, and maximum of the observed execution times.
- `-export-json`: exports the complete benchmark results in JSON format, including the summary statistics, the individual timing of each run, and the exit codes, which were subsequently used for statistical analysis and visualisation.

A typical invocation takes the following form:

```
hyperfine -warmup 5 -runs 10 -export-json output.json "command"
```

2.2.4 Cold-Start Overhead

Each iteration launched by Hyperfine spawns a new operating system process [7]. For the Java Virtual Machine, this entails a full cold-start at every run: class loading and static initialisation. Hyperfine accounts for the overhead introduced by the shell's own process-spawning mechanism through an internal calibration step, subtracting this fixed cost from the reported times; they therefore capture the end-to-end cost of a standalone invocation, including JVM startup effects.

2.3 Pyperf

Beyond the end-to-end comparison between the two implementations, a separate part of the empirical work examines where execution time is actually spent inside the Python implementation of Pyttern, stage by stage. Pyperf is the tool used to collect these per-stage measurements.

2.3.1 Overview

Pyperf [9] is an open-source Python microbenchmarking framework that measures the execution time of isolated Python callables directly within the runtime. Unlike shell-level tools such as Hyperfine, it eliminates process-spawning overhead from the timed region and provides precise control over what is and is not included in the measurement. Pyperf handles warm-up iterations, repeated measurements, and statistical aggregation automatically, reporting the average execution time, its standard deviation, and the observed range.

2.3.2 Rationale for Selection

While Hyperfine provides a language-agnostic, end-to-end view of execution time, it cannot isolate individual stages of the Pyttern pipeline: parsing, PDA construction, and matching. Pyperf addresses this need by allowing each stage to be benchmarked as an independent Python callable, while controlling which setup operations are included in the measured region. Pyperf is also used by *pyperformance* [10], a benchmark suite referenced in the official Python documentation for evaluating CPython performance [11]. It was therefore selected for benchmarking the Python implementation.

2.3.3 Features Employed

The following pyperf features were used in this study, with default values:

- `bench_func()`: benchmarks an isolated Python callable. The function passed to `bench_func()` defines the timed region precisely; any setup executed outside this function is excluded from the measurement.
- `-warmup`: pyperf performs 3 warm-up iterations by default prior to the timed runs, allowing the runtime to reach a stable state before results are recorded.
- `-runs`: pyperf defaults to 5 timed iterations per worker process, a value retained in this study.
- `-inherit-environ`: forwards specified environment variables to the worker processes spawned by pyperf, which is required to pass input paths to the benchmark.
- `-o` (`-output`): exports the complete benchmark results in JSON format for subsequent statistical analysis and visualisation.

A typical invocation in this study takes the following form:

```
python3 microbenchmark.py -bench all -o output.json -inherit-environ
PYTTERN_PATTERN_PATH,CODE_PATH
```

2.3.4 Worker Process Isolation

Pyperf executes each benchmark in a dedicated worker process running in a fresh environment, which enhances reproducibility across benchmarks [9]. Unlike Hyperfine, the worker process is not restarted between individual runs, so the Python runtime remains warm across the timed iterations within a single benchmark.

2.3.5 Stage Isolation

Each stage of the Pyttern pipeline (parsing, PDA construction, pattern matching, and end-to-end execution) is benchmarked independently. For PDA construction and pattern matching, the prerequisite stages are executed in advance and their results passed into the benchmarked function via a closure, so that only the cost of the stage under measurement is captured in the timed region. Internal caches are explicitly cleared before each run to ensure that measured times reflect the true per-stage cost.

2.4 JMH

The same per-stage analysis is also conducted on the Java implementation of Pyttern, both to identify its dominant costs and to put them in perspective with those of the Python implementation. JMH is the tool used to collect these JVM-side measurements.

2.4.1 Overview

JMH (Java Microbenchmark Harness) is the official microbenchmarking framework for the JVM, developed by OpenJDK [5]. Like pyperf, it operates internally within the runtime, measuring the execution time of isolated methods directly. JMH is designed to address JVM-specific measurement pitfalls, most notably the elimination of benchmarked code by the JIT compiler when computed values are not consumed. It handles warm-up iterations, repeated measurements, and statistical aggregation automatically, reporting summary statistics over the timed iterations, as pyperf does.

2.4.2 Rationale for Selection

JMH addresses the same need on the JVM side: it allows each stage to be benchmarked as an isolated Java method, while controlling which setup operations are included in the measured region. As the benchmarking framework developed and maintained by OpenJDK [5], it is the standard tool for microbenchmarking on the JVM. It was therefore selected for benchmarking the Java implementation.

2.4.3 Features Employed

JMH benchmarks are configured via annotations. The following were used in this study [5]:

- `@BenchmarkMode(Mode.AverageTime)`: reports the average execution time per operation.
- `@OutputTimeUnit(TimeUnit.MILLISECONDS)`: sets the time unit of reported results to milliseconds.
- `@Warmup(iterations = 5, time = 1)`: performs five warm-up iterations prior to measurement, allowing the JIT compiler to identify and compile frequently executed code paths before results are recorded.
- `@Measurement(iterations = 10, time = 1)`: specifies ten timed iterations retained for the final statistics.
- `@Fork(1)`: executes the benchmark in a fresh, isolated JVM process, preventing interference between benchmark classes.

Computed values are passed to a `Blackhole` instance, preventing the JIT compiler from eliminating benchmarked code as dead code.

2.4.4 JIT Warm-Up

Unlike Hyperfine, which spawns a new process for each iteration and therefore captures cold-start behaviour, JMH's warm-up phase allows the HotSpot JIT compiler to reach a stable compilation state before results are recorded. The reported times thus reflect stabilised throughput rather than first-invocation performance. Hyperfine and JMH therefore measure two different regimes: cold-start cost on the one hand, stabilised throughput on the other.

2.4.5 Stage Isolation

Each stage of the Java pipeline (parsing, PDA construction, pattern matching, and end-to-end execution) is benchmarked independently. Parsing and PDA construction are performed once per trial in a setup method executed outside the timed region, so that downstream benchmarks measure only the cost of their respective stage. Similarly, after each invocation internal caches are cleared, reducing the risk of residual state distorting individual measurements.

Chapter 3

Problem Statement

The background chapter presented Pyttern and its three-stage pipeline. The language itself is well suited to its intended use, but the current implementation is too slow for the workloads it targets. The slowdown stems from two bottlenecks: parsing the input program, and matching certain patterns whose structure forces the matcher to explore a combinatorially growing number of decompositions.

Relevance

Pyttern's intended setting is the large-scale analysis of student submissions: a single pattern is applied to a whole cohort, typically dozens to hundreds of files at a time. This is also the situation where the current performance hurts the most, and the slowness directly undermines this use case.

Positioning

Other pattern-matching tools exist for source code, but Pyttern is positioned around a combination of features that fits its intended use case: patterns are written as extensions of Python code, making them accessible to users without expertise in program-query languages, while dedicated wildcards allow these patterns to abstract over structural elements of parse trees [2]. Users can therefore describe recurring structures in Python programs without having to learn a separate formalism or work directly with internal program representations. Replacing Pyttern with another tool would lose this balance between approachability and structure-aware matching. The contribution of this thesis is not a new pattern-matching approach but a more efficient implementation of an existing one: the goal is to make Pyttern competitive in practice on the workloads it was designed for, without altering its language or its matching semantics.

Research question

The work is organised around the following question: can a reimplementation of Pyttern, combined with targeted optimisations of the matcher, bring its execution time within a few seconds on realistic batches of student submissions, including on patterns where matching itself becomes combinatorial? The next chapter reports the profiling experiment that quantifies the parsing bottleneck, then motivates the choice of Java for this reimplementation. The following chapters validate the port empirically, and Chapter 7 addresses the matching bottleneck through three targeted optimisations.

Chapter 4

Port Pyttern to Java

The current Python implementation of Pyttern is slow enough to disrupt its use, particularly in teaching contexts. This chapter first benchmarks it to determine where execution time is spent, then motivates a reimplementation in Java on that basis.

4.1 Benchmark

The purpose of this benchmark is to identify the main performance bottleneck of the current Python implementation of Pyttern in order to prioritise optimisation work. At this stage, the objective is to quantify which stage dominates execution time under a realistic usage model.

4.1.1 Benchmark Setup

We benchmark the execution of one student submission against one pattern and record the execution time of each stage as well as the overall end-to-end runtime. This experiment is repeated for 50 student solutions and 6 patterns, resulting in 300 (submission, pattern) pairs. For each pair, we use `pyperf` with its default configuration as described in Section 2.3 (3 warm-up iterations, 5 timed iterations per worker process). The values reported in the figures below are obtained by averaging, for each pattern and each stage, the per-pair pyperf means across the 50 submissions. Parsing is intentionally performed from scratch each time: the parsing stage is fully re-executed for every run and internal caches that could reuse parsing artefacts are cleared before execution, to ensure we measure the true cost of parsing rather than cached results. In this benchmark, the parsing stage refers exclusively to the ANTLR lexer and parser invocation; the refinement visitor described in Section 2.1 is excluded from the timed region, since the goal is to measure the cost of the ANTLR runtime itself, which is the component being replaced.

The input corpus consists of correct student solutions to the same programming exercise from a real Python course; the code shown in Listing 4.2 is an example of a student

submission used in the benchmark. The benchmark includes only code that compiles successfully, since the Python version of ANTLR crashes on syntactically invalid programs while the Java runtime does not, due to differences in ANTLR's error handling between the two implementations.

The six patterns used in this benchmark are summarised in Table 4.1. They are taken from the existing Pyttern test suite. The pattern shown in Listing 4.1 is one of them.

Pattern	**Detects**
`observer`	An observer class, where listeners are registered and notified
`recursion`	A function that calls itself
`indentation`	A function whose body contains three nested indented blocks
`hardcodedlist`	A function that assigns a hardcoded list of at least three elements somewhere in its body
`pattern13`	A function containing a `for` loop over a range with an `if ... == 0` test inside
`piPattern`	A function approximating π by initialising, accumulating in a loop, and returning

Table 4.1: Patterns used in the benchmark.

Each (submission, pattern) pair is benchmarked independently via a Bash script invoking Pyttern for each implementation, and the full outputs are exported in JSON format.

```
def facteurs(?*):
    ?:*
        for ? in range(?*):
            if ? == 0:
                ?*
```

Listing 4.1: Example of a Pyttern pattern

```
def multiplications(n):
    z=0
    list=[]
    for i in range(1,n):
        if n%i == 0 and i not in list:
            z+=1
            list.append(i)
            list.append(n/i)
    return z
```

Listing 4.2: Example of a student submission

4.1.2 Experimental Environment

All experiments are executed on a dedicated university virtual machine running Ubuntu 24.04.4 LTS (Linux 6.8.0-88-generic, x86_64), configured with 4 vCPUs and 4 GB of RAM. The Python implementation is executed with Python 3.12.3.

4.1.3 Results

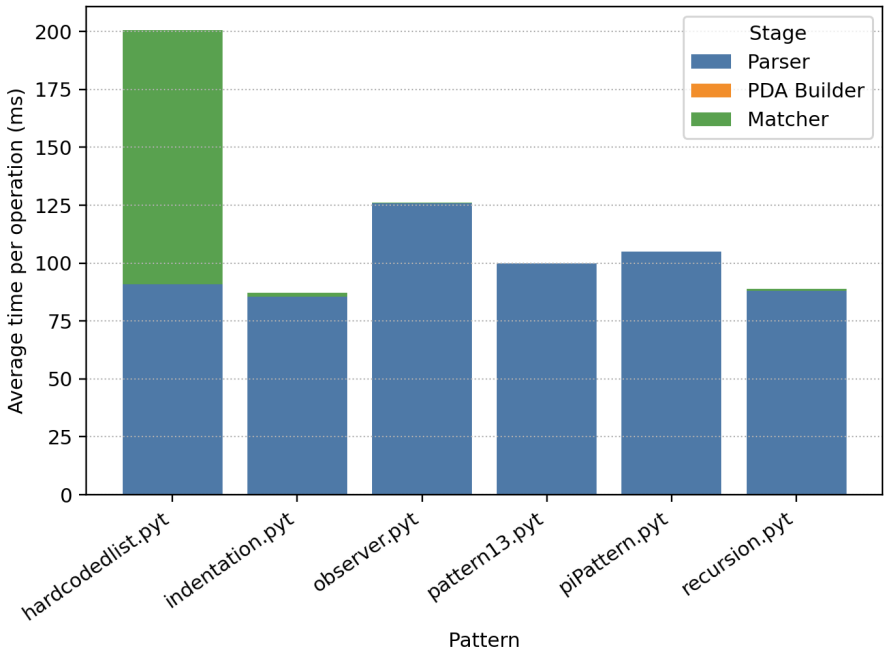


Figure 4.1: Average execution time per stage as a function of pattern. Each bar is the mean across 50 submissions of the per-pair pyperf mean.

The benchmark results in Figure 4.1 reveal a clear bottleneck: for five out of the six patterns, parsing dominates the end-to-end runtime, accounting for between **97.81%** (`indentation`) and **99.95%** (`observer`) of total execution time. In these cases, matching accounts for only **0.046%–2.19%**, while PDA construction (`PDA Builder`) remains negligible (10^{-4}%). Even in the sixth case (`hardcodedlist`) parsing still accounts for a significant part of the runtime. The end-to-end runtime ranges from approximately 88 ms (`indentation`, `recursion`) to 203 ms (`hardcodedlist`) on average.

The only case where parsing does not take the majority of the runtime is `hardcodedlist`, where parsing accounts for **44.07%** of the runtime and matching for **55.93%**, showing that for some patterns the main bottleneck may shift depending on the structure of the pattern. This can be explained by the use of wildcards containing `*`, which match zero or more elements (Table 2.3): the PDA must explore a new branch for each possible decomposition, causing the search tree to grow significantly. For instance, a list such as `[1,2,...,10]`

matched against a pattern like `[?*,?,?*]` can be satisfied in many different ways, making matching the dominant cost for this pattern. These results show that parsing is overall the most time-consuming part of Pyttern.

4.1.4 Threats to Validity

Two factors may limit the generalisability of these results. First, the corpus consists of correct solutions to a single exercise from one course, which limits diversity in both code structure and size; a more heterogeneous corpus could shift the relative weight of parsing and matching. Second, only six patterns are evaluated; while they cover distinct subsets of Pyttern's wildcards, exhaustively covering every possible pattern configuration would not be practical.

4.2 Solution

The benchmark shows that parsing dominates the end-to-end runtime. With the goal of bringing execution time down to at most a few seconds, we reimplemented Pyttern in Java. We hypothesise that it will provide a major speed-up, for three reasons:

1. The Java version of ANTLR is more mature than its Python counterpart.
2. For this kind of workload, the JVM's just-in-time compilation and adaptive runtime can yield better execution performance than the CPython interpreter.
3. Exploratory experiments conducted by Julien Liénard prior to this work suggested substantial gains when parsing is performed in Java, motivating the proposal of this master's thesis. Additional experiments conducted at the start of this work confirmed this hypothesis.

The reimplementation proceeds in two stages:

Stage 1: Java implementation of parsing. We first reimplement the parsing stage using the Java ANTLR runtime, keeping the same logic as the original implementation. This isolates the effect of the runtime change and confirms that it yields a substantial reduction in execution time under the same benchmark conditions.

Stage 2: Java implementation of the full program. As the first stage confirms the anticipated speed-up, the PDA construction and matching stages are then reimplemented in Java as well, while continuing to operate over ANTLR parse trees and preserving the original matching semantics.

For the comparison to remain valid, the Java implementation uses the same grammar, yields an equivalent parse-tree representation, and maintains the conceptual sequence of transformations (parse tree $\rightarrow$ PDA $\rightarrow$ matching result).

4.2.1 ANTLR Preservation

Pyttern is tightly coupled to ANTLR at several levels: the matcher operates on the ANTLR parse tree representation, the PDA used by the matching algorithm is built from this structure, and the pattern language itself is implemented by adapting ANTLR grammars. Replacing ANTLR would therefore not be a local change: it would require redesigning the pattern representation, the PDA construction and the matching logic, as well as adapting language-specific grammars and tooling.

A hybrid Python–Java design, on the other hand, would necessarily introduce additional overhead (cross-language invocation, data exchange, serialisation) and would complicate deployment and maintenance. It was set aside, and would only have been reconsidered if the single-language Java implementation had proved unviable for one of the stages, which was not the case.

4.3 Conclusion

A benchmark on realistic inputs showed that parsing dominates the end-to-end runtime for the vast majority of patterns, making it the primary target for optimisation. One pattern (`hardcodedlist`) revealed that the matcher can also become the dominant cost under specific structural conditions. This bottleneck motivated a full reimplementation in Java, preserving the original pipeline and matching semantics, carried out in two stages: first the parsing stage, then the full program. The next chapter validates the first stage by comparing the parsing performance of the Java and Python implementations.

Chapter 5

Parsing in Java: Validation

The previous chapter identified parsing as the dominant cost of Pyttern and motivated a reimplementation in Java, starting with the parsing stage. This chapter validates that first stage by comparing the Java parser against the existing Python parser, and quantifies the speed-up across inputs of varying size and volume.

5.1 Benchmark

The purpose of this benchmark is to compare the performance of the Java parser against the existing Python parser. We measure how fast each parser processes its inputs, independently of the rest of the application.

5.1.1 Methodology

We compare the performance of two implementations of the same program, one written in Python and one written in Java, both implementing only the parsing part of Pyttern. To ensure that both implementations are evaluated under strictly comparable conditions, we avoid relying on internal timing measurements, which may not be equivalent across languages. Instead, we measure end-to-end execution time using `hyperfine`, which repeatedly invokes a command from the shell and reports execution-time statistics. This approach, external to the program, is independent of the implementation language.

Each `hyperfine` iteration launches a new process. In particular, the Java implementation is executed as a fresh JVM instance for every run. Consequently, the measurements correspond to a cold-start scenario: the JVM is restarted each time and the benefits of just-in-time (JIT) compilation do not carry over across iterations, including warmup runs. This choice reflects a realistic usage pattern in which the tool is invoked as a standalone command.

Two complementary protocols are used in addition to the main parsing benchmark. They aim to determine which of the first two hypotheses advanced in Section 4.2 explains the speed-up: the greater maturity of the Java ANTLR runtime, or the JVM's just-in-time

compilation. The first measures runtime startup overhead by invoking each implementation on minimal programs: a `light` variant consisting of an empty `main`, and a `heavy` variant that exercises class loading without performing any parsing work. The second runs the Java parser with JIT compilation explicitly disabled (`-Xint`), so that the JVM behaves as a pure interpreter. Disabling the JIT serves two purposes: it separates what the JIT contributes from what the rest of the runtime contributes, and it provides an interpreter-vs-interpreter comparison between Java and Python runtimes.

5.1.2 Inputs

Two distinct benchmarks are considered, each with different types of inputs.

File-size benchmark. The inputs are Python source-code files extracted from The Stack dataset hosted on Hugging Face; only files that can be successfully parsed are retained. This benchmark is used to study the effect of input size on parsing performance. Input size is defined by the number of lines of code (LOC), and for each nominal size, files are selected such that their line count falls within a tolerance margin of ±5% around the target value.

Batch-size benchmark. Each input is a directory containing student submissions for the same exercise. The number of submissions in the directory is varied in order to measure how performance evolves with batch size. This benchmark is used to study the effect of the number of submissions processed in a single run.

For each input size, we benchmark both implementations on a corpus of 50 files. Each measurement uses the `hyperfine` configuration described in Section 2.2, and the complete results are exported in JSON format for subsequent analysis and visualisation.

5.1.3 Experimental Environment

All experiments are executed on a dedicated university virtual machine running Ubuntu 24.04.4 LTS (Linux 6.8.0-88-generic, x86_64), configured with 4 vCPUs and 4 GB of RAM. The Python implementation is executed with Python 3.12.3, and the Java implementation is executed with OpenJDK 25.0.2.

5.2 Results

Startup overhead The startup measurements isolate the cost of bringing each runtime up to a state where parsing can begin. Table 5.1 reports the mean execution time over 10 runs for the `light` and `heavy` variants of both implementations.

Variant	Python (ms)	Java (ms)
`light` (empty main)	10.5	47.2
`heavy` (class loading, no parsing)	156.0	112.6

Table 5.1: Runtime startup overhead, mean over 10 runs.

The `light` variant captures the cost of starting each runtime on the simplest possible program. Java pays an additional ∼37 ms relative to Python, attributable to JVM initialisation. The `heavy` variant adds the cost of loading classes representative of an actual application; Java becomes faster than Python in this configuration, which suggests that Python's import mechanism is a significant contributor to its own start-up cost. These two values give the per-invocation overhead that any single-file measurement must amortise.

File size For a parsing run on a single file, the performance gap increases as the file size grows (Figures 5.1 and 5.2). Figures 5.3 and 5.4 report the ratio between the two runtimes: for very small inputs the difference is limited, with Java faster than Python by a factor of only about 1.35. The gap widens quickly at first, then increases more slowly and tends to stabilise as the input grows. For more typical program sizes, around 100 lines of code, Java is faster by a factor of roughly 2.5.

Figures 5.1 and 5.2 also include a third curve, the Java `-Xint` configuration (without JIT). It follows the Python curve closely across the entire size range, remaining within 10% of the Python execution time and consistently slightly slower than Python rather than faster. This indicates that, when the JVM is restricted to interpreted execution, the Java parser does not exhibit any structural advantage over the Python parser on this workload.

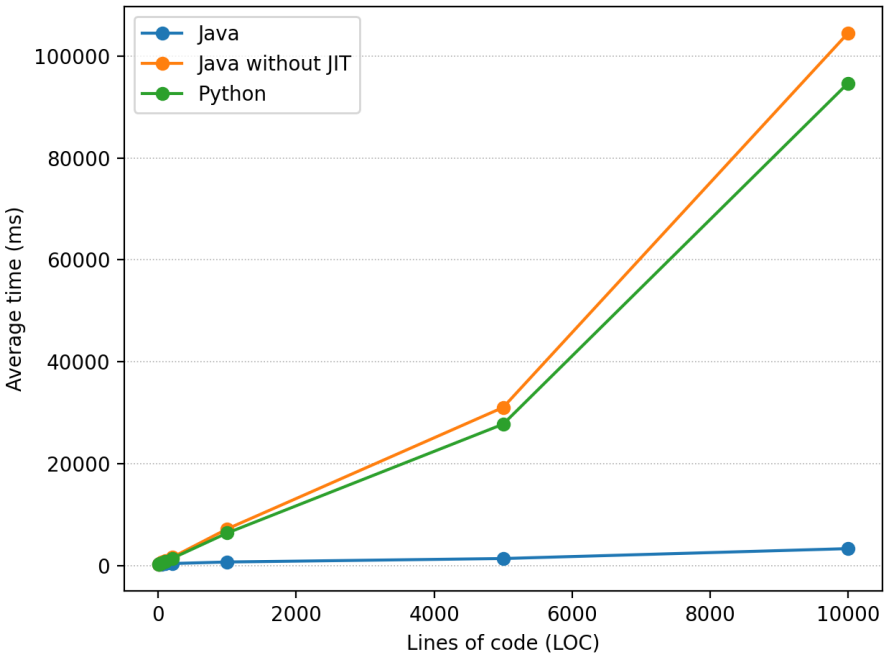


Figure 5.1: Average parsing execution time as a function of input size (LOC)

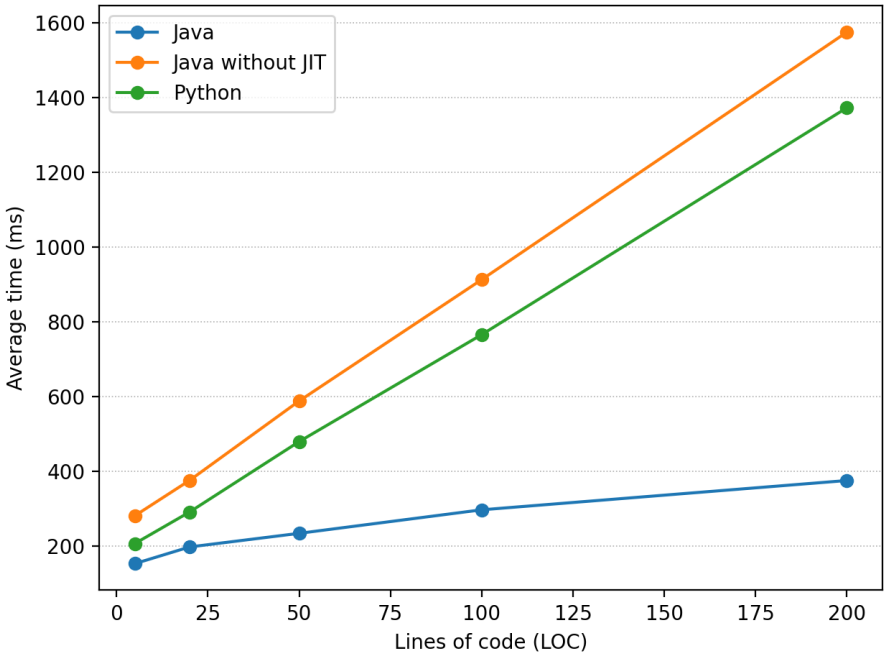


Figure 5.2: Average parsing execution time as a function of input size (LOC)

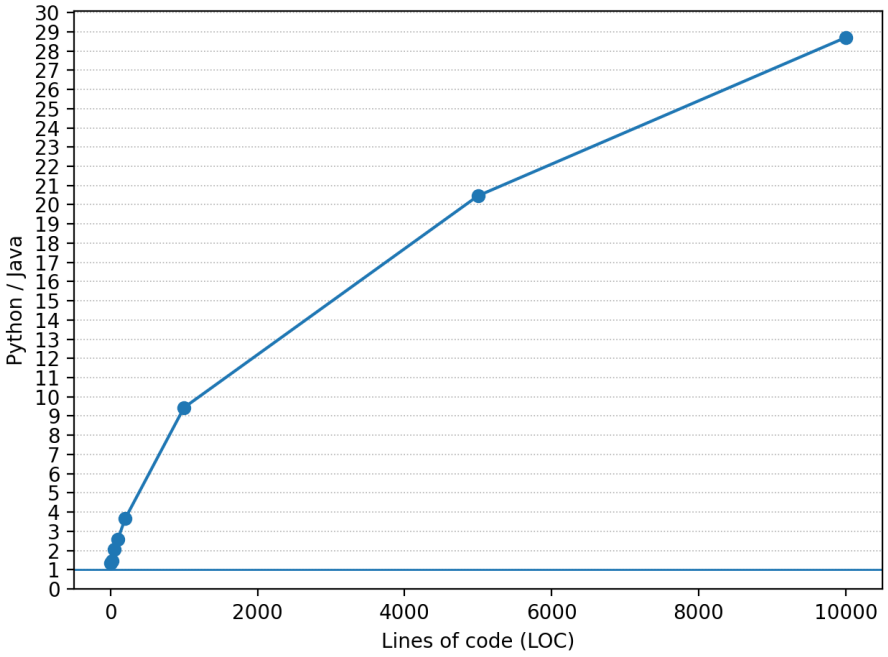


Figure 5.3: Execution time ratio (Python / Java) as a function of input size (LOC)

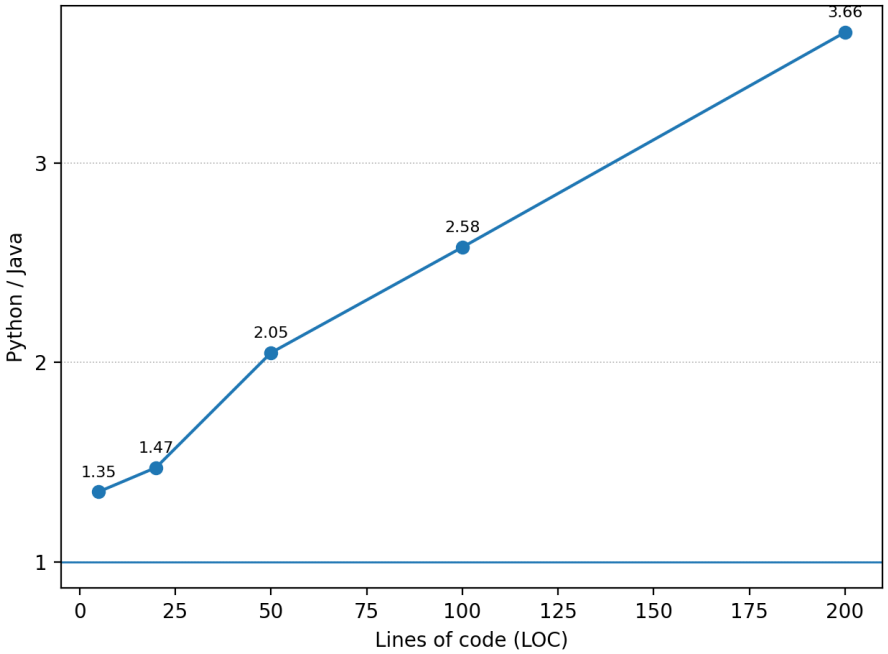


Figure 5.4: Execution time ratio (Python / Java) as a function of input size (LOC)

Batch size For batches of input files, parsing is significantly more efficient in Java than in Python (Figure 5.5). As the number of input files increases, the performance gap becomes larger: the measured improvement continues to grow with batch size rather than staying constant. Figure 5.6 reports the ratio between the two runtimes: for a relatively small batch (around 50 input files), Java is faster by a factor of about 7.5, and as the batch size increases to larger numbers of input files the improvement becomes much more pronounced, reaching a factor close to 20.

In Figure 5.5, the Java `-Xint` curve, in contrast, tracks the Python curve closely across all batch sizes, remaining within approximately 15% of Python execution time. The gap between standard Java and Java `-Xint` widens with batch size, mirroring the gap between standard Java and Python.

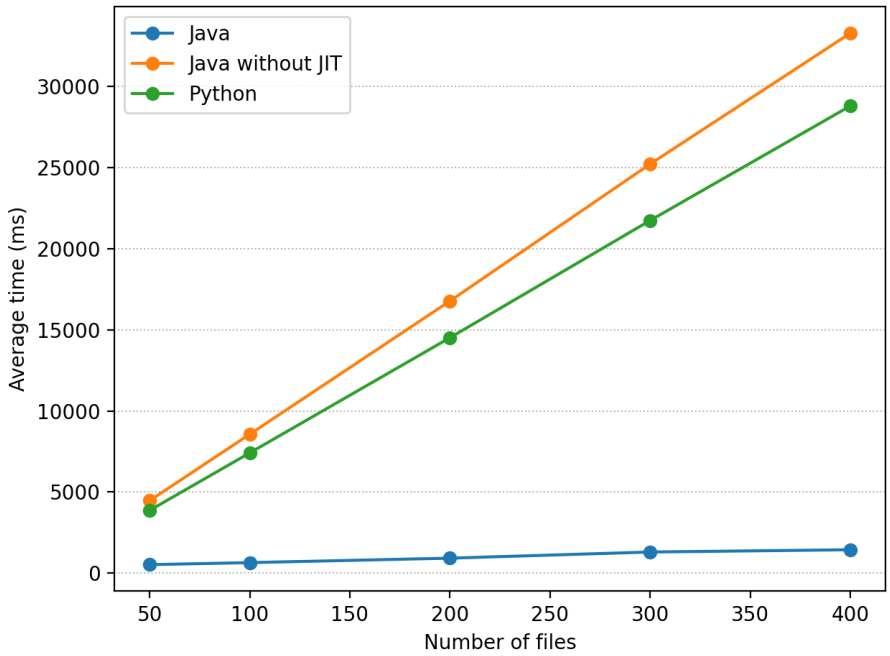


Figure 5.5: Average parsing execution time as a function of the number of input files

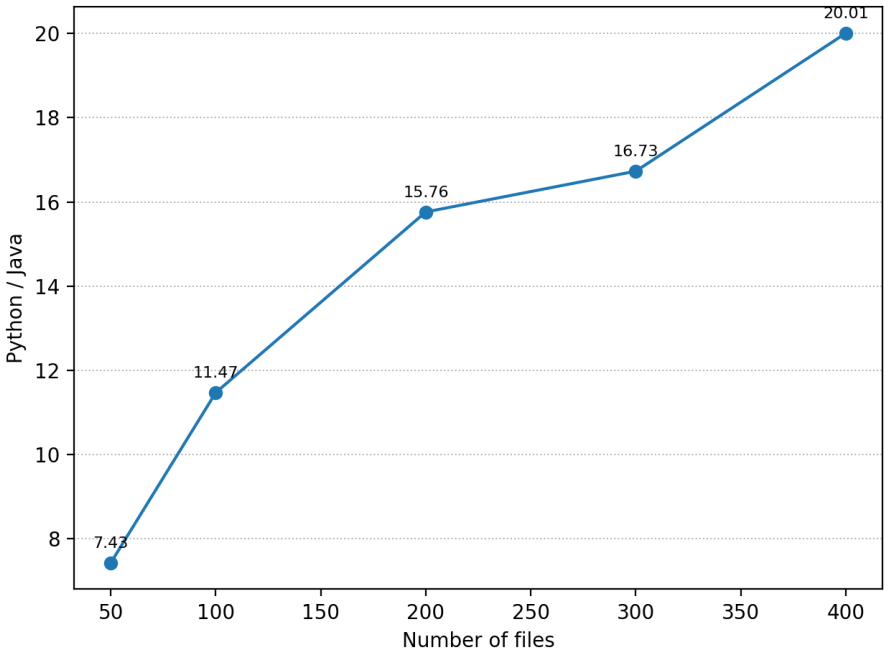


Figure 5.6: Execution time ratio (Python / Java) as a function of the number of input files

5.3 Analysis and Discussion

The speed-up is substantially larger for batch execution than for single-file execution. If total runtime were proportional to the parsing work alone, the ratio between Java and Python would stay stable when moving from a single file to a directory of many files. It does not: the gap widens with the number of files. This means part of the runtime does not scale with the number of files and is amortised differently in the two execution modes.

A first explanation is the presence of fixed runtime costs. In our benchmark methodology, each `hyperfine` measurement executes a shell command repeatedly, and `hyperfine` corrects for shell spawning time through a calibration procedure[7]. Each Java command still requires starting a fresh JVM and initialising the runtime before the application starts[3]. Python is not free of such costs either: the `heavy` startup measurements show its import mechanism making it slower than Java once classes are loaded (156 ms against 113 ms). In a single-file execution, this start-up and initialisation cost can represent a non-negligible fraction of total runtime in addition to the parsing work itself. In batch execution the JVM is started only once for the whole directory, so the same fixed cost is amortised over many files and its relative contribution decreases as batch size increases.

A second explanation is JVM warm-up within a single process. HotSpot does not execute application code uniformly throughout the run: it collects profiling information at run time and progressively optimises frequently executed code paths[4]. Oracle describes it as an adaptive compiler that analyses execution in order to identify performance-critical hot

spots and compile them into optimised native code at run time for improved performance[6]. CPython does not work this way: it compiles Python source to bytecode and runs it through an interpreter[8]. In a short single-file execution, the progressive optimisation allowed by the JIT may have limited influence on total runtime, because the JVM may terminate before repeated code paths have been profiled and optimised sufficiently. In batch execution, the same JVM instance processes many files in sequence, so those paths are exercised more often and HotSpot has more opportunity to optimise them, lowering the average processing cost per file over the batch.

These measurements also address the hypothesis that the Java ANTLR runtime is itself faster than its Python counterpart. This does not hold: when the JIT is disabled (`-Xint`), bringing Java closer to a pure interpreter like Python, the Java parser performs slightly worse than the Python parser on both benchmarks. The gap between Java and Python thus comes from the runtime, not the language.

Of the two mechanisms proposed in Section 4.2, the speed-up is therefore explained by HotSpot's adaptive compilation rather than by the maturity of the ANTLR runtime.

5.4 Threats to Validity

Three factors may limit the generality of the results reported above.

First, the batch benchmark is based on directories of student submissions for the same exercise. These inputs are therefore not fully heterogeneous and likely share recurring structural similarities. While this is consistent with the intended use case of Pyttern, it may also favour repeated execution paths during batch processing. Since the Java runtime can improve the execution of code paths that are used repeatedly as a program continues running [6, 4], such repetition may make the workload especially favourable to runtime optimisation and amplify the measured speed-up relative to a more heterogeneous corpus.

Second, the complementary benchmarks isolate JVM startup and JIT compilation, but other factors not directly measured here may also contribute to the observed gap. These include memory-management behaviour and differences in library implementation between the Python and Java ANTLR runtimes. Neither was isolated here, so part of the residual gap may stem from them.

5.5 Conclusion

The benchmarks in this chapter support the first stage of the port, namely moving the parsing step from Python to Java. The initial profiling showed that parsing dominates end-to-end runtime for most patterns. The subsequent `hyperfine`-based evaluation confirms that this design choice yields a substantial benefit, especially in batch settings where Pyttern is typically used: the observed speed-up increases with batch size and, in the present benchmark, reaches a factor close to 20 relative to the Python implementation. The complementary measurements show that HotSpot's adaptive compilation is the primary

driver of this gain, since disabling the JIT brings Java performance down to Python's level, or slightly below.

The next step is to implement the remaining stages of Pyttern in Java and to evaluate whether the resulting Java version remains at least as efficient as the Python version across the full pipeline. This will determine whether a fully Java-based solution is viable, or whether a hybrid Java–Python approach is preferable for end-to-end performance.

Chapter 6

Matching in Java: Validation

The previous chapter showed a substantial speed-up on parsing. This chapter does the same for the second stage: the full Java implementation (parsing, PDA construction and matching) is compared against the Python one, and the matcher's own contribution is isolated.

6.1 Benchmark

The purpose of this benchmark is to validate Stage 2 of the reimplementation by evaluating the performance of the full Java pipeline against the existing Python pipeline. Since parsing and PDA construction also differ in performance between the two implementations, we isolate the matcher's contribution by measuring both the parsing time alone and the total end-to-end execution time. This allows us to verify that the speed-up observed in Stage 1 is maintained when matching is included, and that the matching stage itself performs consistently across both implementations.

6.1.1 Methodology

We evaluate matching performance at two levels of granularity.

Macro-level. End-to-end execution time is measured using `hyperfine`, which repeatedly invokes each implementation from the shell. This external approach ensures that both implementations are measured in the same way. Two measurements are taken for each input: one covering parsing and PDA construction only, and one covering the full execution including matching. This allows us to directly compare the matching performance of both implementations.

Micro-level. Fine-grained measurements are collected using `JMH` for the Java implementation and `pyperf` for the Python implementation. These microbenchmarking frameworks measure the internal execution time of individual functions in isolation, reflecting the

performance as seen from within each runtime. Unlike the macro-level measurements, they exclude external factors such as process startup time or JVM initialisation. The macro and micro measurements serve as a double verification: if the observed times are proportional across both levels, we gain confidence in the consistency of the results.

For each benchmark, results are aggregated by test category and averaged, providing a structured view of performance across the different wildcard types exercised by the matcher test suite.

6.1.2 Inputs

The inputs are drawn from the matcher's own test suite, which is designed to exercise all the different wildcard types supported by Pyttern. Each test case consists of short Python files paired with a set of patterns. These patterns explore different structural positions for each wildcard type, with both patterns that match the files and patterns that do not.

6.1.3 Experimental Environment

All experiments are executed on a dedicated university virtual machine running Ubuntu 24.04.4 LTS (Linux 6.8.0-88-generic, x86_64), configured with 4 vCPUs and 4 GB of RAM. The Python implementation is executed with Python 3.12.3, and the Java implementation is executed with OpenJDK 25.0.2.

6.2 Results

Macro-level. Figure 6.1 shows the macro-level comparison between both implementations. The parsing time confirms that Java consistently parses faster than Python across all pattern groups, in line with the previous results. For `multiple_body`, the total Python runtime reaches approximately 1350 ms against approximately 230 ms for Java, reflecting the combined gain on both parsing and matching. For all other categories, the global runtime closely follows the parsing time.

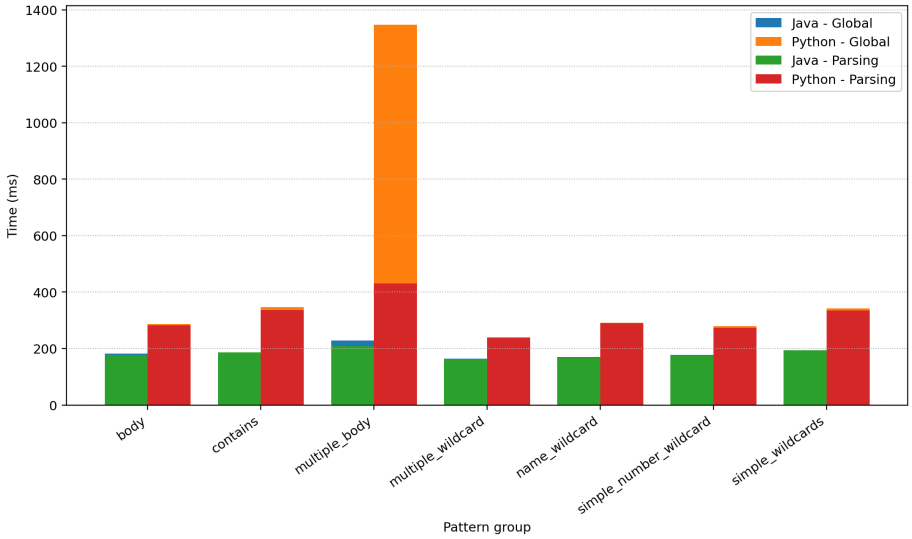


Figure 6.1: Average macro-level execution time per test category for the Java and Python implementations

Micro-level. Figures 6.2 and 6.3 show the micro-level measurements for Java and Python respectively. For both implementations, `multiple_body` stands out as a clear outlier: the matcher contribution (green) dominates the total runtime, reaching approximately 8 ms in Java and 750 ms in Python. For all other pattern groups, the matcher contribution is negligible compared to parsing and PDA construction, confirming that most wildcard types have a very low matching cost.

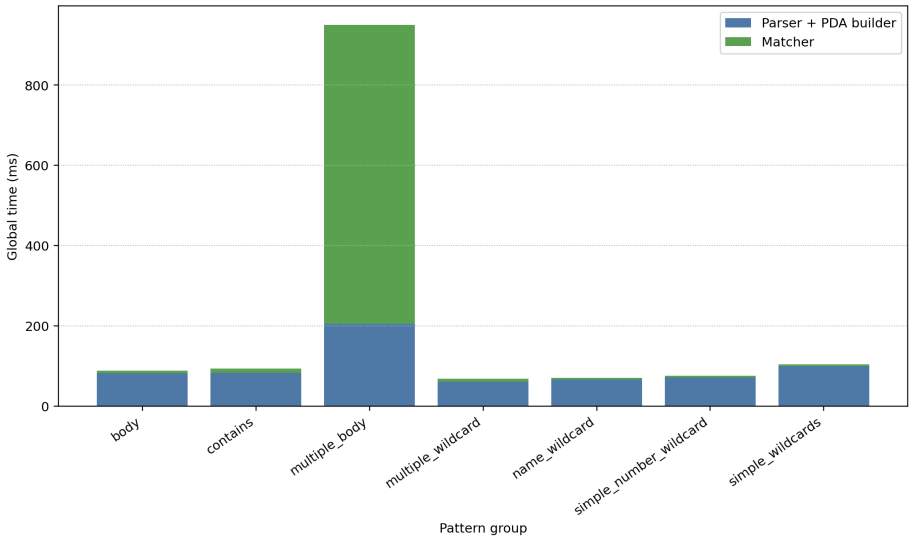


Figure 6.2: Average micro-level execution time per test category measured with `pyperf`

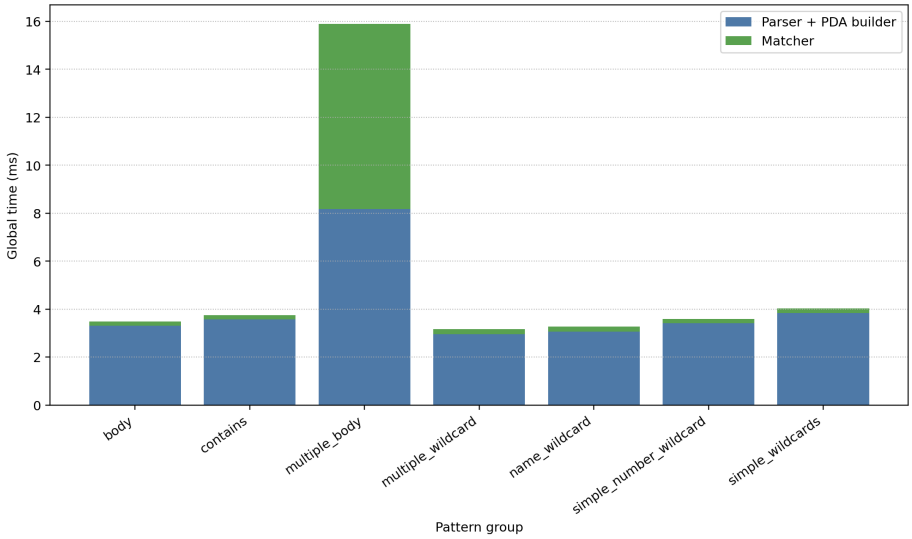


Figure 6.3: Average micro-level execution time per test category measured with `JMH`

6.3 Analysis and Discussion

Speed-up driven by parsing. For most pattern groups, the overall speed-up is primarily determined by the parsing stage, consistent with the profiling results of Chapter 4. This confirms that the gain observed in Stage 1 is maintained in the full implementation.

The multiple_body outlier. As with `hardcodedlist` in the previous chapter, the dominant matching cost for `multiple_body` is explained by the use of wildcards matching zero or more elements: the PDA must explore a combinatorially growing number of branches for each possible decomposition, causing the search space to expand rapidly. This growth is structural: it depends on the number of such wildcards in the pattern, and the cost grows exponentially with that number. In this case, matching becomes the primary cost, and the performance gap between Java and Python widens significantly; as established in the previous chapter, HotSpot's adaptive compilation gives Java a substantial advantage on code paths that are repeatedly exercised. Java reduces the constant cost; nonetheless, the complexity is unchanged and the combinatorial explosion remains.

The matcher bottleneck differs from the parsing one. Parsing time scales linearly with input size and remains practical on realistic programs, so Java's constant-factor improvement is sufficient. The matcher cost, by contrast, grows exponentially with the number of zero-or-more wildcards in the pattern, regardless of input size. Past a moderate threshold, the matcher may fail to return within an acceptable time. The test suite used here limits the number of such wildcard occurrences and stays below this threshold. Chapter 7 considers patterns that expose this limitation directly and develops dedicated optimisations.

Macro and micro consistency. The micro-level measurements do not compare Java and Python directly, since JMH and pyperf measure different internal execution contexts. They validate the macro-level observations independently for each implementation: in both cases, the relative behaviour across pattern groups at the micro level matches the macro-level results, confirming that the macro measurements reflect what happens inside each runtime.

The absolute values reported by `hyperfine` are nonetheless systematically higher than those reported by the corresponding micro-level framework. This is expected and can be attributed to the fixed overhead identified in the previous chapter. Since the test files are small by construction, this fixed cost represents a non-negligible fraction of total macro-level runtime.

6.4 Threats to Validity

The threats to validity identified in the previous chapter apply here as well. The cold-start measurement model inflates Java runtimes due to JVM initialisation, an effect that is more pronounced for the short inputs used in this benchmark. As shown in the previous chapter, this overhead is amortised in batch settings, so the reported speed-ups should be interpreted as conservative estimates of the Java implementation's true advantage.

Additionally, the input corpus is limited to the matcher's own test suite. While this ensures broad coverage of wildcard types, it does not capture the full diversity of real-world inputs. In particular, since the test files are small by construction, the parsing contribution is minimal, which may not reflect typical usage conditions where larger files are processed.

Finally, the observed consistency between macro and micro levels supports the validity of the results, but does not by itself separate the respective contributions of JVM startup, JIT warm-up, differences in library implementation, or memory-management behaviour.

6.5 Conclusion

The results of this chapter validate Stage 2 of the reimplementation. The full Java pipeline outperforms Python across all pattern groups, and the speed-up observed in Stage 1 is maintained when matching is included. For most patterns, the gain remains driven by parsing. For patterns involving wildcards that match zero or more elements, the matcher itself becomes the bottleneck. The Java implementation reduces the constant factor but not the complexity: the combinatorial explosion remains and can still make execution practically unusable. The micro-level measurements confirm the consistency of these observations at the internal execution level for each implementation. A full Java reimplementation of Pyttern is therefore viable and yields substantial performance improvements across the complete pipeline.

The next chapter addresses the combinatorial blow-up of the matcher directly, through three different optimisations.

Chapter 7

Optimisation

As shown in the previous chapter, some patterns reach a point where the matching phase takes longer than the parsing itself. Two wildcards are responsible for this cost: `?:*` and `?*`. These wildcards are particularly problematic because their length is not fixed in advance. At each occurrence, the PDA must try every possible number of elements the wildcard could match, creating one branch per choice. When several appear in the same pattern, the search space grows exponentially with their number.

Three optimisations were developed to reduce this cost. The first targets the construction of the PDA itself, eliminating a redundancy in how `?:*` is encoded. The second prunes the pattern tree before matching, removing successions of `?:*` or `?*` that are semantically equivalent to a single occurrence. The third avoids re-exploring configurations that are reached more than once during matching. The following sections describe each optimisation in turn and report its impact on a benchmark suite.

7.1 Benchmark

The purpose of this benchmark is to evaluate the impact of the three optimisations introduced in this chapter on the matcher's execution time. Since the optimisations target the two wildcards responsible for the exponential matching cost (`?:*` and `?*`), the benchmark exercises each in isolation, on patterns of varying configurations to probe their limits.

7.1.1 Methodology

We evaluate the optimisations at two levels of scope.

Global scope. The full pipeline is timed end-to-end inside a single benchmark, covering parse tree generation, PDA construction, and matching. This scope reflects the impact of an optimisation on the total cost paid by the user.

Matcher scope. The parse tree and the PDA are constructed once in the `@Setup` phase, outside the timed region, and only the call to the matcher is measured. This scope isolates the matcher's contribution.

All measurements are collected with `JMH`, configured as described in the corresponding background section.

7.1.2 Inputs

The inputs are organised into two suites, each built around one of the two wildcards responsible for the exponential matching cost. Each test case consists of a Python source file (`.py`) paired with a pattern file (`.pyt`) that the matcher is run against.

Each suite contains four patterns sharing the same code skeleton. The placement, repetition, and nesting of the targeted wildcard are varied across the patterns to expose different configurations that may trigger the exponential matching cost. The patterns are numbered continuously from 1 to 8 across the two suites, to keep the labels consistent in the result figures.

The `multiple_body` suite targets the `?:*` wildcard. Its skeleton is a `multiplications(n)` function with nested `for` loops, in which `?:*` wildcards are introduced inside the loop bodies at varying depths. The target code is shown in Listing 7.1, and the four patterns in Listings 7.2–7.5.

```
def multiplications(n):
    count = 0
    for i in range (1,n+1):
        for j in range (1,n+1):
            for j in range (1,n+1):
                for j in range (1,n+1):
                    for l in range (1,n+1):
                            if n==i*j :
                                count+=1
    return (count)
```

Listing 7.1: Target code of the `multiple_body` suite

```
def multiplications(n):
    ?*
    ?:*
        ?:*
            ?:*
                count+=1
    return (count)
```

Listing 7.2: `multiple_body` pattern 1

```
def multiplications(n):
    count = 0
    for i in range (1,n+1):
        for j in range (1,n+1):
            ?:*
                ?:*
                    if n==i*j :
                        count+=1
    return (count)
```

Listing 7.3: `multiple_body` pattern 2

```
def multiplications(n):
    ?*
    for i in range (1,n+1):
        ?:*
            ?:*
                for j in range (1,n+1):
                    ?:*
                        ?:*
                            ?:*
                                if n==i*j :
                                    count+=1
    return (count)
```

Listing 7.4: `multiple_body` pattern 3

```
def multiplications(n):
    ?*
    for i in range (1,n+1):
        ?:*
            for j in range (1,n+1):
                ?:*
                    if n==i*j :
                        count+=1
    return (count)
```

Listing 7.5: `multiple_body` pattern 4

The `multiple_wildcard` suite targets the `?*` wildcard. Its skeleton is a `test()` function ending with a list assignment, in which `?*` wildcards are introduced both in the statements above the list and inside the list itself. The target code is shown in Listing 7.6, and the four patterns in Listings 7.7–7.10. For readability, long enumerations of consecutive integers are abbreviated; the actual benchmark uses the full enumeration.

```
def test():
    count1 = 1
    count2 = 2
    count3 = 3
    list1 = [1, 1, 1, ..., 1]  # 23 ones
    count4 = 4
    list = [1, 2, 3, ..., 100]
```

Listing 7.6: Target code of the `multiple_wildcard` suite

```
def test():
    count1 = 1
    count2 = 2
    count3 = 3
    list1 = [1, 1, 1, ..., 1]
    count4 = 4
    list = [1, 2, 3, ?*, 15, 16, 17, ?*, 30, ?*, 45, 46, ?*,
    70, ?*, 79, ?*, ?*, 88, 89, 90, ?*, 100]
```

Listing 7.7: `multiple_wildcard` pattern 5

```
def test():
    count1 = 1
    count2 = 2
    count3 = 3
    list1 = [?*, 1, ?*, 1, ?*, 1, ?*, 1, ?*, 1, ?*, 1, ?*, 1,
    ?*, 1, ?*, 1, ?*, 1]
    count4 = 4
    list = [1, 2, 3, ..., 100]
```

Listing 7.8: `multiple_wildcard` pattern 6

```
def test():
    count1 = 1
    ?*
    ?*
    ?*
    list = [1, 2, 3, ..., 100]
```

Listing 7.9: `multiple_wildcard` pattern 7

```
def test():
    count1 = 1
    ?*
    ?*
    ?*
    count4 = 4
    list = [1, 2, 3, ?*, 15, 16, 17, ?*, 30, ?*, 45, 46,
    ?*, 70, ?*, 87, ?*, ?*, ?*, 100]
```

Listing 7.10: `multiple_wildcard` pattern 8

7.1.3 Experimental Environment

All experiments are executed on a dedicated university virtual machine running Ubuntu 24.04.4 LTS (Linux 6.8.0-88-generic, x86_64), configured with 4 vCPUs and 4 GB of RAM. The Java implementation is executed with OpenJDK 25.0.2.

7.2 PDA Optimisation

The first of the three optimisations targets the construction of the PDA itself. The analysis of the portion of the PDA generated for `?:*` reveals a redundancy in how the first-entry case is encoded. This redundancy causes a large exponential blow-up when several `?:*` wildcards are traversed during matching. This section begins by showing how the wildcard

is represented in the PDA, then explains which transition introduces the problem, proposes a correction, and finally discusses the performance gains observed on a benchmark.

7.2.1 PDA Representation of `?:*`

Compiling `?:*` produces the portion of the PDA depicted in Figure 7.1. The entry transition pushes a dedicated symbol B onto the stack and leads to a dispatch state, from which the automaton chooses between the two semantic cases of the wildcard. The first case corresponds to matching one or more compound blocks. It is represented by a route that passes through an intermediate traversal state. From this state, the automaton has two options: it can move horizontally to the next block at the same indentation level through a self-loop, or it can descend vertically into the body of the current block through a back transition that pushes a second symbol I onto the stack and returns to the dispatch state. This route reaches the exit state through a transition conditioned by the node class `Block`, which can only be taken when the matcher is positioned on an indented block in the code. The second case corresponds to matching zero blocks and is handled by a skip transition leading directly from the dispatch state to the exit state. Both routes terminate at the same exit state.

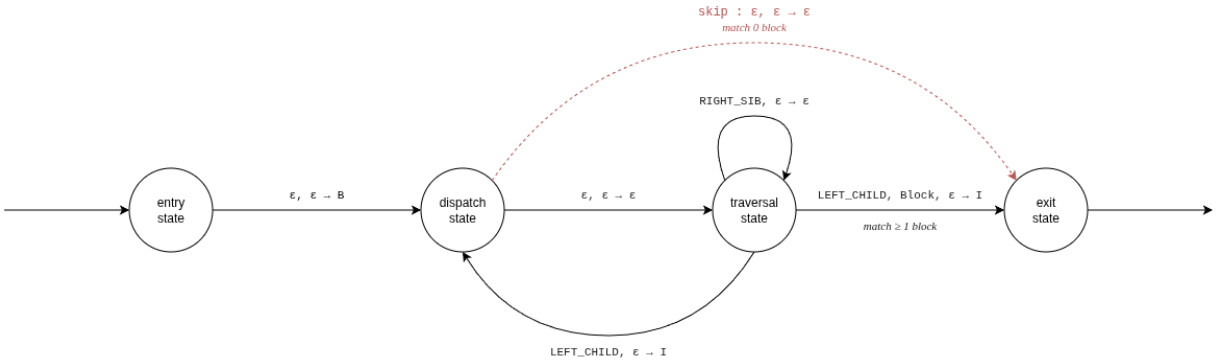


Figure 7.1: Portion of the PDA generated for the `?:*` wildcard. From the dispatch state, two routes reach the exit state: a route through the traversal state gated by a transition conditioned by the node class `Block` (matching one or more compound blocks), and a skip transition taken when no block is to be matched.[2]

7.2.2 Problem

The issue does not appear when the wildcard is entered for the first time. At that point, the matcher follows the appropriate transition depending on whether a concrete `Block` node is present in the code. If no such node is available, the skip transition is taken; otherwise, the matcher proceeds through the `Block`-conditioned transition.

[2]XML layout produced with the assistance of Claude/ChatGPT.

The problem arises when the dispatch state is re-entered. After descending into a child subtree via the back transition, which pushes I on top of B, the automaton returns to the dispatch state to process the inner level. At that point, the skip transition is still enabled, since it has no condition on the stack. As a result, the automaton may leave the wildcard immediately, even though it can also continue along the `Block`-conditioned transition. These two executions eventually reach the same final configuration, as illustrated in Figure 7.2.

Consequently, each valid match is produced twice. More generally, every traversal of a `?:*` wildcard doubles the number of equivalent traces, so the total number of such traces grows exponentially with the number of `?:*` wildcards encountered during matching.

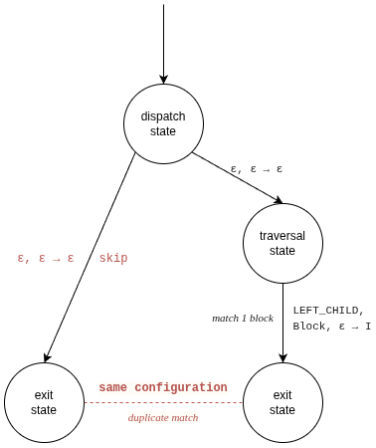


Figure 7.2: Convergence of the two transitions once the dispatch state is re-entered. With I on top of B, the skip transition is still enabled and can be taken even though the `Block`-conditioned transition is also applicable. Both executions eventually reach the same final configuration, duplicating every valid match[4].

7.2.3 Fix

The fix adds a condition on the stack to the skip transition: B must be at its top for the transition to be taken. This condition holds only at first entry into the wildcard, where B has just been pushed, and is violated after any descent, where I sits on top of B. In practice, the transition reads B from the top of the stack and writes it back, so the stack is left unchanged while the transition becomes unavailable whenever I is on top. The corrected portion of the PDA is shown in Figure 7.3. The language accepted by the PDA is unchanged, but duplicate matches no longer occur and the exponential blow-up disappears.

[4]XML layout produced with the assistance of Claude/ChatGPT.

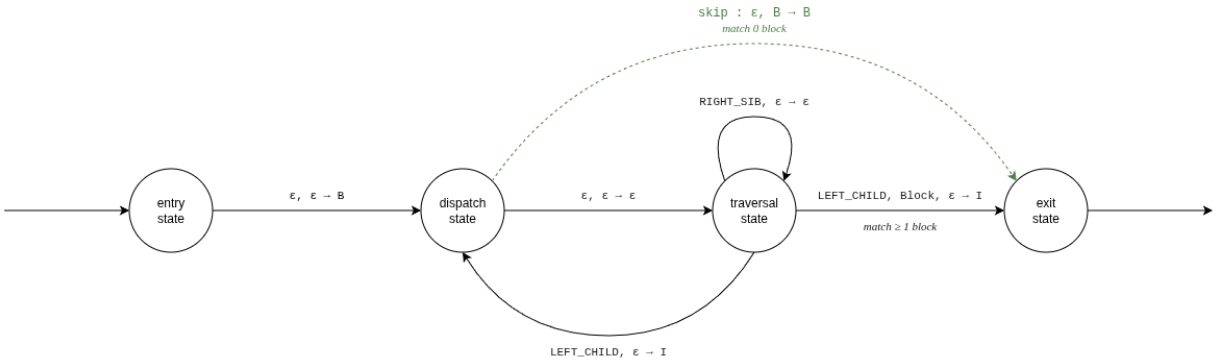


Figure 7.3: Corrected portion of the PDA. The skip transition now reads B from the top of the stack and writes it back, so that it can only be taken at first entry into the wildcard. After a descent has pushed I, the condition is violated and the skip transition is suppressed[6].

7.2.4 Performance

To assess the practical impact of the correction, we evaluated it on four patterns differing in the maximum number of `?:*` wildcards nested directly inside one another, with no concrete construct in between. The baseline implementation (global) is compared against the version using the corrected PDA construction (Opti PDA); the results are shown in Figure 7.4.

The correction improves execution time across the entire benchmark suite, but the magnitude of the improvement depends on the length of the longest uninterrupted chain of nested `?:*` wildcards. On pattern 4, representative of typical wildcard usage, where each `?:*` is separated from the next by a concrete construct, the gain is about 12%. By contrast, on the other patterns containing uninterrupted chains of `?:*`, the speed-ups reach factors of 48, 2.4, and 207 for patterns 1, 2, and 3 respectively.

Such chains are not expected in well-written patterns, since a single `?:*` already captures the same semantics. They may nevertheless arise in user-written queries, especially when the author is less familiar with the language, which is why they are included in the benchmark.

These results match the expected behaviour. In the baseline implementation, each traversal of a `?:*` doubled the number of equivalent traces, so the gain from removing the duplication grows with the length of the chain.

[6]XML layout produced with the assistance of Claude/ChatGPT.

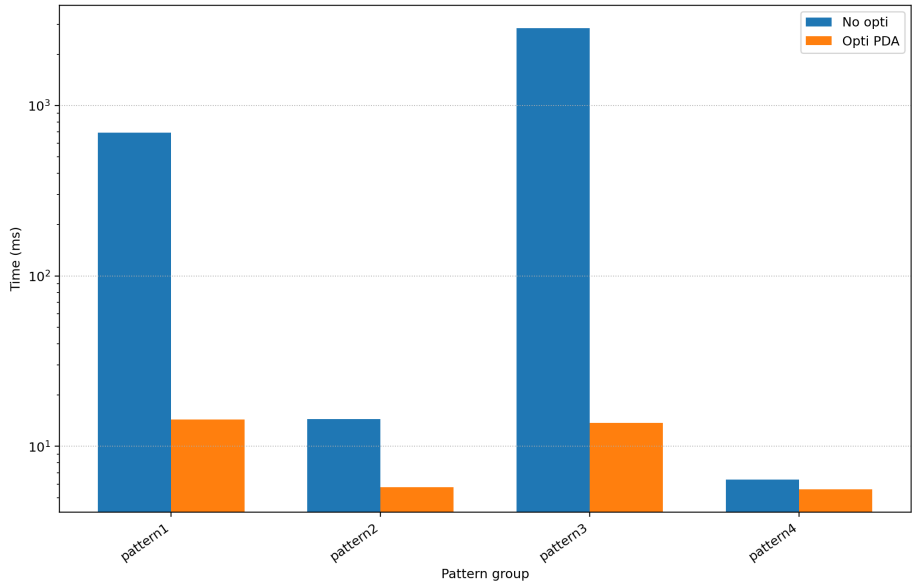


Figure 7.4: Execution time of the different versions on the `multiple_body` benchmark suite. Log scale on the y-axis.

7.3 Pruning

The second optimisation acts on the parse tree of the pattern, before the PDA is constructed. Since both `?:*` and `?*` match zero or more elements, a sequence of consecutive `?:*` or `?*` wildcards is equivalent to a single occurrence. Such redundant wildcards do not change the result, but each of them adds a non-deterministic branch in the PDA, which multiplies the set of configurations the matcher has to explore. Removing them at the parse tree level eliminates this cost at the source, before the PDA is even built.

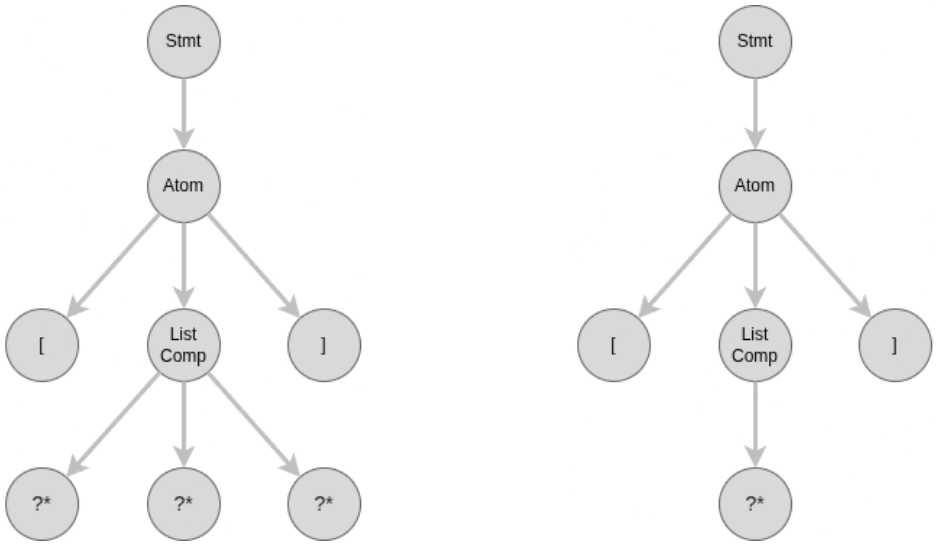
7.3.1 Integration

Pyttern already includes a visitor, `Tree_Pruner`, applied to the parse tree produced by ANTLR before it is handed to the PDA construction step. Its initial purpose was to simplify the tree by discarding non-semantic tokens and flattening grammatical structures that are more detailed than the matching process requires. The resulting tree is smaller and easier to handle in the subsequent stages. The pruning of redundant wildcards naturally extends this visitor with additional rewriting rules.

7.3.2 Rules

The additional rules detect consecutive wildcards of the same kind and keep only one of them, removing the redundant occurrences. The effect of this mechanism on a representative case is illustrated in Figure 7.5.

Three such rules are needed, one for each position in the pattern where consecutive wildcards can appear. The first rule targets `?*` wildcards stacked as successive statements inside a block. The second rule targets `?*` wildcards placed side by side in a list of parameters or arguments. The third rule targets `?:*` wildcards nested directly inside one another.



(a) Before pruning. (b) After pruning.

Figure 7.5: Effect of the pruning rules on the parse tree of a list containing consecutive `?*` wildcards. The left tree is produced after the base simplifications of `Tree_Pruner`; the right tree is obtained after the additional pruning rule has been applied[8].

7.3.3 Performance

The practical impact of the optimisation was measured on two benchmark suites. The first suite is the same as in Section 7.2, built around patterns that contain nested `?:*` wildcards. The second suite, displayed in Listings 7.7–7.10, is built around patterns that contain consecutive `?*` wildcards inside lists. The baseline implementation (global) is compared against the version including the pruning rules (Opti Pruner). The results on both suites are reported in Figures 7.6 and 7.7.

On patterns 1, 2, 3, 5 and 8, where at least one of the three rules applies, the gain is substantial. On the first suite, patterns containing nested `?:*` wildcards are reduced by two orders of magnitude or more, with speed-ups by a factor of 130 and 433 on patterns 1 and 3 respectively. On the second suite, patterns 5 and 8, which contain sequences of consecutive `?*` inside a list, show speed-ups by a factor of 198 and 237. On patterns 6 and 7, where every wildcard is already isolated from its neighbours, no rule applies and the execution time is essentially unchanged.

[8]XML layout produced with the assistance of Claude/ChatGPT.

The mechanism described above accounts for this: each redundant wildcard produces in the PDA an additional non-deterministic branch that multiplies the set of configurations explored by the matcher. Removing it before the PDA is built eliminates the branching at the source, and the gain grows with the number of redundant wildcards that can be collapsed.

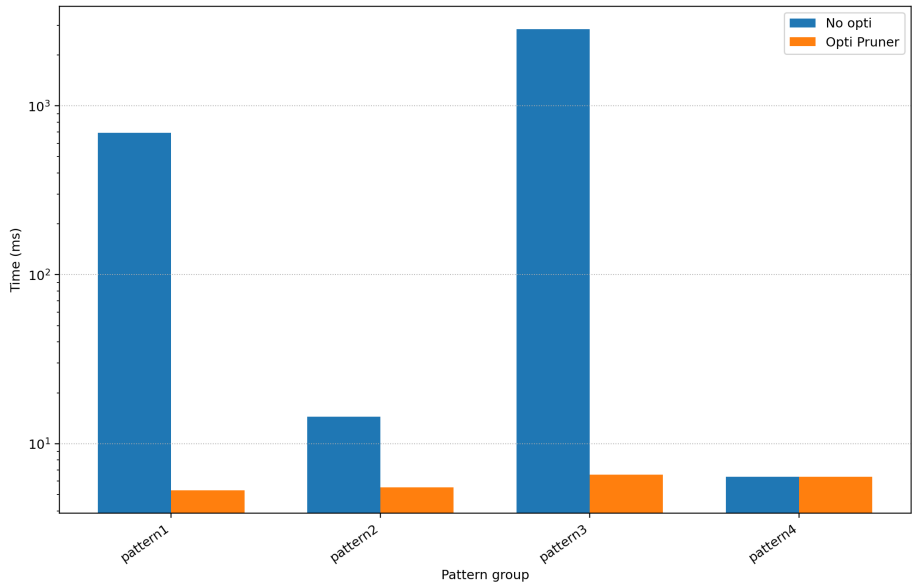


Figure 7.6: Execution time of the different versions on the `multiple_body` benchmark suite. Log scale on the y-axis.

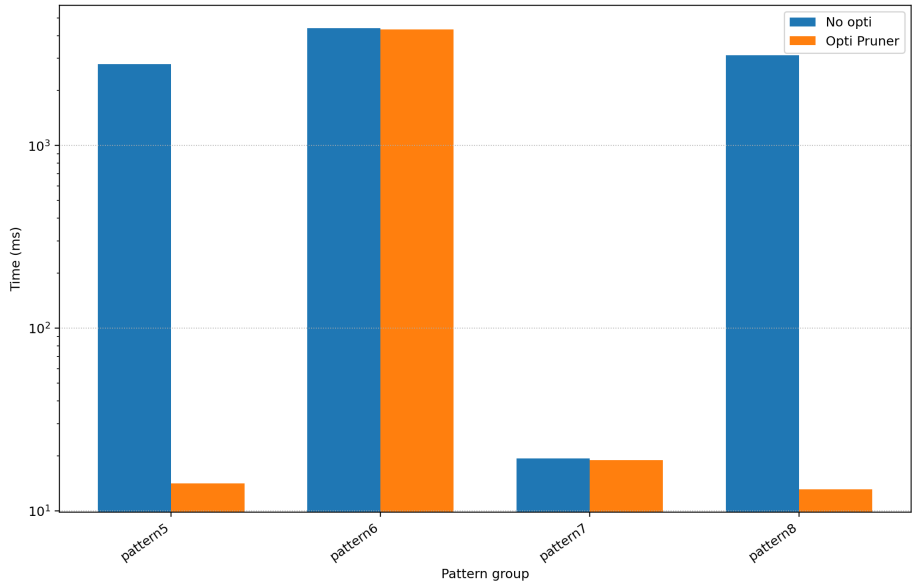


Figure 7.7: Execution time of the different versions on the `multiple_wildcard` benchmark suite. Log scale on the y-axis.

7.4 Reducing redundant exploration during matching

The third optimisation acts on the matcher itself. During the exploration of the PDA, the same configuration can be reached multiple times through different sequences of transitions, leading to redundant work. A natural starting point is memoisation, a classical technique that stores the outcome of a computation the first time and reuses it on subsequent occurrences. This section explores three variants and presents the one eventually retained.

7.4.1 Naive Memoisation

The most natural form of memoisation would keep a set of configurations already seen and skip any configuration that appears a second time. Although standard in graph exploration, this approach leads to incorrect results in the matcher. A given configuration can legitimately contribute to several distinct matches: the same intermediate state can be reached through different sequences of choices that will eventually diverge. Skipping a configuration on its second occurrence cuts off every match that would have been produced from that point on, and the matcher returns fewer matches than expected.

7.4.2 Dead-end Detection

A second approach takes the opposite perspective. Instead of memoising configurations that have been successfully explored, the matcher memoises only the configurations that have been proven to lead nowhere. A configuration is added to this set once all of its outgoing

transitions have been explored without producing any descendant that can reach the final state of the PDA. Whenever the matcher later encounters a configuration already in this set, it skips it immediately.

A configuration is only marked as a dead end once its entire subtree of exploration has been shown to produce no match. Since a configuration captures the full state of the matcher, two distinct paths that converge on the same configuration share the same future behaviour, and if one produces no match, neither will the other. A dead end must first be fully explored before it can be reused, so the optimisation speeds up repeated explorations rather than preventing the first one.

7.4.3 Counting Convergent Paths

The retained approach preserves the benefit of memoisation while keeping the correct number of matches. Instead of discarding a configuration when it is reached a second time, the matcher stores the fact that several distinct paths have converged to the same configuration. The configuration itself is kept only once in the queue, but an associated counter records how many paths currently lead to it.

The matcher maintains two auxiliary structures in addition to the queue of configurations. The first one associates each pending configuration with the number of paths that have reached it. The second one records which configurations are already present in the queue. As a result, when a path reaches a configuration that is already waiting in the queue, the matcher does not enqueue a duplicate; it only increments the corresponding path counter.

When a configuration is popped from the queue, it is removed from both auxiliary structures. At that point, the matcher retrieves the number of paths that had converged to this configuration. The configuration is then processed once, but its effects are multiplied according to the counter. In particular, if the configuration reaches a final state, the resulting match is reported once for each converging path. This reproduces the result that would have been obtained by exploring each path independently, while avoiding duplicate work as long as the configuration is pending.

Note that removing the configuration does not mark it as permanently visited. If the same configuration is reached again later, after it has already been processed, it can be inserted into the queue with a new path count. The mechanism only merges configurations that are pending at the same time; it does not prevent a configuration from being processed again if it is reached later.

The two approaches, counting convergent paths and dead-end detection, were also evaluated together in a single matcher. The execution times of the combined variant were essentially equivalent to those of counting convergent paths alone. The dead-end set brings no additional benefit. Counting convergent paths was therefore retained as the final solution.

7.4.4 Performance

The impact of the matcher optimisation was measured on the same two suites as the previous sections, comparing the baseline matcher against the convergent-paths variant (opti_pendingWays). Naive memoisation is reported alongside for reference, with the caveat already discussed that it does not preserve the number of matches.

On the `multiple_body` suite, reported in Figure 7.8, counting convergent paths produces a speed-up close to a factor of 4 on the matcher stage across all four patterns. Patterns 1 and 3 are dominated by nested `?:*` wildcards, and as the most time-consuming patterns of the suite, they drop respectively from about 690 ms to 170 ms and from 2.9 s to 770 ms.

On the `multiple_wildcard` suite, reported in Figure 7.9, the gain varies much more with the pattern. Three of the four patterns show a significant speed-up: pattern 5 drops from about 3.7 s to under 200 μs (more than four orders of magnitude), pattern 8 from 3.2 s to 29 ms (a factor of about 110), and pattern 6 from 4.3 s to about 1 s. Pattern 7 executes below the millisecond in the baseline and is essentially unchanged.

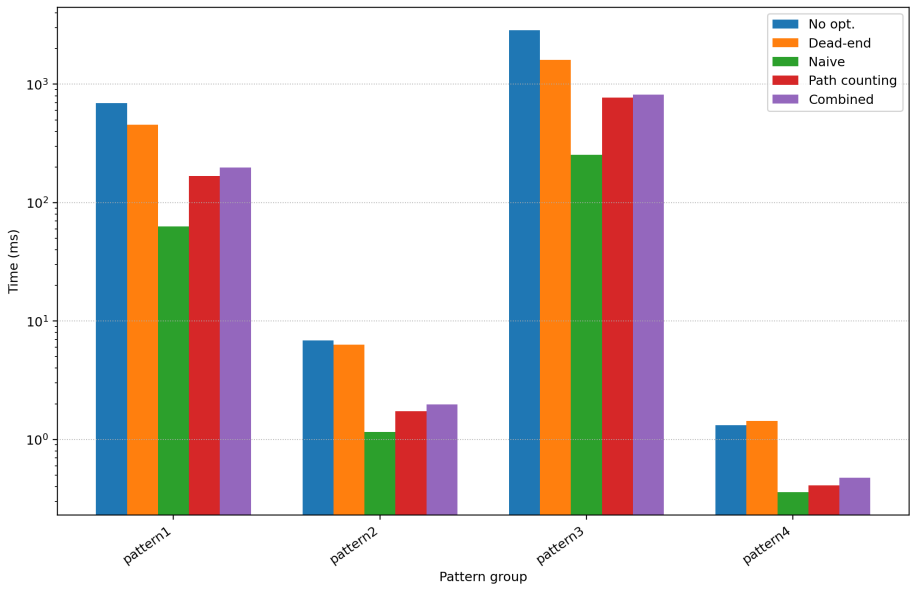


Figure 7.8: Execution time of the different matcher variants on the `multiple_body` benchmark suite. Log scale on the y-axis.

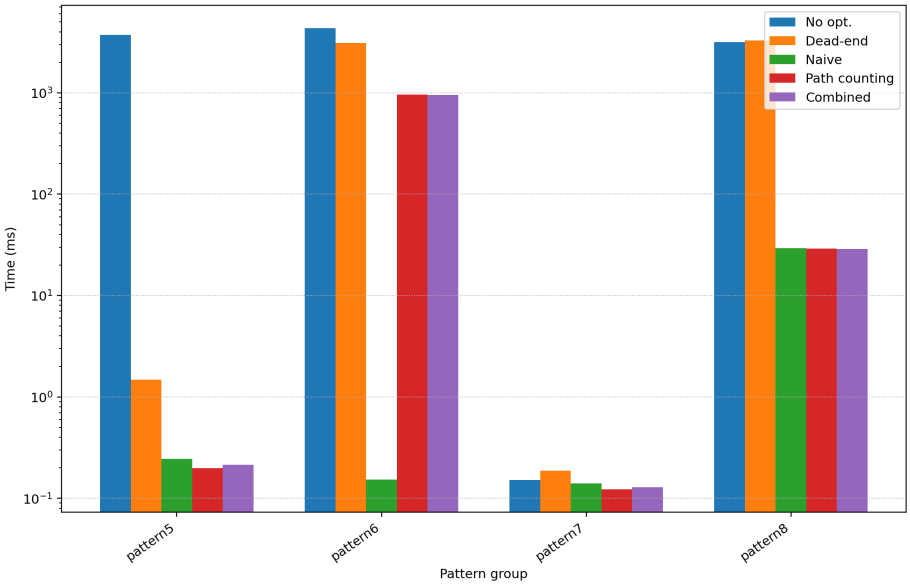


Figure 7.9: Execution time of the different matcher variants on the `multiple_wildcard` benchmark suite. Log scale on the y-axis.

7.5 Optimisation Interference

As the three optimisations target different sources of inefficiency, we would expect their gains to combine multiplicatively. In practice, their effects overlap: the cumulative gain on the benchmark suites stays close to that of the single most effective optimisation rather than reaching the product of the three. This section analyses how the optimisations interfere with one another and identifies the cases where one of them is sufficient to capture the entire gain.

The pruning optimisation acts at the source: by removing redundant wildcards from the pattern before the PDA is built, it also removes the redundant non-deterministic branches that the other two optimisations would otherwise have to handle. On the `multiple_body` suite, reported in Figure 7.10, the PDA optimisation alone produces two to three orders of magnitude of speed-up on the patterns with nested `?:*` wildcards, but once the pruner has already removed those nested wildcards, the PDA optimisation corrects far fewer wildcards. The final combination is therefore essentially equivalent to the pruner alone. The matcher optimisation is masked in the same way: the pruner greatly reduces the number of paths explored, and the number of convergent paths decreases accordingly.

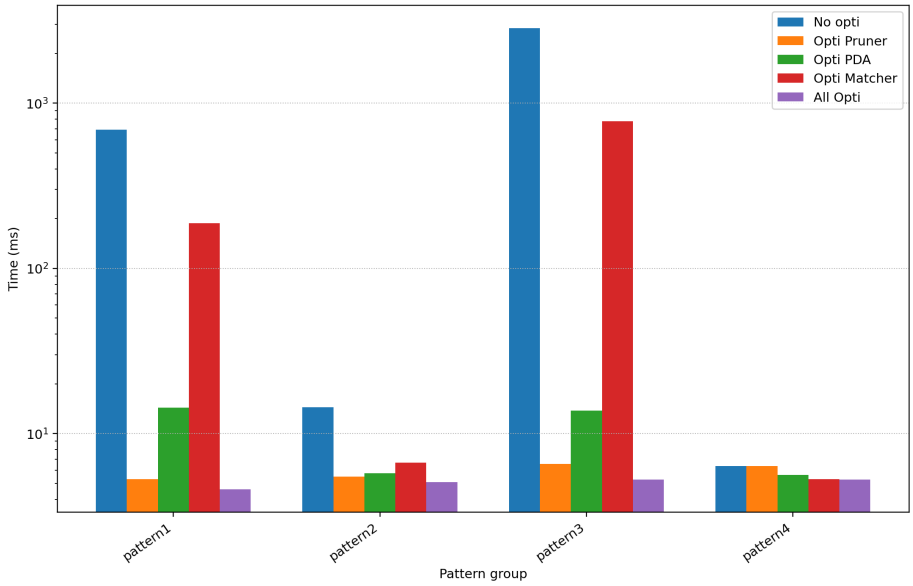


Figure 7.10: Execution time of the different optimisation variants on the `multiple_body` benchmark suite. Log scale on the y-axis.

The `?*` wildcard is not affected by the PDA optimisation, which only targets the encoding of `?:*`. Its cost is controlled almost entirely by the pruner, which collapses consecutive `?*` occurrences before they reach the matcher. Figure 7.11 shows this clearly on the `multiple_wildcard` suite: on patterns 5 and 8 where consecutive `?*` appear, the pruner alone brings the execution time down to a few milliseconds, while the PDA optimisation leaves it unchanged. Pattern 6 of the `multiple_wildcard` suite resists most of the optimisation effort: the `?*` wildcards appear in a list but are separated by concrete elements, so the pruner cannot collapse them, and the combinatorial structure of the pattern produces a very large number of legitimate matches against the target code. The matcher optimisation still helps, since intermediate transitions toward those distinct matches do share convergent paths, and execution time falls from about 4.3 s to about 1 s. The optimisations eliminate redundant exploration, but they cannot compress a combinatorial explosion that belongs to the pattern itself.

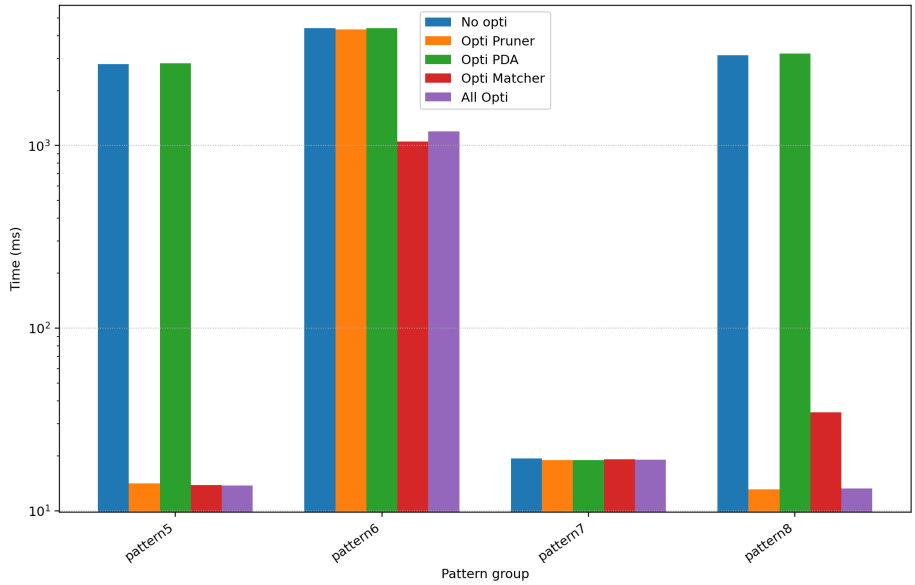


Figure 7.11: Execution time of the different optimisation variants on the `multiple_wildcard` benchmark suite. Log scale on the y-axis.

Across both benchmark suites, the combined version of the matcher brings execution time down to a few milliseconds on every pattern except the irreducible case mentioned above. The three optimisations do not combine multiplicatively, but they cover complementary situations: the pruner handles redundant wildcards at the source, the PDA optimisation handles the `?:*` wildcards that remain after pruning, and the sharing of convergent paths reduces the cost of the exploration that is still carried out.

7.6 Performance

The three optimisations described above were designed to address a pathological case identified in Chapter 6, in which the matcher of the `multiple_body` pattern group was taking longer than the parsing step. Figure 7.12 shows the execution time of the baseline implementation across the pattern groups used in the general benchmark, decomposed into the time spent parsing and building the PDA on one hand, and the time spent matching on the other. On all groups except `multiple_body`, the matcher represents only a small fraction of the total execution time; on `multiple_body`, it is of the same order of magnitude as the rest of the pipeline and drags the total execution time up to about 16 ms, making this group stand out from the others.

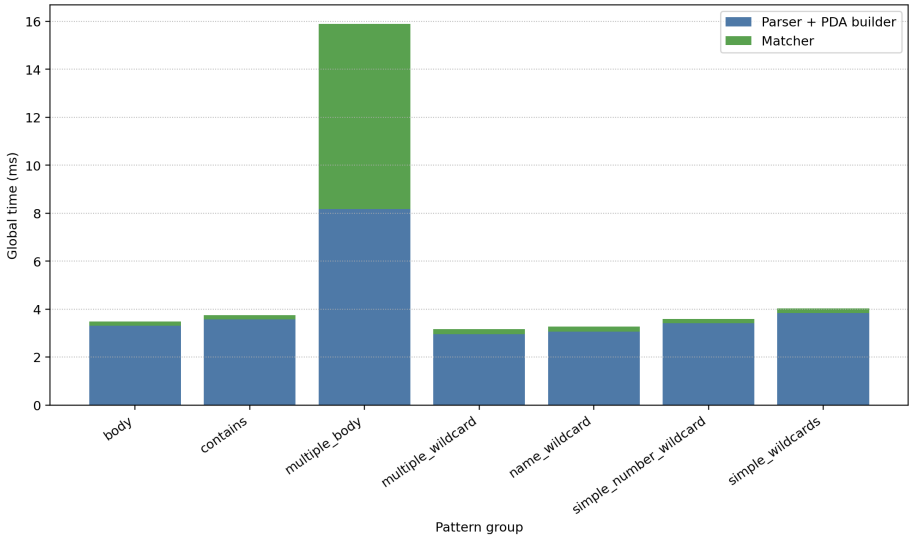


Figure 7.12: Average micro-level execution time per test category measured with `JMH`

Once the three optimisations are combined, the situation on `multiple_body` is resolved. The matcher's execution time on this group drops from about 7.7 ms to about 0.3 ms, a speed-up by a factor of roughly 25, and the total execution time of the group falls from 16 ms to 7 ms, essentially aligning it with the other groups. The matcher ceases to be the dominant cost on this group and the remaining time is spent on parsing and PDA construction, as on the other groups. Figure 7.13 shows the execution time of the fully optimised version across the same groups.

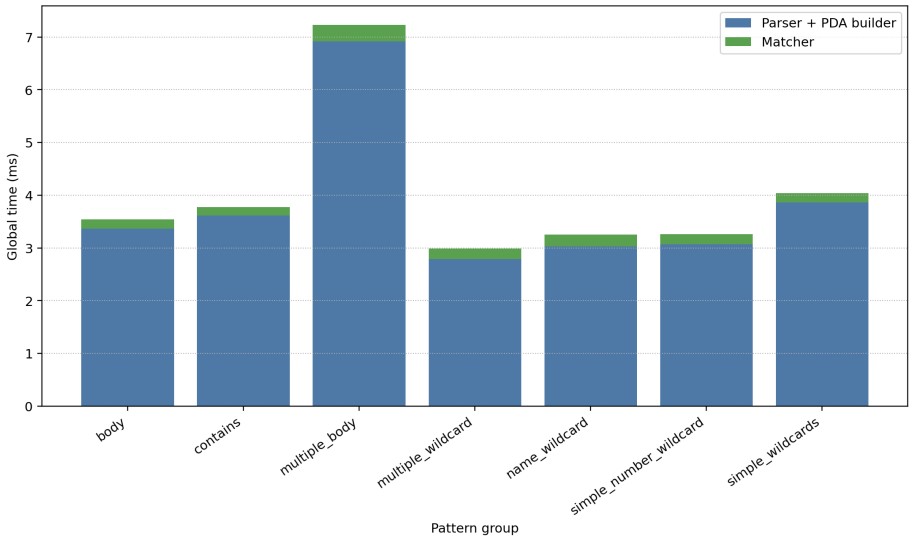


Figure 7.13: Average micro-level execution time per test category measured with `JMH`, with all optimisations enabled.

7.7 Conclusion

This chapter identified two wildcards, `?:*` and `?*`, as the source of an exponential matching cost responsible for making `multiple_body` the only outlier of the general benchmark. Three optimisations were introduced to address this cost: a correction to the PDA encoding of `?:*`, a pruning step that collapses consecutive `?:*` or `?*` wildcards before the PDA is built, and a matcher variant that counts convergent paths to avoid re-exploring configurations reached more than once.

Combined, these optimisations reduce the matcher cost on `multiple_body` from about 7.7 ms to 0.3 ms, bringing the total execution time of the group from 16 ms down to 7 ms and aligning it with the other pattern groups. The benchmark now behaves consistently across all groups.

One pattern of the benchmark remains slower than the others after optimisation: its cost is dominated by the enumeration of a large number of legitimate matches, an irreducible component that no form of redundancy elimination can address.

Chapter 8

Conclusion

This thesis set out to make Pyttern fast enough to no longer disrupt the user workflow on realistic batches of student submissions, without changing its matching semantics. Two bottlenecks were identified: parsing the input program, and matching certain patterns whose structure forces a combinatorial exploration. The work has addressed both.

Contributions

The full Pyttern pipeline has been reimplemented in Java, preserving the original ANTLR grammar, the pushdown automaton construction, and the matching semantics. The port was carried out in two stages: first the parsing stage alone, to confirm that the change of runtime would yield a substantial speed-up; then the PDA construction and the matcher, while preserving the conceptual sequence parse tree, PDA, matching result. A complementary configuration with HotSpot's JIT disabled was used to determine whether the gain comes from the runtime or from the language itself.

Three optimisations have been developed on top of that to address the residual matching cost: a fix in the pushdown automaton that eliminates duplicate matches caused by an under-constrained skip transition on the `?:*` wildcard; an extension of Pyttern's tree pruner with three rules that collapse consecutive wildcards before the automaton is built; and a matcher that shares the exploration of paths converging on the same pending configuration without altering the match count.

Results

Java port. On parsing, the port brings a speed-up of about a factor of 2.5 on individual files of typical size, and up to a factor of 20 in batch settings, where the JVM startup cost is amortised over many files. A complementary configuration with HotSpot's JIT compiler disabled shows that the language change alone does not account for this gain; most of it comes from HotSpot's adaptive compilation, while the maturity of the Java ANTLR runtime contributes nothing measurable on this workload. On matching, the port reduces

the constant factor but leaves the combinatorial structure of the problem untouched, leaving one pattern group, `multiple_body`, as a clear outlier where matching dominates the total runtime.

Matching optimisations. The three optimisations bring matching time down by several orders of magnitude on the most affected patterns. The most redundant patterns of the pruning benchmark show gains of two orders of magnitude or more, and the patterns with the longest chains of nested `?:*` on the PDA benchmark show a similar factor. On the general matcher benchmark, `multiple_body` drops from approximately 16 ms to 7 ms in total runtime and is now aligned with the other groups, where matching is negligible compared to parsing. The other groups show either no measurable change or a small improvement. The three optimisations do not combine multiplicatively, since the pruner acts at the source and masks part of what the other two would otherwise correct, but each remains useful on patterns where the others leave redundancies behind. One pattern of the benchmark resists most of the optimisation effort: its execution time remains close to one second after optimisation, because it produces a combinatorial number of legitimate matches that have to be enumerated, and no amount of sharing or pruning can reduce that work.

Objectives revisited

The qualitative goal stated in the introduction was that typical workloads should complete within a few seconds. On the workloads covered by this thesis, this goal is met: parsing is fast enough on batches of realistic size, and the matcher no longer dominates on the pattern groups that previously made it unusable. One combinatorial case characterised in Chapter 7 remains above this threshold.

Chapter 9

Lessons learned

Porting a Python codebase to Java is rarely a direct translation, even when both implementations rely on the same external tools. Several observations made during this work may help anyone undertaking a similar effort.

ANTLR is not the same library in both languages. The Python and Java runtimes of ANTLR share the same grammar format and the same general structure, but they diverge in details that matter in practice. The class hierarchies of the parse-tree nodes are not identical between the two runtimes, which makes casts and instance-of checks less directly transposable from one to the other. Error handling also differs significantly: a syntactically invalid program will crash the Python runtime but be tolerated by the Java one. Even when the same tool is used on both sides, its implementation can differ enough to require careful adaptation.

Typing is the main source of friction. Python's optional typing allows a function to manipulate values whose actual type is only discovered at runtime. Translating such code to Java requires committing to a concrete type at every step, and a poor choice at this stage can degrade performance disproportionately, well beyond what would be expected from a translation. Finding the right type for a given variable sometimes requires digging through several layers of class hierarchies in external libraries such as ANTLR, which can become a significant part of the porting effort.

Aliasing rules differ between the two languages. Python and Java do not handle object aliasing in the same way, and a literal translation can require inserting explicit clones in the Java version to preserve the original semantics. When these clones occur on a hot path executed a lot of times (in particular in the matching process), the cost becomes prohibitive and alternative strategies are needed. Cloning patterns should therefore be examined carefully during the port rather than translated literally.

Chapter 10

Future Work

This work opens two main directions for improving Pyttern. Both concern end-to-end execution time, but address different parts of the system: the startup cost of the runtime environment and the cost of parsing Python source code.

10.1 JVM Startup Optimisation

Depending on the use case, the JVM cold-start can account for a significant part of Pyttern's total runtime, especially when the matching phase completes quickly. Future work could therefore investigate JVM deployment techniques that reduce startup overhead, such as class-data sharing, ahead-of-time class loading, or native-image compilation.

This direction is orthogonal and could be applied without touching the implementation. Any such optimisation should nevertheless be evaluated on larger inputs as well, to ensure that gains observed on small files do not vanish once execution is dominated by processing time rather than startup cost.

10.2 Replacing the Parser

The parser remains the largest contributor to the optimised pipeline's total execution time, which makes investigating an alternative the most promising direction for further performance improvements. Tree-sitter would be one credible candidate, but such a change would not be a drop-in replacement. The grammar would have to be ported to the new parser's format, and the visitor-based implementation that walks the parse tree would need to be rewritten against its API. Depending on the candidate, additional dependencies may also be introduced, which could complicate other deployment strategies. Whether this trade-off is worthwhile, it would need to be assessed empirically, by comparing parsing performance and the engineering effort required against the gain on end-to-end matching time.

Bibliography

[1] Julien Liénard. Pyttern. https://github.com/JulienLie/Pyttern, 2026. Accessed: 2026-03-01.

[2] Julien Liénard, Kim Mens, and Siegfried Nijssen. The Pyttern program query language. In Jonathan Edwards, Roly Perera, and Tomas Petricek, editors, *Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025)*, volume 134 of *Open Access Series in Informatics (OASIcs)*, pages 23:1–23:15, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[3] OpenJDK. JEP 483: Ahead-of-time class loading and linking. https://openjdk.org/jeps/483, 2023. Accessed: 2026-03-08.

[4] OpenJDK. JEP 515: Ahead-of-time method profiling. https://openjdk.org/jeps/515, 2025. Accessed: 2026-03-08.

[5] OpenJDK. JMH: Java microbenchmark harness. https://github.com/openjdk/jmh, 2026. Accessed: 2026-04-14.

[6] Oracle. Java HotSpot virtual machine performance enhancements. https://docs.oracle.com/en/java/javase/11/vm/java-hotspot-virtual-machine-performance-enhancements.html, 2018. Java SE 11 documentation. Accessed: 2026-03-08.

[7] David Peter. hyperfine: A command-line benchmarking tool. https://github.com/sharkdp/hyperfine, 2026. Accessed: 2026-03-08.

[8] Python Software Foundation. CPython internal documentation: The bytecode interpreter. https://github.com/python/cpython/blob/main/InternalDocs/interpreter.md, 2026. Accessed: 2026-03-08.

[9] Python Software Foundation. pyperf: Toolkit to write, run and analyze benchmarks. https://github.com/psf/pyperf, 2026. Accessed: 2026-04-14.

[10] Python Software Foundation. pyperformance: The Python performance benchmark suite. https://github.com/python/pyperformance, 2026. Accessed: 2026-04-14.

[11] Python Software Foundation. What's new in Python 3.14. https://docs.python.org/3/whatsnew/3.14.html, 2026. Accessed: 2026-04-14.

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