The Pyttern Program Query Language

Julien Liénard
ICTEAM/INGI, UCLouvain, Louvain-la-Neuve, Belgium

Kim Mens
ICTEAM/INGI, UCLouvain, Louvain-la-Neuve, Belgium

Siegfried Nijssen
DTAI, KU Leuven, Belgium and ICTEAM/INGI, UCLouvain, Louvain-la-Neuve, Belgium

Abstract

Despite the availability of numerous tools and languages for detecting structural patterns in programs, their complexity often presents a steep learning curve. This highlights the need for a program query language that is easier to learn, use, and read while remaining sufficiently expressive for defining and detecting relevant structural coding patterns in program code. To address this challenge, we present Pyttern, a query language that extends Python syntax with regular-expression-inspired wildcards, enabling intuitive pattern-based querying of Python code. Its implementation relies upon a custom pushdown automaton describing how to match patterns over program parse trees, thus providing a robust foundation for structural code analysis. We evaluate Pyttern's usability and effectiveness through a study involving 35 master's students, who were asked to write seven different patterns to identify known programming misconceptions. The results demonstrate that Pyttern is both easy to learn and practical to use, at least for analysing small-scale programs.

2012 ACM Subject Classification Software and its engineering → Software verification; Software and its engineering → Automated static analysis; Software and its engineering → Specialized application languages; Software and its engineering → Software maintenance tools; Information systems → Query languages; Theory of computation → Grammars and context-free languages; Theory of computation → Automata extensions

Keywords and phrases Pyttern, Program Query Languages, Python, Pattern Matching, Parse Tree, Pushdown Automaton, Static Code Analysis, Wildcards, Tree Pattern Matching

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 Introduction

Understanding and analysing the structure of source code is a fundamental task in software engineering, particularly for code comprehension, quality assessment, and code improvement. While expert developers may leverage sophisticated static analysis tools, non-expert software engineers often lack skills to express and execute structured source code queries in such tools.

This paper focuses on *Pyttern*, a program query language specifically designed to facilitate pattern-based querying of Python code. Unlike many existing program query languages and tools, where queries often correspond to parse tree visitors, Pyttern is a language for expressing structural patterns in Python programs using a syntax that closely resembles that of Python. The language extends Python syntax with expressive, regular-expression-inspired wildcards, enabling intuitive queries that are easy to write and read. By maintaining syntactic similarity with Python, Pyttern minimises the learning curve, making it more accessible to junior developers and non-expert software engineers.

Pyttern is an expressive pattern language, which raises the important question of how to design an efficient algorithm for matching its expressive patterns to program code. Key design criteria for this matching algorithm include the following: (1) it should be sufficiently generic to facilitate adaptation to programming languages other than Python; (2) it should

support an easy addition of additional constructs to the pattern language, such as *named wildcards*; and (3) the algorithm should be sufficiently efficient in practice.

Our initial language prototype [10] did not satisfy these requirements. In this paper, we introduce an extended version that does. It uses a generic parser to construct a parse tree from the program code, thus making the approach applicable to any language with an existing parser. We then define a new matching algorithm that operates on these parse trees. This algorithm is based on a novel pushdown automaton for trees, which describes how the matcher traverses the parse tree of the program being matched. This automaton allows for a generic, efficient and maintainable implementation of the matching algorithm.

This paper has two main objectives: (1) to provide a detailed account of the implementation principles and theoretical foundations underlying a matching algorithm suitable for a language like Pyttern, and (2) to assess the usability of Pyttern through a new validation study involving junior developers. To achieve these objectives, we describe the language syntax, the design decisions that guided its implementation and then discuss its underlying implementation architecture and formalism. We then conduct an empirical evaluation in which 45 master's students used Pyttern to express queries aimed at detecting seven known programming misconceptions.

The remainder of this paper is structured as follows: Section 2 provides background on related program query languages. Section 3 presents and illustrates the Pyttern language. Section 4 discusses Pyttern's implementation architecture and underlying formalism. Section 5 presents our empirical validation and results and discusses the implications of our findings. Section 6 summarizes our contributions and outlines directions for future work.

2 Related Work

In this section, before presenting our Pyttern language, we present some related languages that served as a source of inspiration for Pyttern's language design and implementation.

First of all, code search engines and languages like FaCoY [8], Lancer [19], and SLAMPA [20] use code snippets as their query format and attempt to find matching or relevant code based on the given snippet. This philosophy that "code queries code" makes these search tools simpler and more developer-friendly than static analysis tools that require more structured queries. Such snippet-based methods allow natural code input instead of requiring developers to learn a complex query language, thus focussing on developer convenience, rather than deep program analysis with more exhaustive correctness guarantees.

Code with holes takes a more structured approach by supporting explicit "holes" (placeholders) in queries. Instead of relying on a search engine to determine which code fragments match a given code snippet, this approach represents partial programs using variables or wildcards to specify unknown parts [14, 11, 12]. Some approaches extend this idea further by introducing custom query languages that significantly modify or extend an existing programming language [6, 9].

This contrasts with more traditional program query languages (like CodeQL [3, 17, 1] or Datalog-based tools [15]) and static analysis techniques that rely on explicitly structured queries (e.g., "Find all methods with an integer parameter and a return type of float"), graph-based representations of code (like ASTs, control-flow graphs, or program dependence graphs), or even abstract interpretation or other formal methods for deeper semantic reasoning.

De Roover et al.'s [5] program query language tries to achieve the best of both worlds by mixing a powerful logic-based query language with the possibility of including concrete source code templates as part of the queries, and matching these queries against a combination of

structural and behavioural program representations, including call-graphs, points-to analysis results and abstract syntax trees. The idea is that the source code excerpts embedded in the logic queries act as prototypical samples of the structure and behaviour they intend to match. To use this approach, however, a developer is still required to be aware of the semantics of the underlying logic-based language and the various analysis techniques that it uses when matching queries, making it harder to use than pure snippet-based query languages.

The technique of regular expressions (RegEx) shares similarities with the concept of "code with holes" in that both allow for pattern-based matching with flexible or unknown components. Like code with holes which uses placeholders, regex wildcards (e.g., '.*', '\w+') can be used to represent unspecified code fragments, enabling more flexible searches. Regular expressions, however, operate on flat text, whereas code with holes is typically applied at the abstract syntax tree (AST) level, allowing for structure-aware and semantically meaningful queries. This makes code with holes more expressive and precise than regex-based approaches, as it respects the syntactic and semantic rules of programming languages rather than relying solely on textual patterns.

Codegex [18] and many other program query languages and analysis tools [4, 2] leverage regular expressions to query and analyse program code. Many *lint* tools [7], as well as other program analysis tools, operate by first parsing source code into a tree representation and subsequently *traversing* this tree to examine the program's structure.

Pyttern draws significant inspiration from SCRUPLE [14], a tool designed for querying C source code through high-level patterns. SCRUPLE introduces a symbolic representation that allows syntactic entities in C code to be substituted with placeholders ('holes'). The expressiveness of its pattern language stems from four key types of pattern symbols: wildcards representing individual syntactic entities, wildcards for groups of entities, named wildcards, and additional specialized features.

To perform pattern matching, SCRUPLE employs an extended nondeterministic finite state automaton referred to as the Code Pattern Automaton (CPA), which also operates on an input tree. SCRUPLE's algorithm provides inspiration for the matching algorithm implemented in Pyttern. However, the full theoretical basis underlying SCRUPLE has not been published by its author [14] nor in follow-up work. We found in our earlier work [10] that knowledge of the details of this theoretical basis is crucial in a practical implementation. For instance, when matching labeled wildcards, it is important to define precisely how the automaton would treat such labeled wildcards. With our paper, we aim to provide pattern matching on generic parse trees a firmer theoretical basis that allows for efficient, generic and maintainable implementations of languages such as Pyttern and SCRUPLE, while also allowing for a deeper future theoretical analysis.

3 Pyttern

Pyttern's goal is to provide a program query language that balances ease of use and expressiveness. It shares similarities with code snippet-based search engines, "code with holes" approaches, and other query languages that support concrete source code templates. Like these approaches, Pyttern aims to minimise the learning curve by staying close to the syntax of the host programming language. To make the language easy to learn, Pyttern's syntax is a mere extension of Python syntax, augmented with dedicated wildcards for pattern matching. Despite its simplicity, Pyttern remains expressive, offering powerful wildcards inspired by regular expressions. It performs matching at the level of parse trees, ensuring robust and flexible pattern recognition beyond mere syntactic similarity.

Wildcard	Meaning
?	Matches a single node in the parse tree.
?*	Matches zero or more nodes in a parse tree list node.
?{n, m}	Match between n and m nodes.
?name	Matches a single node and names it for later reference.
?[*Type*, ...]	Match a parse tree node of any of the given types.
?:	Matches the body of this wildcard at one nesting level.
?:*	Matches the body of this wildcard at any nesting level.

Table 1 Wildcards supported by the Pyttern program query language.

The list of wildcards supported by Pyttern, our dedicated Python pattern matching query language, can be found in Table 1. We blend those wildcards within the Python language making it easy to learn for any developer that knows a bit of Python.

An example of a typical coding idiom in an imperative language like Python is an accumulator, of which an example is given in Listing 1. In essence, an accumulator pattern refers to a technique where one first initializes an accumulator variable (line 2), and then iterates over a sequence (line 3), updating the accumulator variable with the result of some operation during the iteration (line 5). In the end, the accumulator variable that retains the cumulative result of all these operations is returned as result (line 6).

```
1 def count(lst, predicate):
2   tot = 0
3   for val in lst:
4     if predicate(val):
5     →tot += 1
6   return tot
```

Listing 1 Example of the accumulator coding idiom in Python.

```
1 def ?(?*):
2   ?accumulator = 0
3   ?:*
4     for ? in ?:
5       ?:*
6         ?accumulator += ?
7   return ?accumulator
```

Listing 2 The accumulator coding idiom expressed as pattern in Pyttern.

The structure of this coding idiom can be expressed straightforwardly as a Pyttern pattern, as illustrated in Listing 2. The Pyttern pattern follows the syntax of a Python program, but with wildcards acting as placeholders for matches. For example, the function name, loop variable and sequence, and the value added to the accumulator are abstracted by a single node wildcard `?` on lines 1, 4 and 6, respecitively. The accumulator variable is represented by a *named wildcard* `?accumulator` on lines 2, 6 and 7, which ensures that in multiple places the same variable name is matched. The function's parameter list on line 1 is abstracted by the `?*` wildcard matching a list of nodes. And finally the `?:*` wildcards on lines 3 and 5 provide the flexibility for the statements nested in the body of that wildcard to appear at any nesting level.

4 Implementation

We now provide a more detailed explanation of the current implementation of Pyttern.

Overall Architecture.

As previously mentioned, Pyttern's language design and pattern matching strategy are inspired by those of SCRUPLE. Pyttern's overall architecture, depicted in Figure 1, consists of four main components: two **parsers** (one for the Pyttern pattern and one for the Python program being matched), a **custom push-down automaton** (PDA), and a **matcher** which uses the PDA to visit the Python parse tree looking for matches, and returns a match set. Intuitively, the PDA describes the set of transitions the matching algorithm needs to follow to walk through a Python program in order for it to match the Pyttern pattern described by that PDA. Each match in the final match set is a valid set of transitions that leads to a match between the Pyttern pattern and the Python code being matched.

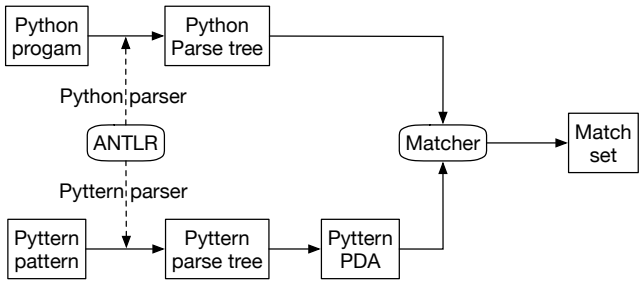

Figure 1 Overall architecture of Pyttern pattern matching.

Parsers.

To facilitate reasoning about nested programming constructs, Pyttern employs tree matching. This requires Python programs to be parsed into a parse tree. To do so, we chose ANTLR due to its maturity and the availability of an already defined Python 3 grammar.[1] ANTLR uses an adaptative LL(*) parser [13] to generate a parse tree for a given Python program. Figure 2 shows the parse tree produced for the Python program in Listing 1.

For parsing Pyttern expressions, which are essentially Python programs with specific wildcards, we extended ANTLR's Python3 grammar by adding extra production rules for these wildcards. Pyttern expressions can then be transformed into a parse tree. It is these two parse trees, the Python and the Pyttern one, which will then need to be matched.

We decided to use ANTLR due to its widespread use, its ease of use, and the grammars available for it. This allows us to define a technique that can be reused for other languages. It should allow us to adapt Pyttern to other languages that are already defined by ANTLR or even to use it to create grammars for languages not yet supported by ANTLR.

Custom Push-Down Automaton.

The next step in our approach concerns the transformation of the Pyttern pattern, referred to as a *pyttern*, in an automaton that is used by our matching algorithm. An example of such an automaton is given in Figure 3. It is produced by a visitor that walks through the pyttern parse tree and generates the PDA depending on the wildcards it encounters. Due to space limitations, we cannot describe here in full detail how each node in the pyttern parse

[1] `https://github.com/antlr/grammars-v4/tree/master/python/python3`

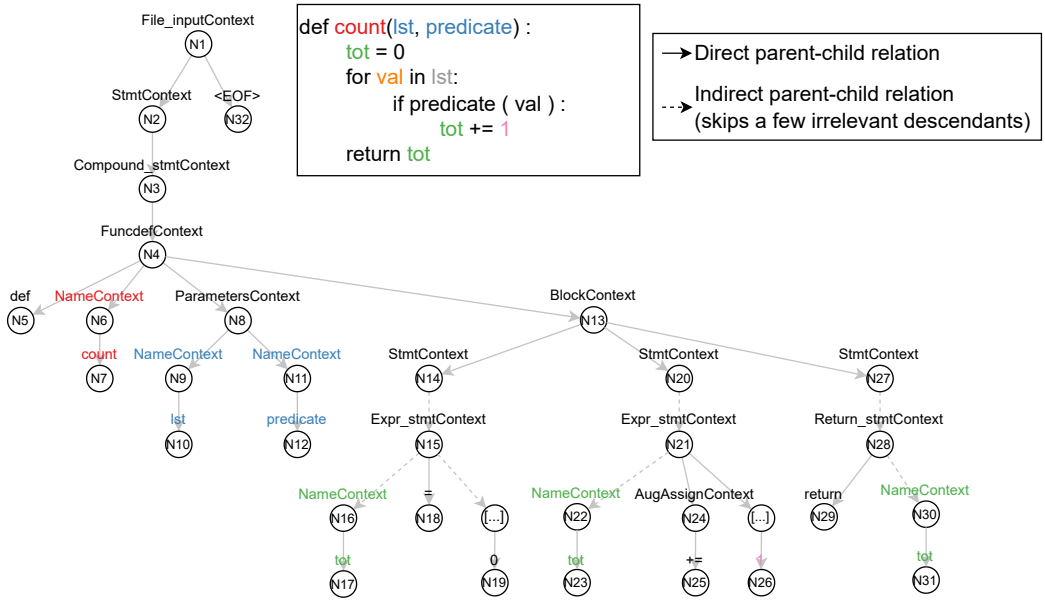

Figure 2 Parse tree generated for the Python code in Listing 1.

tree is transformed into PDA transitions. Nevertheless, we will explain these transformations intuitively through the example that we will use in the following paragraphs.

Formalism

Traditional Push-down Automata (**PDA**) are capable of recognizing context-free grammars (**CFG**) in strings. However, in our approach the input of the matching algorithm is a parse tree, as this allows the use of standard parsers to create such trees. Hence, we already have information with respect to the nested nature of the code, which could allow to find matches of patterns more efficiently. To build our matcher, we therefore propose a **PDA** that operates directly on the parse tree of Python source code.

Our custom pushdown automaton can be described as a 6-tuple $(Q, \Sigma, T, \delta, q_0, F)$ where:

- Q is a finite set of states generated by the pyttern to **PDA** transformer;
- Σ is the alphabet of labels of the nodes in the trees we will parse;
- T is an additional alphabet representing pyttern labeled wildcards ($\Sigma \cap T = \emptyset$);
- δ is the transition relation, described below, indicating the program of the automaton;
- $q_0 \in Q$ is the start state;
- $q_f \in Q$ is the accepting state.

Each transition (instruction in the program) is a 6-tuple $(q, a, A, t, q', \alpha) \in \delta$ where:

- $q \in Q$ is the current state;
- $a \in \Sigma \cup T \cup \{\epsilon\}$, indicating a label or labeled wildcard condition on the current node in the input tree in order to execute the transition;
- $A \in \{\epsilon, B, I\}$ indicates a condition on a symbol on the top of the stack of the automaton[2];
- $t \in \{\text{left-child}, \text{right-sibling}, \text{parent}\}$ is the *navigation direction*;

[2] We chose B to denote Body and I to denote Indentation. These labels are arbitrary and primarily reflect the parse tree structure, rather than the complete syntactic structure of the code.

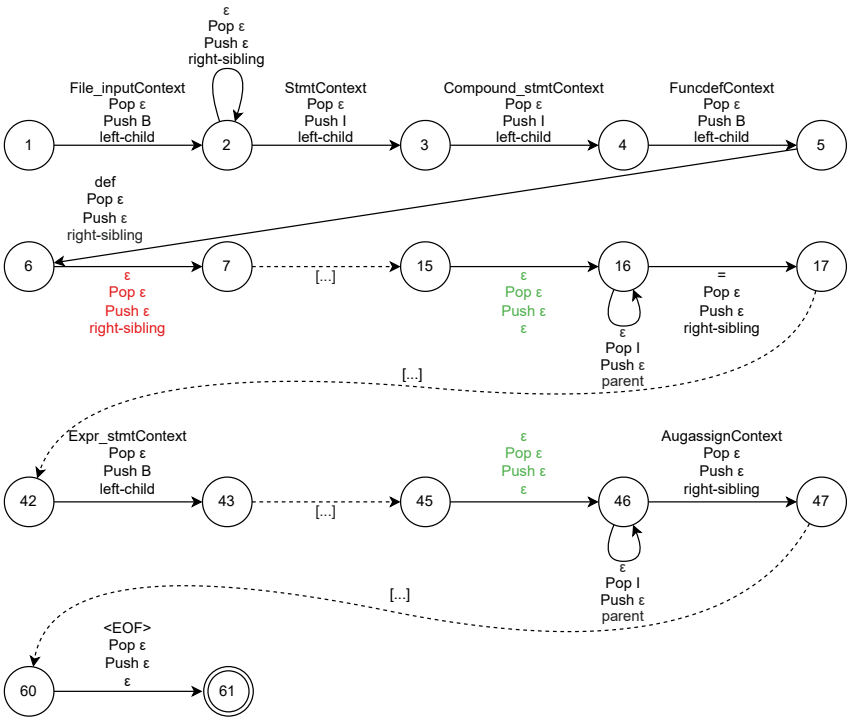

Figure 3 Custom PDA generated by the pyttern of Listing 2.

- $q' \in Q$ is the next state where the automaton should move to;
- $\alpha \in \{\epsilon, B, I\}$ is the stack symbol to push to the stack after the transition.

The **PDA** uses the transitions to move from one *configuration* to another. A *configuration* in our **PDA** is defined by a 4-tuple (q, v, s, m), where:

- $q \in Q$ is a state in the **PDA**;
- $v \in V$ is the node that we are currently considering in the Python parse tree, where V is the set of nodes of the parse tree;
- $s \in \{B, I\}^*$ is a string of symbols currently on the stack[3];
- $m : T \to V \cup \{\epsilon\}$ is a function indicating a mapping from named wildcards to a node in the Python parse tree.

Please note that the amount of information stored in one configuration is limited, making searching through a space of configurations efficient.

The initial configuration of the automaton is $(q_0, v_0, \epsilon, \{\tau \mapsto \epsilon | \tau \in T\})$, where v_0 is the root of the Python parse tree. In other words, the automaton starts in its starting state on the root of the input tree, with an empty stack and an empty mapping function. We can transition from one configuration to another, denoted by $(q, v, s, m) \vdash (q', v', s', m')$, if there is a $(q, a, A, t, q', \alpha) \in \delta$ in the program of our **PDA** that allows this. The requirements specified in the transition need to hold as follows:

- depending on the label condition a specified in the transition, one of the following holds:

[3] In the stack, B will be used to denote a node with multiple children (thus, when coming back up with parent, we will stop at this node) and I will be used for node with single child (allowing us to go back up until B).

- $a = \epsilon$: the transition does not impose a condition on the input label $label(v)$ in order to be executed;
- $a \in \Sigma$: the transition imposes a condition on the input label $label(v)$. If $label(v) = a$, where $label(v)$ is the label of node v in the input tree, the label condition is satisfied; intuitively, if the transition specifies an input label in Σ, we must find that input label at the current node in the input in order to move to the state q';
- $a \in T$: the transition imposes a labeled wildcard condition. If $m(a) \neq \epsilon$ (i.e., we have already matched the labeled wildcard previously), and $subtree(v)$ is isomorphic to $subtree(m(a))$, the labeled wildcard condition is satisfied; intuitively, if the transition specifies a wildcard in T, the structure and the labels of the subtree below the current node v must be isomorphic to the subtree and labels below the previous node $m(a)$ that was matched for the same named wildcard;

- $A = \epsilon$ or $top(s) = A$, where $top(s)$ is the last symbol of the stack s; intuitively, if the transition specifies a stack symbol, we must find that stack symbol at the top of the stack;
- $v' = next(v, t)$ is the next node in the input tree to consider, defined as follows:

$$next(v,t) = \begin{cases} \text{the right sibling of } v & \text{if } t\text{=right-sibling;} \\ \text{the first child of } v & \text{if } t\text{=left-child;} \\ \text{the parent of } v & \text{if } t\text{=parent;} \end{cases} \tag{1}$$

- $s' = s\alpha$ if $A = \epsilon$, or $s' = pop(s)\alpha$ if $A \neq \epsilon$, where $pop(s)$ corresponds to s without the last symbol, and $s\alpha$ indicates the concatenation of s and α; in other words, we add the symbol in the transition at the top of the stack, and remove the top of the stack if we detect a required symbol at the top of the stack;
- $m' = m \cup \{a \mapsto v\}$ if $a \in T$ and $m(a) = \epsilon$; otherwise, $m' = m$. In other words, if the transition specifies a label in T, and that label in T is not mapped to a node in V yet, this mapping is stored.

If one of the conditions of a transition is not met, we cannot use that transition to move to another configuration. We define that the **PDA** *matches* the input tree if there is a sequence of transitions from the starting configuration $(q_0, v_0, \epsilon, \{\tau \mapsto \epsilon | \tau \in T\})$ to an end configuration (q_f, v_f, s, m), where q_f is the end state and v_f is the last node in the input tree.

Matcher

The matcher component of Figure 1 finally boils down to finding a sequence of transitions for this automaton. Let us illustrate a sequence of transitions that parses a pattern on a concrete example. In the example of Figure 2, we can match the tree using these transitions

from the starting configuration to an end-configuration using the PDA from Figure 3:

$$
\begin{aligned}
(0, N1, \epsilon, \{\}) &\vdash (1, N2, B, \{\}) && \text{(Transition 1)} && (2)\\
&\vdash (2, N2, BI, \{\}) && \text{(Transition 2)} && (3)\\
&\vdash \dots && && (4)\\
&\vdash (15, N16, BI \dots IBI, \{\}) && \text{(Transition 15)} && (5)\\
&\vdash (16, N16, BI \dots IBI, \{\text{accu} \mapsto N16\}) && \text{(Transition 16)} && (6)\\
&\vdash (16, N15, BI \dots IB, \{\text{accu} \mapsto N16\}) && \text{(Transition 17)} && (7)\\
&\vdash \dots && && (8)\\
&\vdash (45, N22, BI \dots IBI, \{\text{accu} \mapsto N16\}) && \text{(Transition n-1)} && (9)\\
&\vdash (46, N22, BI \dots IBI, \{\text{accu} \mapsto N16\}) && \text{(Transition n)} && (10)\\
&\vdash (46, N21, BI \dots IB, \{\text{accu} \mapsto N16\}) && \text{(Transition n+1)} && (11)\\
&\vdash \dots && && (12)\\
&\vdash (60, N32, B, \{\text{accu} \mapsto N16\}) && \text{(Transition f-1)} && (13)\\
&\vdash (61, N32, B, \{\text{accu} \mapsto N16\}) && \text{(Transition f)} && (14)
\end{aligned}
$$

In these transitions, we specifically illustrate how our mapping for named wildcards operates. As shown in transitions $15 \rightarrow 17$, when a named wildcard is encountered and is not yet present in the mapping m, it is added to m. Conversely, transitions $n-1$, n, and $n+1$ demonstrate the alternative scenario: when the named wildcard is already in m, we verify that the structure and labels of the subtree below the current node v are isomorphic to those of the subtree below the previously matched node $m(a)$ for that named wildcard.

Our **PDA** is non-deterministic; for finding such a sequence of transitions we implemented a depth-first backtracking search that branches when multiple transitions are possible. While parsing **CFG** with PushDown Automata can generally be done in $O(n^3)$, adding the named wildcards and the mapping can make our search space explode. This makes our custom **PDA** non-polynomial when using named wildcards and each named wildcard used would make this complexity grow even bigger. We already use some optimisations like dynamic programming in our matching algorithm but we will not go into all details here.[4]

5 Validation

To validate our Pyttern language, we asked 35 CS master students to write Pyttern patterns, or pytterns in short, to detect symptoms of 7 distinct misconceptions in a dataset of potential answers to a first year bachelor course. Table 2 lists the different misconceptions for which we asked them to write a pyttern.

To obtain a controlled test set, we generated 507 code samples containing symptoms of specific misconceptions for the first year bachelor course, using the technique described by Steveny et al. [16], which defines a set of transformations to be applied to a base code set. By iteratively applying these transformations, we produced a series of code samples which contain the various misconceptions to be detected. We also included some dummy transformations that did not introduce any misconceptions, ensuring our test set to contain a mix of both relevant and irrelevant variations to the base code set. With this technique, eventually we

[4] If you want to learn more implementation details, you can check the project on Github: `https://github.com/JulienLie/Pyttern`.

Misconception	Description
NoEmptyInit	A class containing an `__init__` method with `pass` as only statement.
InitReturnsObject	A class `C` where the `__init__` method returns an object of the class itself, i.e. `return C()`.
EraseArg	When a function parameter gets reassigned to a different value, often a constant, within the function's body.
LoopVarUnused	A program containing a loop, with a loop variable that is not used inside this loop.
IncrForLoop	A program containing a for-loop, where the loop variable is incremented manually.
EarlyReturn	A program contain a `return` statement in the body of a loop, causing the loop to terminate prematurely.
PrintReturn	A function that has no `return` statement at the end but just a `print` statement.

Table 2 List of programming misconceptions which we asked our study participants to detect with a Pyttern pattern. The description describes a misconception a novice programmer may have, i.e. a wrong belief about a certain programming construct or concept.

obtained a validation set containing 132 (26%) programs with no misconceptions, 249 (49%) with one misconception and 123 (25%) with two different misconceptions.

Before starting to implement their 'pytterns' we aalsosked our study participants about their background knowledge on some concepts related to programming and program analysis. We aggregated these results in Figure 4. From this graph, we can observe that our participants considered themselves quite knowledgeable in Python though not experts. They also have some knowledge on design patterns (from a software engineering course which they followed). However, they had little or no experience with using AST visitors or other program query languages to analyse programs for relevant structural patterns. We thus consider our study participants as non expert programmers with very limited background in program analysis.

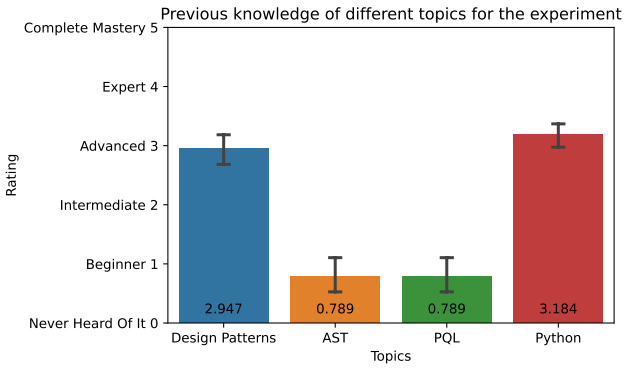

Figure 4 Background knowledge of the study participants

To understand how non-expert programmers adopt and utilise Pyttern, we analysed the queries they implemented. An example of a pyttern they wrote is shown in Listing 3. This code is a good solution for the InitReturnsObject misconception found in Listing 4. We compared their pytterns against a manually validated set of reference pytterns. This evaluation intended to determine how well our particiapnts could write Pyttern patterns, comprehend the language's syntax, and use the different wildcards provided by the language.

```
class ?name:
    def __init__(self, ?*):
        ?*
        return ?name(?*)
```

Listing 3 Example of a Pyttern written by a participant.

```
class Person:
    def __init__(self, name,
        surname):
        self.name = name
        self.surname = surname
        return Person(name,
            surname)
```

Listing 4 Example of a code with the InitReturnsObject misconception.

We analysed their implementations to understand some of the challenges they faced and to get an idea of the learning curve of the language.

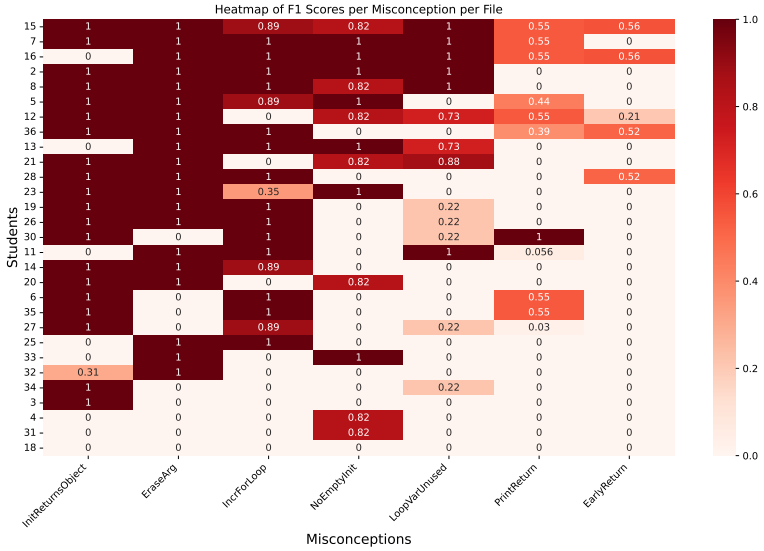

Figure 5 Heatmap of the students F1 score for each misconception.

From the 35 students who attempted to use Pyttern, (only) 29 were able to write at least one valid pyttern, and 28 successfully matched at least one file (containing a misconception to be detected). We ran each student's pytterns on our validation set and compared their results with our ground truth. Figure 5 presents a heatmap of F1 scores for the detection rate of each misconception across all files, allowing us to quickly identify which misconceptions students detected most effectively and which posed greater challenges.

From Figure 5 we can observe that certain misconceptions were consistently well-identified across most files, whereas others show lower F1 scores and appear more challenging to capture. In particular, some misconceptions display strong clustering of high scores, indicating that many students managed to write these patterns satisfactorily. These corresponded to patterns that could be defined with a simple Pyttern expression. Not surprisingly, the ones with lower score corresponded to patterns that required more complex pytterns or pytterns with more complex structures.

Meanwhile, Figure 6 displays a stacked bar plot of F1 scores per participant, allowing us to assess how each participant performed across the seven misconceptions to be detected. There seems to be quite some variation in the ability of participants to use Pyttern to write patterns for detecting misconceptions. Whereas some participants (on the left) seem successful at

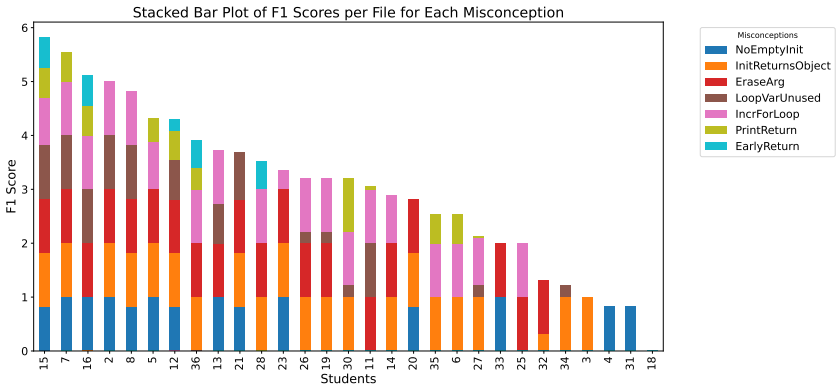

Figure 6 Stacked bar plot of student F1 scores for each misconception.

writing patterns for most misconceptions, others (on the right) only manage a few. We can also observe from this figure again that for some misconceptions most participants managed to write valid pytterns, whereas for other misconceptions this was much less the case.

Nevertheless, our results do show that the majority of study participants successfully developed valid Pytterns, in spite of their limited prior knowledge about program query languages. Their previous knowledge of Python helped them to understanding and effectively utilise Pyttern, at least for simpler patterns, suggesting that the language design facilitates a gentle learning curve. This result seems to suggest that Pyttern can be adopted by users with basic Python skills without too much difficulty. The limited scope and duration of the experiment unfortunately did not allow us assess whether, with more time and experience, the participants would have been able to write more advanced pytterns successfully.

Although the misconceptions for which we asked our participants to write Pyttern checkers were presented in a certain order, they had the freedom to implement them in any order, within a given time period (4 hours). Nevertheless, we checked whether the order in which the misconception were presented affected the mean F1 score for this misconception. This analysis is summarised in Figure 7. We performed a one-tailed Pearson correlation test between the order of the question and the mean F1 score. Our hypothesis was that later questions yield a lower F1 score (in other words that students would focus first on the misconceptions that were presented first). This Pearson test gave us an r of -0.648 and a $p\text{-}value$ of 0.0577. This means that we probably have a moderate-to-strong negative relationship between the order of the question and the mean F1 score, even though we cannot claim that the correlation is there with an α of 0.05.

Obviously there are still many threats to validity of this validation of our new implementation of Pyttern. First, the participant sample used for this evaluation was limited and may not accurately represent the intended population of potential users nor their level of expertise that Pyttern is designed for. Additionally, the version of Pyttern used by our participants was not entirely bugfree during the evaluation. This could have led to students writing buggy pytterns and missing matches. (It could also explain why 6 participants were not able to write any valid pyttern.) Some (few) participants also had previous experience with Pytterns, as they participated in an earlier experiment with the first prototype of the Pyttern language a year earlier. This may have biased their performance in the positive sense, as the learning curve would have been lower for them. Finally, the set of misconceptions that we selected for this validation might not fully capture the range of possible patterns that we could define with Pyttern. It could also be not representative for the set of all possible misconceptions

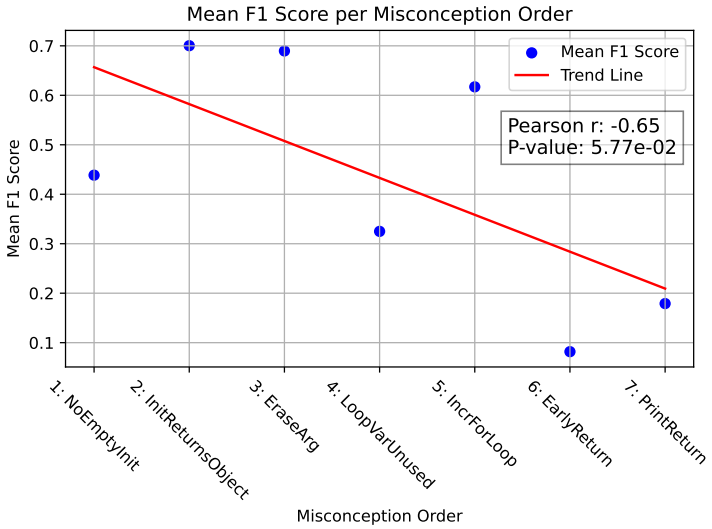

Figure 7 Correlation between the order of the misconceptions asked and the mean F1 score

that we could find in small programs.

6 Conclusion and Future Work

In this paper, we introduced a new formalism for Pyttern, a novel approach to pattern matching in Python source code using a modified nondeterministic pushdown automaton. We describe the automaton in detail, along with the foundational principles of this new version of Pyttern, which we believe establishes a robust basis for further improvement. Our formalism presents a general technique for parse tree pattern matching that could be adapted for other programming languages. Pyttern is an implementation of this formalism in Python to investigate whether defining a query language closely aligned with the host language can assist non-expert developers in writing concise patterns.

Our validation study with master students suggests that the learning curve for Pyttern is generally good, though there is room for improvement. Participants with little or no experience in program query languages were able to implement simple pytterns using their previous Python knowledge. However, the creation of more complex patterns proved challenging, highlighting areas where further improvements in the language's design could enhance usability. Overall, the successful implementation of different patterns and the constructive feedback received indicate a promising path forward for Pyttern.

Future work will focus on refining the ease of expressiveness of Pyttern, in particular its ability to compose patterns in terms of more simple patterns and combining patterns with logical operators. Also, for now, Pyttern is totally based on Python. We plan to extend the idea to other programming languages and transform the core features of Pyttern (i.e. the PDA and the Matcher) to a pattern matching framework for any language. In fact, we already started this endeavour as we are currently implementing a Java version of Pyttern.

References

1 Codeql website. URL: `https://codeql.github.com/`.
2 Spotbugs website, 2006. URL: `https://spotbugs.github.io/`.

3 Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2016.

4 Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sep. 2008. doi:10.1109/MS.2008.130.

5 Coen De Roover, Theo D'Hondt, Johan Brichau, Carlos Noguera, and Laurence Duchien. Behavioral similarity matching using concrete source code templates in logic queries. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 92–101, 2007.

6 Katsuro Inoue, Yuya Miyamoto, Daniel M German, and Takashi Ishio. Code clone matching: A practical and effective approach to find code snippets. *arXiv preprint arXiv:2003.05615*, 2020.

7 Stephen C Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

8 Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, pages 946–957, 2018.

9 Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 601–614, 2018.

10 Julien Liénard, Kim Mens, and Siegfried Nijssen. Pyttern: a python-based program query language. In *Belgium-Netherlands Software Evolution Workshop (BENEVOL)*, 2024.

11 Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 997–1016, 2012.

12 Rohan Mukherjee, Swarat Chaudhuri, and Chris Jermaine. Searching a database of source codes using contextualized code search. *arXiv preprint arXiv:2001.03277*, 2020.

13 Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10):579–598, 2014.

14 Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

15 Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC '16, page 196–206, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892208.2892226.

16 Guillaume Steveny, Julien Liénard, Kim Mens, and Siegfried Nijssen. Feedback with bert: When detecting students' misconceptions becomes automatic. In *Responsible Knowledge Discovery in Education (RKDE)*, 2024.

17 Tamás Szabó. Incrementalizing production codeql analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1716–1726, 2023.

18 Xiaowen Zhang, Ying Zhou, and Shin Hwei Tan. Efficient pattern-based static analysis approach via regular-expression rules. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 132–143, March 2023. doi:10.1109/SANER56733.2023.00022.

19 Shufan Zhou, Beijun Shen, and Hao Zhong. Lancer: Your code tell me what you need. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1202–1205. IEEE, 2019.

20 Shufan Zhou, Hao Zhong, and Beijun Shen. Slampa: Recommending code snippets with statistical language model. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 79–88. IEEE, 2018.