

An Exploratory Study of Ad Hoc Parsers in Python

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Softwareengineering and Internet Computing

eingereicht von

Andreas Olschnögger, BSc

Matrikelnummer 11702809

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assoc. Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Mitwirkung: Dipl.-Ing Michael Schröder, BSc

Wien, 2. Mai 2025

Andreas Olschnögger

Jürgen Cito

An Exploratory Study of Ad Hoc Parsers in Python

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Softwareengineering and Internet Computing

by

Andreas Olschnögger, BSc

Registration Number 11702809

to the Faculty of Informatics

at the TU Wien

Advisor: Assoc. Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Assistance: Dipl.-Ing Michael Schröder, BSc

Vienna, May 2, 2025

Andreas Olschnögger

Jürgen Cito

Erklärung zur Verfassung der Arbeit

Andreas Olschnögger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Mai 2025

Andreas Olschnögger

Danksagung

Im Rahmen meiner Arbeit wurde ich von vielen Personen unterstützt, bei denen ich mich herzlich bedanken möchte. Seitens der TU Wien möchte ich mich bei meinem Betreuer Assoc. Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc, sowie bei Dipl.-Ing. Michael Schröder, BSc, der mich mit seiner Expertise beraten hat, bedanken.

Zudem möchte ich mich bei Asst. Prof. Robert Dyer, Ph.D., bedanken, der mich bei Fragen bezüglich Boa unterstützt hat.

Ein weiterer Dank gilt Carolin, meiner Familie und allen Personen, die mich während des Studiums unterstützt haben.

Acknowledgements

During my thesis, I was supported by many people, whom I would like to thank. My thesis was guided by Assoc. Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc, and mentored by Dipl.-Ing. Michael Schröder, BSc, who assisted and advised me throughout the study.

I would also like to express my gratitude to Assistant Professor Robert Dyer, Ph.D., for his support with my questions regarding Boa.

Finally, I want to thank Carolin, my family, and all others who supported me during my studies.

Kurzfassung

Ad hoc Parsing bezeichnet das Verarbeiten von Strings, ohne dass formale Parsing-Regeln verfolgt werden. Ad hoc Parser treten als Codeabschnitte in vielen Stellen im Code auf und werden oft spontan geschrieben. Esentielle Bestandteile sind String-Manipulationsfunktionen wie `split` oder `replace`. Diese Arbeit ist eine explorative Studie in Python, welche das Vorkommen, Charakteristiken und Implementierungsmuster von ad hoc Parsern untersucht.

Wir entwickelten eine Methode um mit der Nutzung des Frameworks Boa aus einem Datenset mit 1,710 Python Projekten auf GitHub string Variablen zu erkennen und ausgehend von diesen Variablen ad hoc Parser Codestücke zu generieren. So war es uns möglich ein Datenset bereitzustellen, dass 34,925 ad hoc Parser beinhaltet.

Unsere Auswertung zeigt, dass in 75% aller Projekte ad hoc Parsing verwendet wird. Unsere wichtigsten Ergebnisse sind: 1. Ad hoc Parser sind typischerweise kompakt (Mediengröße von 4 Zeilen) und treten über die ganze Methode verteilt auf; 2. Funktionen zur Bearbeitung von strings wie `split` und `replace`, sowie Typkonvertierungen sind vorherrschend; 3. 11% der ad hoc Parser verwenden reguläre Ausdrücke. Diese werden hauptsächlich im hinteren Teil des Parsers zur abschließenden Verarbeitung verwendet; 4. Fehlerbehandlungen werden in ad hoc Parsern kaum behandelt. 80% der Parser, die potentiell Fehler werfen könnten, sind ohne Fehlerbehandlung; und 5. Schleifen in ad hoc Parsern weisen überwiegend eine flache Verschachtelung und lineare Grenzen auf.

Diese Arbeit stellt einen umfangreichen Datensatz an ad hoc Parsern in Python zur Verfügung und bietet Einblicke in ihre syntaktischen Charakteristiken und Muster. Die Erkenntnisse der Arbeit dienen der Entwicklung statischer Analysewerkzeuge, Parsergeneratoren, und weitergehender Forschung in diesem Bereich.

Abstract

Ad hoc parsing is the processing of strings without following formal parsing rules. Ad hoc parsers occur as code snippets in various places of source code and are typically written on the fly. Characteristic features of ad hoc parsers are string manipulating functions like `split` or `replace`. This thesis is an exploratory study investigating prevalence, characteristics, and implementation patterns of ad hoc parsers in Python.

We create a method to extract ad hoc parser code snippets out of source code. Utilizing the Boa framework, we analyze a dataset of 1,710 Python projects on GitHub and generate a dataset containing 34,925 ad hoc parsers.

Our results show that 75% of all projects contained ad hoc parsing. The most important findings are: 1. Ad hoc parsers are typically compact, with a median line of code of 4, and are found everywhere in code. 2. String transforming functions like `split` and `replace`, as well as type conversion are dominant in ad hoc parsers. 3. 11% of ad hoc parsers use regular expressions. Regular expressions are predominantly located in the latter section of an ad hoc parser for final processing. 4. The majority of ad hoc parsers do not contain exception handling. 80% of ad hoc parsers, which potentially throw errors, have no local exception handling. 5. Ad hoc parsers show mainly a narrow nesting and linear bounds.

In this work, we present an extensive dataset containing ad hoc parsers in Python and provide insights into syntactic characteristics and patterns of ad hoc parsers. The results of this work serve the development of static analysis tools, parser generators, and future research in this field.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
2 Background and Related Work	3
2.1 Background	3
2.2 Related Work	7
3 Ad Hoc Parser Mining	9
3.1 Identification of String Variables	11
3.2 Parser Determination	14
4 Findings	23
4.1 Dataset Categorization and Analysis	23
4.2 Frequency	25
4.3 Location	26
4.4 Size	28
4.5 Input Sources	30
4.6 Function Calls	31
4.7 Regular Expressions	34
4.8 Loops	36
4.9 Error Handling	37
4.10 Threats to Validity	38
5 Conclusion and Future Work	41
List of Figures	43
List of Tables	45
List of Algorithms	47
	xv

CHAPTER 1

Introduction

Ad hoc parsers are code snippets that transform an input string in some form, without following common parsing rules or patterns. In Python, methods like `slice`, `trim`, etc., are often used for this purpose. This means that the programmer decides ad hoc how to transform a string. Ad hoc parsers are frequently used in applications, yet there has been little research on ad hoc parsers despite the risks they pose [5]. Many characteristics of ad hoc parsers, such as average size, used functions, or typical parsing patterns, are not known

This exploratory study aims to provide a dataset of ad hoc parser program snippets and to investigate and analyse the semantic and syntactic properties of these ad hoc parsers. To achieve this objective, we conduct a study based on a pre-registered research protocol [26] that allows a thorough and systematic methodical examination of the subject.

For collecting the dataset of parsing snippets, we mine and analyze source code using the Boa framework [9]. Boa provides several source code data sets for different programming languages. We focus on programs written in Python because it is an extremely popular and widely used language for data analysis and machine learning tasks.

To this end, we address the following research questions in this thesis:

RQ1 What are location characteristics of ad hoc parsers in Python projects?

Ad hoc parsers may be found in many locations within the source code. By analyzing the projects that contain an ad hoc parser, it is possible to gain insight into their positioning and distribution. This gives us insight into whether ad hoc parsers are predominantly located at the beginning or end of functions. If they are distributed throughout the code or concentrated in specific sections. An important aspect of this investigation is the assessment of the average size of ad hoc parsers in terms of lines of code or expressions, as well as the usage of temporary variables and method chaining. This helps

us to understand how ad hoc parsers achieve syntactic compactness while encompassing intricate functionality.

RQ2 What are the syntactic characteristics of ad hoc parsers?

We want to analyse various syntactical characteristics of ad hoc parsers from beginning to end. Where does the input of an ad hoc parser originate from? Is it a global variable, a return value of a function call, or is the input read from a file? Further, we want to know how the input is processed. What functions or regular expressions are used in the parsing process, and does parsing include error handling? If there are looping constructs, are they functional (e.g., `map`, `split`) or are they more direct iterations (e.g., `for`, `while`)?

Background and Related Work

2.1 Background

2.1.1 Boa

For the generation of the parser code snippets, we use Boa. Boa provides a domain-specific language and infrastructure for analysing a wide range of software projects. Boa already offers several ready-made datasets for different programming languages. In this paper we focus on the dataset **2022 February/Python**, which consists of 102,424 projects from GitHub. The dataset includes only Python files.

Table 2.1: Summary Statistics of the Dataset **2022 February/Python**

	min	mean	median	max
files	1.0	232.84	30	145 711
AST size	4.0	37710.5	5 832	47 655 140
stars	24.0	242.94	59	138 438
authors	1.0	7.91	3	10 895
commits	1.0	314.69	46	203 889
earliest commit	2005	2016.5	2017	2021
latest commit	2007	2019.04	2020	2021
created	2008	2016.69	2017	2021
timespan in years	0	2.51	1.62	16.57

Table 2.1 lists the summary statistics about the dataset, which include information about the number of files, authors, stars in the GitHub repository, and more. It can be seen that the dataset contains a wide variety of projects. For example, the range of files per project goes from 1 to over 100,000. 50% of all projects have between 13 and 82 files.

```

136 get_used_strings := traversal(node: CFGNode): VARLIST {
137     vars: VARLIST;
138     vars := vars_used_as_string(node.expr);
139     foreach (i: int; def(node.successors[i]))
140         vars = union(vars, getvalue(node.successors[i]));
141     return vars;
142 };

...

160 is_string := traversal(node: CFGNode): bool {
161     str := false;
162     if(getvalue(node, is_declared_as_string)) str = true;
163     used_strings := getvalue(node, get_used_strings);
164     ...
170 }

61 cfg := getcfg(m);
62 traverse(cfg, TraversalDirection.FORWARD, TraversalKind.DFS,
63     get_used_strings);
63 traverse(cfg, TraversalDirection.FORWARD, TraversalKind.DFS,
64     is_string);

```

Figure 2.1: Boa snippet: Graph traversal to identify string variables.

The data set does not contain any projects with fewer than 24 stars. The median for the number of commits is 46.

Boa is its own domain-specific language, which means that the Python AST is compiled from the source projects into a Boa AST. The ASTs can be analysed using Boa queries. To analyse the Boa AST, Boa provides its own syntax inspired by object-oriented visitor patterns. Another very interesting feature of Boa for our work is the analysis of program graphs. Boa provides several functions to generate standard graphs, such as: control flow graph (CFG), control dependency graph (CDG), data dependency graph (DDG), and program dependency graph. Thus, a corresponding graph can be generated from each method. Each graph consists of nodes that represent a corresponding AST node. To work with these graphs and to analyze them, we can write graph traversals. Graph traversals are similar to functions. The only input parameter is the corresponding node. Traversals can optionally return a value. This value is associated with the corresponding node within the traversal and is accessible from other nodes. Figure 2.1 shows an example of code traversals. The traversal `get_used_strings` stores per node all string variables that are used from here or later.

2.1.2 Dependence Graphs

For constructing ad hoc parsers snippets, we use different dependence graphs. This section covers control flow graphs, data dependence graphs, and program dependence graphs, which are important structures in program analysis and used for capturing distinct aspects of program behavior, making them crucial for compiler optimizations, software verification, and static analysis tools.

Data Dependence Graph

The data dependence graph (DDG) [18] can be used to visualize data dependencies inside a program. A node in the DDG can represent, e.g., statements, tasks, functions, and the edges represent the dependence according to the flow of data [6]. This means a directed edge from node i to node j states that j is data dependent on statement i . Figure 2.2b shows an example of a DDG. The Statements L2, L3, L4, L6 dependent because of the variable s , while L6 is a data-dependent statement on L5 because of the variable *indent*.

Control Flow Graph

A control flow graph (CFG) [2] is a directed graph describing all potential paths of program code that is executed. Nodes represent basic blocks. Allen [2] describes a basic block as a linear sequence of consecutive program instructions containing a single entry and exit point. CFGs have been applied in many areas of research [28, 15, 16]. Edges describe the control flow. An edge from block $B1$ to block $B2$ exists only if there is a possible execution to proceed directly from the last statement of $B1$ to the first statement of $B2$. Figure 2.2d shows an example of a CFG.

Control Dependence Graph

The control dependence graph (CDG) is a directed graph modeling the conditional execution dependencies within a program. Contrary to the CFG, the CDG does not contain the sequential information and emphasises the relationship of control predicates (e.g., conditions and `if` or `for` statements) and the program parts they control. Nodes of the CDG correspond to executable elements of the program, like statements, basic blocks [10]. For constructing a CDG, the post-dominant relationships of the CFG have to be constructed. The post-dominant relationships represent the edges, like visible in Figure 2.2c.

Program Dependence Graph

The Program Dependence Graph (PDG) combines data dependence and control dependence in a single graph structure. Ferrante et al. [10] introduced the PDG as a way to represent the essential control and data relationships in a program executing without the often very strict sequencing imposed by the traditional CFG. Nodes in the PDG represent operators and operands, while edges represent either control dependencies or data

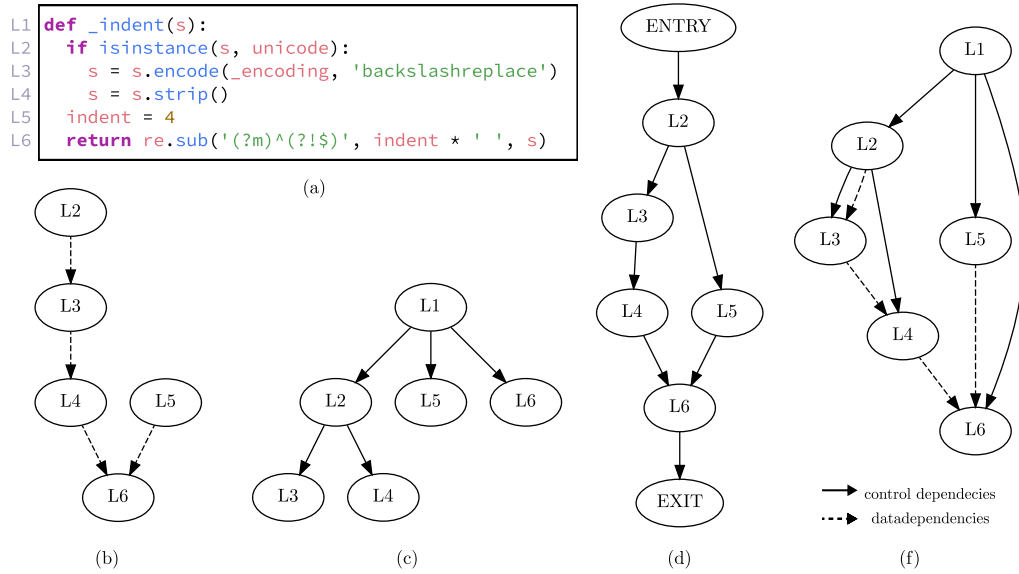


Figure 2.2: Example of dependency graphs. (a) code sample; (b) data dependence graph; (c) control dependence graph; (d) control flow graph; (e) program dependence graph; (f) program dependence graph with control and data dependencies.

dependencies. They capture the constraints on the execution order and computational relationship between the nodes (Figure 3.7f).

2.1.3 Program Slicing

In this study, we use program slicing to extract ad hoc parsing snippets from Python source code. Weiser [30] introduced program slicing as a powerful technique for extracting program snippets based on their data and control flow. The algorithm to construct a program slice requires two key pieces of information: the control flow graph and data flow information. The data flow information is defined by two sets for each node n :

- $DEF(n)$: a set of variables defined at node n
- $REF(n)$: a set of variables that are referenced at node n

The algorithm involves computing sets of relevant variables for each node in the CFG for a slicing criterion $C = \langle n_0, V_0 \rangle$. The set $R_C(n)$ represents all relevant variables and is computed as follows:

1. Initialization: The set of relevant variables for n_0 is initialized to V_0 . All other sets $R_C(n)$ are initialized to the empty set \emptyset .
2. Iteration: The algorithm iteratively computes for each node s the set of relevant variables based on its successors and its own DEF/REF sets. This is done until a fixed point is reached, meaning no more variables can be added to R_C .

Once the sets of relevant variables for all nodes are computed, the slice S is constructed. A statement node n is included in the slice if the set of relevant variables of $n+1$ intersects with the set of variables defined by $DEF(n)$ or n is a control predicate that determines whether statements in S are executed.

Weiser work established the foundation for program slicing and has been widely employed since [12, 19, 32, 31]. AlAbwaini et al. [1] introduced a new model for identifying dead code in programs using decomposition slicing [13].

2.1.4 Software Measures and Metrics

2.2 Related Work

Ad hoc parsers: There is little research in the area of ad hoc parsers. Schröder and Cito have already dealt with ad hoc parsers and propose an automatic grammar inference system for ad hoc parsers. The vision is to transform an ad hoc parser from a source language like Python into an intermediate representation, which represents the domain-specific language for parsing. By inference, a language model is created from the intermediate representation, from which an appropriate grammar can be generated [25].

Analyzing language features in Python: The first study to analyze language features and automatically identify their use in Python was conducted by Peng et al. [24]. By developing an automatic language feature recognizer, they analyzed 35 popular Python projects from 8 different domains and found that single inheritance, decorator, keyword argument, for loops, and nested classes are the top 5 used language features. By analyzing the different domains, they could give insights into the preference for certain language features per domain. Projects in DevOps, for example, use exception handling frequently. The results in exception handling are specifically interesting because exception handling also plays a crucial part in ad hoc parsing. Peng et al. found that developers care most about ImportError, ValueError, AttributeError, KeyError, and OSError, which account for 80% of all errors.

Large scale analysis with Boa: There are several studies that utilized Boa for static large-scale analysis. Dyer and Chauhan [8] utilized Boa to analyze predominant paradigms in Python. They analyzed about 101,000 projects and found that many files and projects favor the object-oriented paradigm, while single-file projects mostly favor procedural or mixed paradigms. The analyzed files rarely change their predominant paradigm over time. Apart from Python, Boa also provides datasets for Java projects. Asaduzzaman et al. [3] used this Java dataset to conduct an empirical study on how developers use exception handling. By analyzing a dataset holding more than 274,000 Java projects, they found that improper exception handling practices are not uncommon within Java applications. These improper exception handling practices are not affected by the experience of developers. Kery et al. [17] and Nakshatri et al. [23] also utilized Boa for analyzing exception handling in Java using a dataset containing nearly 8,000,000

GitHub repositories. Kery et al. found that developers would rather handle exceptions locally than propagate them by throwing an exception. Programmers tend to use actions like Log, Print, Return, or Throw in catch blocks. Bad practices in exception handling, like an empty catch block or catching exceptions, are widespread. Another large-scale analysis using Boa was performed by Flint et al. [11]. They investigated type inference in Kotlin. Therefore, they used the provided dataset from Boa, containing about 500,000 projects, and found that type inference is frequently used by developers when declaring local variables or methods that are defined outside of the file.

In a study from Yang et al., 3,000,000 Python files from 51,000 different projects on GitHub have been analyzed to address the questions of how complex Python features are used. Their findings showed that the usage of dynamic features that pose a threat to static analysis is infrequent, but the usage of context managers and decorators is widespread [33].

Static analysis in other languages Sihler et al. also performed a large-scale static investigation of real-world R code. They looked at more than 50,000,000 lines of code to analyze feature characteristics of R code. They found that commonly used features are assignments with `<-` and `=`, `for` loops, `if` conditionals, and name-based indexing operations with `$` [27]. Another large-scale analysis was done by Mariano et al.. They investigated syntactic and semantic features of loops found in Solidity smart contracts. Based on their Findings, they built a domain-specific language and a tool for automatically summarizing solidity loops, by using a combination of k-means clustering and manual sampling [20]. Gopstein et al. analyzed 14 popular and influential open source C and C++ projects to find ‘atoms of confusion’, which are extremely small code patterns that can cause misunderstanding, like the conditional operator. Their results show that 15 types of confusing micro patterns are found millions of times in programs like the Linux kernel and GCC. They found that there is a strong correlation between atoms of confusion and bug fixing commits. Projects with a lot of confusing micro patterns tend to have a higher rate of security vulnerabilities [14].

CHAPTER 3

Ad Hoc Parser Mining

For the collection and analysis of a large dataset of Python projects, we utilized Boa [9], a source code mining language and infrastructure. Using this framework has the added advantage of ensuring the reproducibility of our analysis, allowing it to be easily applied to other datasets. Additionally, Boa’s language-agnostic nature makes it relatively straightforward to adapt the analysis to other programming languages, especially when compared to creating custom analysis scripts. All Boa scripts for extracting the parsers and evaluation scripts are publicly available on GitHub¹. To extract ad hoc parsers from the dataset, we employ a form of program slicing [30] using the built-in static analysis capabilities of the Boa framework. The following approach is also visible in Figure 3.1.

1. All methods from all Python files in each project are converted into a PDG [10].
2. For each method, all string variables are identified, including the arguments. Since Python is typically untyped, a coarse but effective type inference is performed by consulting an extensive list of methods whose arguments or return values are un-/known to be strings. If type hints are available, they are also considered (Section 3.1).
3. For each string variable, we create a forward slice of the program starting from its first occurrence (unless it is already part of a previous slice). We utilize the PDG to construct the slice and continue it if the data dependencies are inputs for additional parsing operations. This ensures capturing the core of the parser, including intermediate results and transformations, without obtaining a slice that is the size of the entire method (Section 3.2).
4. If a program slice does not contain methods that impose constraints on the input string (e.g., if the string is simply repeatedly appended), it is discarded.

¹<https://github.com/schnoeggi/ad-hoc-parsers-in-python>

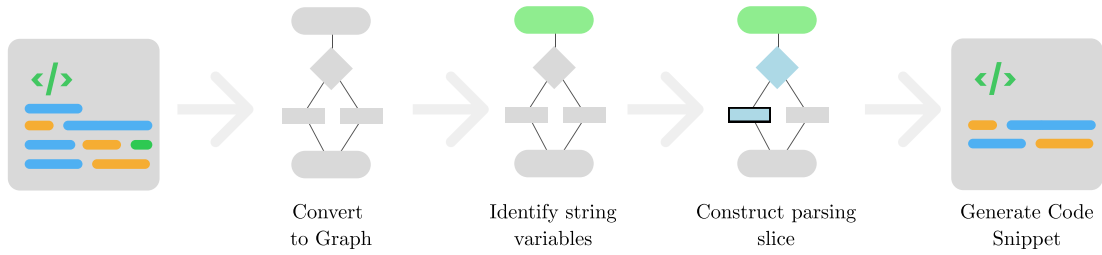


Figure 3.1: Workflow of Parser Slicing

Figure 3.2 shows an example of a Python method and its corresponding CFG. The input value of this ad hoc parser is the method argument `content_type`. It has been recognized as a string variable because the `split` method is used on `content_type` in line 3. The detection of string variables is explained more precisely in Section 3.1. Starting from the node of the input string, we explore the graph using depth-first search. During this process, we analyze each subsequent node for data dependencies on the previous node and check if it performs a parsing operation. In this manner, the parser (highlighted in blue in the example) is constructed. A detailed description of the parser creation process is provided in Section 3.2

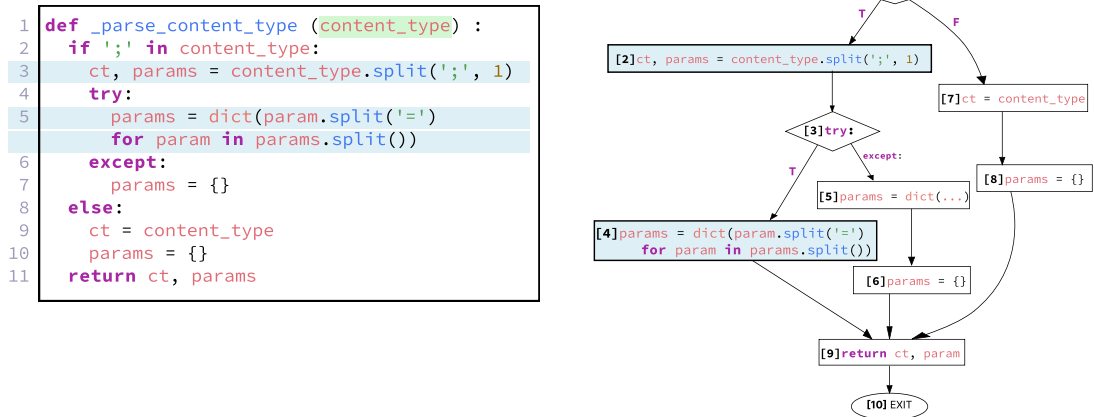


Figure 3.2: The green nodes visualize the source of the input. The blue nodes depict the set of parsing nodes and supporting nodes. The set of parsing nodes is visualized by the bigger stroke.

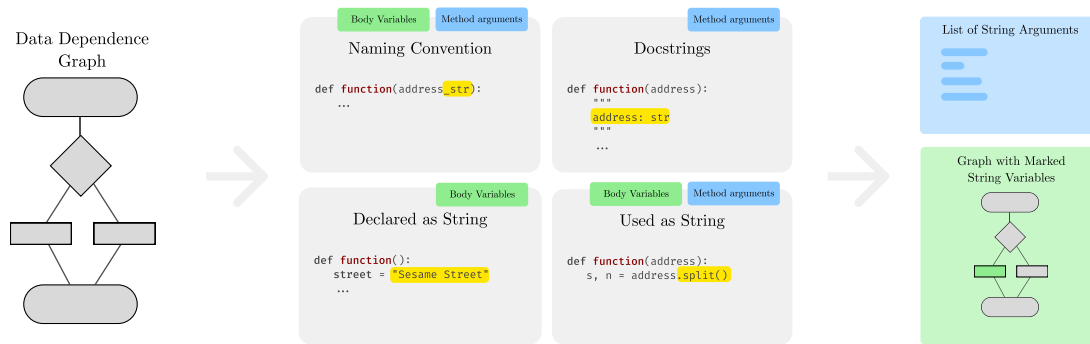


Figure 3.3: String determination: Each method argument and each node is analysed for the four properties. Some properties only apply to method arguments, and some only to body variables. The result is a list of string arguments and the DDG with marked nodes containing a string definition.

3.1 Identification of String Variables

Since Python is an untyped programming language, the initial step for creating parser snippets is to determine which variables represent strings. We employed various approaches, including analysis of naming conventions, comments, and examining methods or operations applied to the variable.

In this study, we distinguish between two primary types of variables that can be defined as strings: method parameters and variables created within the method body. We use the DDG constructed by Boa for this analysis. The identification of whether a variable is a string is explicitly evaluated through the use of Boa. To do this, the DDG of a method is traversed. Boa's traversal offers the option of saving return values for individual nodes. In this way, each node that contains a string declaration can be marked accordingly. As method arguments do not reflect individual nodes in the DDG, method arguments that are strings are stored separately in a list. Four different aspects are used to determine whether a variable is a string: naming convention, references in docstrings, if they have been declared as a string, or if they have been defined as a string. Each of these aspects is inspected separately. If one of them applies to the variable, it is recognized as a string. This process is visualised in Figure 3.3.

3.1.1 Naming convention

We utilize the variable name to infer its nature as a string. Through qualitative analysis of numerous diverse codebases, it has been observed that variables named with the prefixes or suffixes `name`, `text`, `str`, or `string` are typically categorized as strings. When these identifiers are used as prefixes or suffixes, they are separated by an underscore (`_`).

3.1.2 Docstrings

In order to extract the variable types of method arguments, the docstring of the method to be analyzed was examined. A specific regular expression was employed in order to encompass the various commonly used docstring types, including those of the Google Style and Numpydoc Style, as well as those that are individually documented. The type of method arguments is identified by examining the docstrings within the methods themselves. The following regex is used where `<argument_name>` is replaced by the name of the argument that is to be checked:

```
<argument_name>[:\-=(\s`]+\s*(the|a)?\s*str(ing)?(?!\\w)
```

Figure 3.5 visualises this regular expression as a graph.

Other type annotations, like type hints, are not captured by Boa and therefore are not covered within the analysis. Figure 3.4 shows an example of a method argument being recognized as a string by the docstring definition.

```
1 def _parse_content_type (content_type):
2     """
3     content_type : string
4     """
5     ...
```

Figure 3.4: Method with docstrings indicating the type of the method argument.

3.1.3 Declared as String

When assigning a value to a variable, the expression undergoes analysis to ascertain whether it yields a string result. To achieve this, we traverse the computed DDG and scrutinize the expression at each node that assigns a value. We determine the return type of the assignment expression. We only focus on the return types string and string array, since these are decisive for this task. The computation of the return type involves distinct checks depending on the nature of the expressions. Table 3.1 lists the different

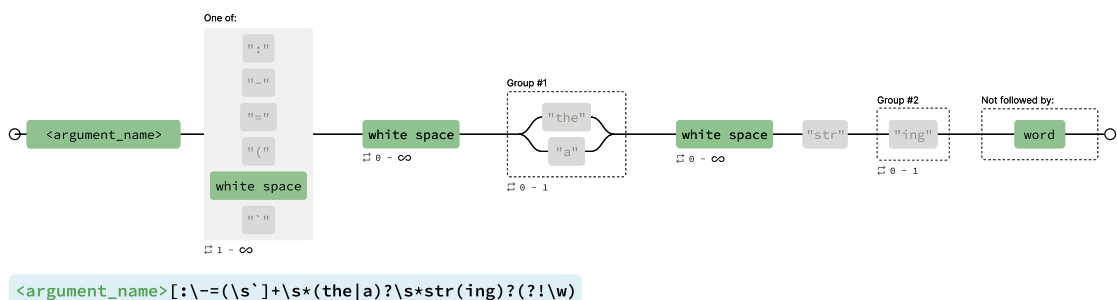


Figure 3.5: Visualisation of the regular expression for identifying string parameters in docstrings

Table 3.1: The calculation of the return type of an assign node is based on the assigned expression.

Expression Type	Description
Literal <code>"example_string"</code>	If the expression is a literal and begins with a single quote (') or double quote ("), it is recognized as a string.
Method Call <code>sample_string.trim()</code>	When the expression is a method call, an examination is conducted to verify if the invoked method returns either a string or a string array.
Array Access <code>foo = string_list[1]</code>	In cases where an array is accessed within a string list and <code>string_list</code> is known to be a string array <code>foo</code> is marked as a string.
Variable Access <code>sample_string</code> <code>sample_string_array</code>	When declaring a variable, its type is recorded for subsequent reference whenever the variable is accessed.
String Concatenation <code>sample_string + "example"</code>	If a string is combined with an expression, it implies that the expression is a string type.
Modulo String Formatting <code>"example %s" % "string"</code>	Modulo formatting of string values will return a string value.
Conditional Expressions <code>"example1"</code> <code>if sample_condition == 2"</code> <code>else "example2"</code>	When conditional expressions are used to declare a variable.
New Array <code>["example1", "example2"]</code>	The creation of a new array containing strings.

expression types. Certain identifications, such as literals, are straightforward, whereas others require more complex analysis. For example, to detect the declaration of a string array, we look for expressions of the type `NEW_ARRAY`. In this case, we must determine the return values of the child expressions, which are the items of the array. If these child expressions are of type string, the array can be identified as a string array. To detect the return type of the child expressions, we check the expressions against all the types listed in Table 3.1. Additionally, we keep track of already declared variables and their types to utilize this information when a variable is accessed.

If a value is assigned to a variable, the expression is analyzed to determine whether it re-

turns a string. For this, a list of known methods that return a string is used (Table 3.2). In the same way, it is also determined whether the assignment is a list of strings. In this way, strings accessed through indexing (e.g. `street_number = address.split(' ')[1]`) are identified.

3.1.4 Used as String

Similar to determining the return type of an expression, we analyze every expression to retrieve a list of variable names used as strings. Specifically, for expressions classified as method calls, we scrutinize whether the invoked method is among our predefined set of known methods that work with strings. If the class function of a variable is a string method (e.g. `name.trim()`) or if a variable serves as a parameter in a method that exclusively deals with strings (e.g. `"Hello Bob!".find(name)`), we mark the variable as a string. The string methods and methods with string parameters we utilize are listed in Table 3.2. Additionally, we also check for instances where a variable is used in a conditional expression and the other operand is a string (e.g., `name == "Bob"`).

3.1.5 Limitations

Relying on a list of known string methods leads to limitations, as there is no guarantee that a class function of a variable of unknown type is a string method. It could be a user-defined method or even part of a package. For instance, the `split` method from the Python library `TensorFlow` is neither a string method nor does it return a string array. In the evaluation process, we found three packages (`TensorFlow`, `PyTorch`, and `NumPy`) that were predominantly utilized with their `split` methods. As these `split` methods are not string methods, we explicitly excluded them from the identification process to mitigate the misidentification of string variables.

3.1.6 Evaluation

Evaluating the results of identifying string variables, precision was analysed to measure the accuracy of the algorithm. Precision measures the proportion of correctly identified string variables out of all variables identified as strings. For precision assessment, a random sample of 100 nodes identified as strings was inspected. Among these, 93 out of 100 nodes were accurately identified as strings. Thus, the precision of identifying string variables is 93%, indicating a high degree of accuracy in correctly identifying strings.

3.2 Parser Determination

Following the identification of string variables, we generate a dataset containing all ad hoc parser snippets. Figure 3.6 shows the main steps performed to generate the dataset. For each string variable, we construct a program slice, resulting in a set of program slices for each method, each representing a parser. In the second step, each slice is then transformed into a Python file, only containing the parsing class. As a result, we obtain

Table 3.2: List of known methods divided into categories they are used for.

	Parsing	Parsing parameters	Parser Terminating	String Returning	String Array Returning	String Methods	Methods with String Parameters
split	✓				✓	✓	✓
parse	✓	✓					
unpack	✓						
replace	✓			✓	✓	✓	
find	✓					✓	✓
rfind	✓					✓	
lfind	✓						
partition	✓				✓	✓	✓
rpartition	✓				✓	✓	✓
format	✓					✓	✓
strftime	✓					✓	✓
encode	✓					✓	
loads		✓					
unquote		✓					
int		✓					
float		✓					
chr		✓					
dumps		✓					
quote		✓					
bytes		✓					
read			✓	✓			
open			✓				
from_file			✓				
connect			✓				
readline			✓				
end			✓				
get			✓				
call			✓				
run			✓				
load			✓				
join				✓		✓	
translate				✓		✓	
strip				✓		✓	
lstrip				✓		✓	
rstrip				✓		✓	
lower				✓		✓	
upper				✓		✓	
realpath				✓			

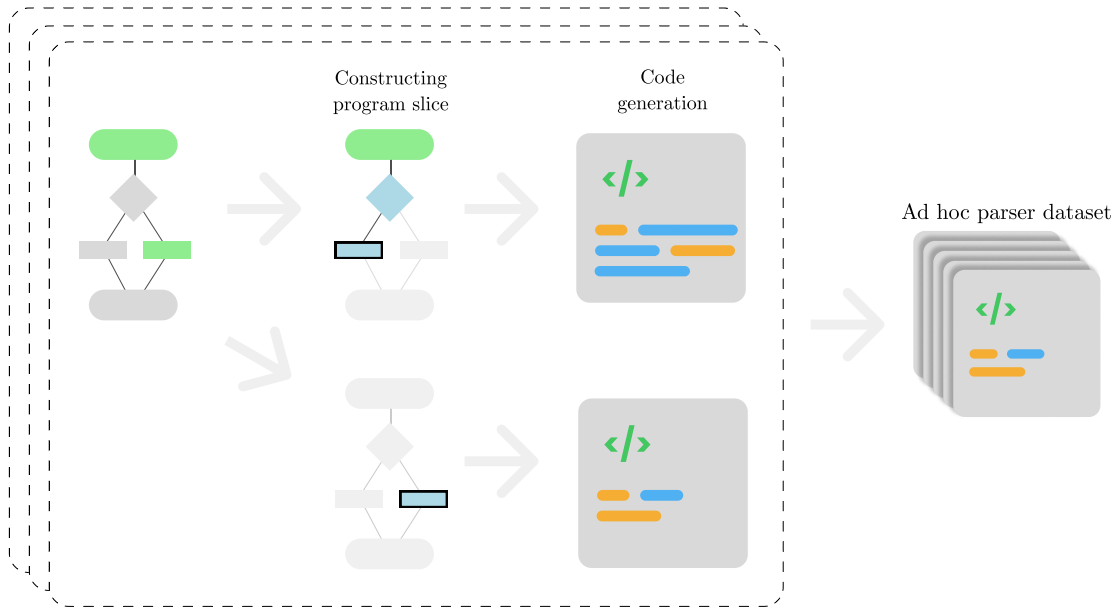


Figure 3.6: The data set of ad hoc parsers is generated in two steps: 1. Constructing a program slice 2. Generating Python code.

a dataset of all ad hoc parsers, each in its own file. In this section, we cover the process of constructing an ad hoc parser slice by traversing the DDG.

3.2.1 Constructing the Program Slices

This section contains a detailed description of how the program slice of an ad hoc parser is generated. This construction is performed by analyzing the DDG of each function. The result of the program slice is a set of nodes, where each node represents a statement of the AST. We distinguish between two types of nodes: parsing and non-parsing nodes. Parsing nodes perform the actual parsing tasks. An example of a parsing node would be `params = content_type.split(';')`. There are several criteria to identify parsing nodes, which are explained in Section 3.2.2. Supporting nodes are nodes that do not contain parsing tasks but are still necessary for the control flow of the parser logic.

When constructing parser slices, we consistently initiate the process from the input variables. The process of retrieving parsers within a method is described in Algorithm 3.1. For each method, we generate a set of parsers. The parsers are constructed starting by the input variables. These variables are either method arguments or variables assigned within the method body. When the input variable is a method argument, the starting node of the snippet is the entry node of the DDG. It should be noted that the entry node in the DDG has successors for all method parameters. Therefore, it is crucial to constrain these successors to utilize only those that possess actual data dependence with the input variable for further search.

Algorithm 3.1: Parser-Detection**Input:** Data dependence graph DDG , set of string arguments $StringArgs$ **Output:** Set R of tuples (P_i, S_i) where P_i is a set of parser nodes and S_i is a set of supporting nodes

```

1 Function Main:
2    $R \leftarrow \emptyset$  // parsers in method
3    $A \leftarrow \emptyset$  // already assigned nodes
4   foreach  $node \in DDG$  do
5     if  $node$  is entry node then
6       for  $argument \in StringArgs$  do
7          $(P, S) \leftarrow \text{computeParser}(node, argument)$ 
8          $R \leftarrow R \cup \{(P, S)\}$ 
9     else if  $node$  is a string declaration &  $node \notin A$  then
10       $(P, S) \leftarrow \text{computeParser}(node)$ 
11       $R \leftarrow R \cup \{(P, S)\}$ 
12       $A \leftarrow A \cup P \cup S$ 
13   return  $R$ 

14 Function computeParserForArgument( $node, methodArgument$ ):
15    $P \leftarrow \emptyset$ 
16    $S \leftarrow \emptyset$ 
17   foreach  $succ \in Successors(node)$  do
18     if  $methodArgument$  is not used in  $succ$  then
19       continue
20      $(P_{succ}, S_{succ}) \leftarrow \text{computeParser}(succ)$ 
21     if  $P_{succ} \neq \emptyset$  then
22        $S \leftarrow S \cup \{node\}$ 
23        $P \leftarrow P \cup P_{succ}$ 
24        $S \leftarrow S \cup S_{succ}$ 
25   return  $(P, S)$ 

26 Function computeParser( $node$ ):
27    $P \leftarrow \emptyset$ 
28    $S \leftarrow \emptyset$ 
29    $expr \leftarrow Expressions(node)$ 
30   if  $expr$  is parser terminating expression then
31     return  $(\emptyset, \emptyset)$ 
32   if  $expr$  is parsing expression then
33      $P \leftarrow P \cup \{node\}$ 
34   foreach  $succ \in Successors(node)$  do
35      $(P_{succ}, S_{succ}) \leftarrow \text{computeParser}(succ)$ 
36     if  $P_{succ} \neq \emptyset$  then
37        $S \leftarrow S \cup \{node\}$ 
38        $P \leftarrow P \cup P_{succ}$ 
39        $S \leftarrow S \cup S_{succ}$ 
40   return  $(P, S)$ 

```

To construct a parser, we iterate through the nodes of the DDG looking for a starting point. A potential starting node of an ad hoc parser is either a string declaration or a string method parameter. If a node has been marked as a string declaration (Section 3.1) and is not already part of another ad hoc parser, we start the slicing process, as you can see in line 10. Because method parameters are not represented in a DDG, we need to handle them explicitly. In this case, the starting point is the entry node of the DDG. In a DDG, the successor nodes of the entry node include all different data flows. Therefore, we need to restrict our slicing to those that have successor nodes that are actually using the string parameter (line 18). For that node, the slicing process is started. In the slicing process, each successor node in the DDG is subsequently traversed and analyzed. If the respective node is identified as a parsing node, it is added to the parsing set. If it is not a parsing node, it is added to the supporting set, provided that this node has at least one successor that is a parsing node (line 22). Capturing supporting nodes ensures that the code snippet encapsulates the core of the ad hoc parser, while maintaining its completeness and capturing those statements that might be relevant for the parsing process.

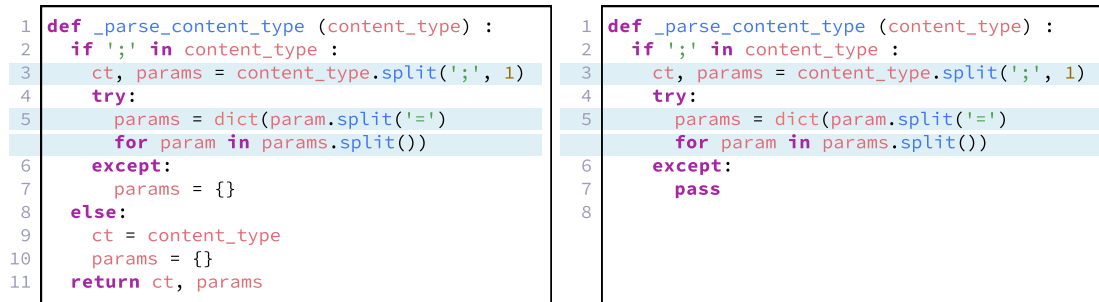
There are also methods that interrupt a parsing process. Reading files or sending a network request often takes a string as input, like a file path or URI, and returns another string. We call these functions parser-terminating functions because the loading of data from a file or an API is treated as an interruption of the parsing process, as the result is not a transformation of the input string. For instance, the preparation of a URL string may require parsing to ensure its validity. Then a parser terminating function like `requests.get()` is called to load some data. The data might be the input of a second parser but not a part of the first one. All methods that are considered parser-terminating are listed in Table 3.2.

3.2.2 Identifying Parsing Nodes

To establish precise criteria for the identification of parser methods, different metrics were analyzed. We used a list of known methods that perform parsing operations. These methods are listed in Table 3.2. If a node executes one of these methods, it is considered a parsing node. Additionally, to this list of known functions, we also tried to look for other function usage that might perform parsing tasks, like methods that accept a string as input and return a string as output. However, this included too many false positive parsers due to the broad specification, and to inspect the internal workings of other methods. We analyzed methods solely used for parsing. These methods were identified by their unique structure, having a string method parameter as parsing input and an ending node representing the return of the parsed data. We could construct an initial naming metric, thereby enhancing the parsing detection precision. Additionally, we considered the deconstruction of strings, such as `x, y = strings`, as a parsing operation, further refining the criteria for identifying parsing nodes.

3.2.3 Code Generation

To extract program slices that only include the parsing code, we transform the identified parsing sets back into Python source code. The parsing set is created by examining the DDG, and although it contains all of the statements of the parser, transforming only these statements into source code does not guarantee syntactic correctness. To illustrate, if parsing statements are present within a `try` branch but not within the `except` branch, only the `try` statement would be included in the parsing set. Consequently, the parsing snippet set would not include the `except` statement. To guarantee syntactic correctness in Python code, the `except` statement must be included, even if it has no parsing statements. Thus, the source code is reconstructed based on the abstract syntax tree (AST), with only the relevant elements relevant to the parser included in the source code based on the generated parsing set. An example of how a parser snippet result may look is visualized in Figure 3.7.



<pre> 1 def _parse_content_type (content_type) : 2 if ';' in content_type : 3 ct, params = content_type.split(';', 1) 4 try: 5 params = dict(param.split('=') 6 for param in params.split()) 7 except: 8 params = {} 9 else: 10 ct = content_type 11 params = {} 12 return ct, params </pre>	<pre> 1 def _parse_content_type (content_type) : 2 if ';' in content_type : 3 ct, params = content_type.split(';', 1) 4 try: 5 params = dict(param.split('=') 6 for param in params.split()) 7 except: 8 pass </pre>
---	--

Figure 3.7: Comparison of original source code on the left to generated parser snippet result on the right. The identified parsing set is highlighted in blue.

In the event that a statement is identified as part of the parser but its contained suite is not, the suite will be replaced by the expression `pass`. To illustrate, in instances where an `if` statement incorporates parsing operations or an `except` statement is mandatory due to syntactic correctness, yet the suite within does not encompass any parsing operations, the `pass` expression is included to ensure syntactic correctness. This can be observed in Figure 3.7, where the suite inside the `except` statement is not part of the parser and is therefore replaced by `pass`.

In order to transform the AST into Python code, a bespoke `prettyprint` method was employed, as the existing `prettyprint` method of BOA is incomplete. The lack of precision in the BOA AST results in the loss of certain characteristics of the original Python code. For instance, var-positional and var-keyword parameters are not identified, and similarly, lambda expressions within list comprehensions are not recognized. Additionally, the slice notation is not identified. Of the 37.788 generated parser snippets, 2.286 were filtered out. Such errors also contribute to the generation of Python code snippets that are, at least partly, syntactically incorrect. Code that is not compilable in Python 3 is automatically excluded from the study. Some examples where the Boa AST is incorrect are listed in

Table 3.3: Examples of not correctly compiled Boa AST and whether they are excluded from the final data set.

Original Code	Boa AST	Excluded from final dataset
<code>**kwargs</code>	<code>kwargs</code>	in some cases
<code>*args</code>	<code>args</code>	in some cases
<code>densityarr[0,:]</code>	<code>densityarr[0] []</code>	yes
<code>inBuf[:]</code>	<code>inBuf[]</code>	yes
<code>inBuf[:1]</code>	<code>inBuf[1]</code>	no

Table 3.3. Unfortunately, there are cases where it is not possible to ascertain whether the AST has been compiled correctly or not. Consequently, we cannot guarantee that the newly generated Python code is semantically identical to the original source code.

3.2.4 Evaluation

In order to verify the quality of the parser recognition methodology, we performed an evaluation based on a random sample analysis in which randomly selected methods from the dataset were manually checked.

Test criteria and methodology

True positives were identified by systematically testing a random sample of methods. A sample was defined as a randomly selected set of techniques, which were then analyzed manually. The aim of this analysis was to determine whether the respective method contained a parser. The classification was binary with the categories “Yes” (contains a parser) and “No” (does not contain a parser).

For each method, the contained string variables were examined, as these typically form the basis for further processing by parsing methods. The definition of what is considered a parsing method has already been described in detail in Section 3.2.2.

Precision

The precision of the parser identification and slicing process was evaluated by analyzing a sample of 126 parser snippets. Of these, 108 were correctly identified as parsers, while the remaining snippets were incorrectly identified. To assess the precision more rigorously, we calculated a 95% confidence interval for the proportion of correctly identified parser snippets. Based on the analysis, the confidence interval for the precision of parser snippets is approximately (0.8040, 0.9360). This indicates a 95% confidence level that the true proportion of correctly identified parser snippets lies within this interval.

We examined the reasons behind the incorrect classification of certain code snippets as parsers. Table 3.4 shows the occurrence of the reasons. Overall, the errors were caused in three different stages of creating the slice: the string identification, ad hoc

Table 3.4: False positive instances and the reasons behind their classification as such. Out of 128 instances, 18 were identified as false positives

Reason for the classification as a false positive	Amount
String method has no actual constraints on input	4
Issue with the source code generation	3
A wrong input is identified for this parser	3
Input is not of type string	3
Parser is just a smaller part of the identified snippet	2
Others	3

parser extraction, and source code generation. Further studies could specifically analyse the identified causes of errors and develop strategies to minimize them. Examples of this could be the improvement of string recognition through semantic analysis or the differentiation between real and overwritten parsing methods. Such approaches would contribute directly to improving precision and reducing the number of false positives.

Challenges in Recall Calculation

In evaluating the performance of a heuristic, recall is a critical metric. Recall is measuring the proportion of true positive instances correctly identified out of all actual positives. However, the significance of this metric depends on sufficient true positive instances within the sample. When the prevalence of true positives in the dataset is low, the calculation of the recall becomes inherently unstable and less meaningful.

We have examined a sample of 100 methods, out of which only 4 were identified as actually containing a parser (true positives). This corresponds to a prevalence of only 4%. Such a low number of true positives poses two major challenges:

Statistical instability With only 4 true positives, every fluctuation in the identification of true positives (overlooked or additionally recognized parsers) leads to a disproportionate modification of the recall value. If, for example, only 3 out of the 4 parsers would be identified, the recall drops to 75%. This sensitivity makes the metric unreliable for robust conclusions.

Non-representative sample When the sample just contains a few true positives, the distribution of positive instances in the dataset might not be represented correctly. A recall calculated on such a sample may therefore not be generalizable and does not adequately reflect the actual performance of the heuristic on the total population.

CHAPTER 4

Findings

This chapter deals with the summarization, analysis, and presentation of the obtained metrics of the ad hoc parser dataset and is divided into nine parts, each focusing on different aspects of ad hoc parsers: the frequency of ad hoc parsers (4.2), where they are located (4.3), their size (4.4), information about the input variable (4.5), what functions are used and how they are used (4.6), usage of regular expressions (4.7), the nature of loops (4.8), and error handling (4.9).

4.1 Dataset Categorization and Analysis

For our analysis, we categorize the main dataset into subsets based on the parser size and other features: 1. One Liners 2. Small 3. Medium 4. Large 5. Very-Large 6. File Input 7. Test File 8. Test Method 9. Regular Expressions 10. Parsing Method.

4.1.1 Size-Based Categories

The size-based categories — One Liners, Small, Medium, Large, Very-Large — were derived based on the number of lines of code of each parser. This separation allows us to analyze trends across parsers of different sizes.

1. One Liners (1 line)
2. Small (2-5 lines)
3. Medium (6-10 lines)
4. Large (10 -50 lines)
5. Very-Large (>50 lines)

4.1.2 Feature-Based Categories

In addition to size-based categories, we also categorized parsers based on specific features.

1. File Inputs: Parsers process input from a file (Figure 4.1). 5% of all ad hoc parsers use files as parser input.

```

1 def GenerateMessages(messageSet, args):
2     for i, line in enumerate(lines):
3         line = line.strip()
4         if not multiline:
5             if line.startswith('msgid '):
6                 text = line.split('msgid ')[1]

```

Figure 4.1: Example of an ad hoc parser with file as input

2. Test Methods: This set contains parsers that are within a test method. All methods starting with `test_` are considered to be test methods.

```

1 def test_diss_indexer_run_2(self):
2     expected = pd.read_pickle(os.path.join(VIS_PATH, 'tests',
3         'expecteds', 'test_dissonance_thorough.pickle'))
4     expected.replace('Z', '-', inplace = True)
5     expected.replace('0', '-', inplace = True)

```

Figure 4.2: Example of an ad hoc parser inside a test method

3. Test Files: If a file name starts with `test`, we consider those as test files. This set contains all parsers located inside a test file.

```

1 def fixup(m):
2     text = m.group(0)
3     if text[2] == "&#":
4         try:
5             if text[3] == "&#x":
6                 return unichr(int(text[3: -1], 16))
7             else:
8                 return unichr(int(text[2: -1]))
9         except ValueError:
10             pass
11     return text

```

Figure 4.3: Example of an ad hoc parser inside a test file

4. Parsing Methods: These are functions with the goal of parsing an input. If the parameter of a function is also the input of an ad hoc parser and the returning value of the function is the returning value of the same parser, we call this function a parsing method. Figure 4.4 and Figure 4.6 show examples of parsing methods.

```

1 def csv_repr(v, decimal_sep = "."):
2     if isinstance(v, float):
3         return "%0.2f" % v.replace(".", decimal_sep)

```

Figure 4.4: Example of a parsing method

5. **Literal Input:** This subset contains only the parser taking literals as an input variable. Figure 4.5 shows an ad hoc parser taking a literal as input. We want to analyse if this kind of parser has a reduced complexity due to relying on predefined patterns, reducing the need for complex tokenization rules.

```

1 def errmsg(msg, lineno):
2     fmt = '%s: line %d column %d - line'
3     return fmt % (msg, lineno)

```

Figure 4.5: Example of an ad hoc parser with literal as input

6. **Regex Usings:** A significant proportion of parsers rely on regular expressions (11%).

```

1 def _indent(s, indent = 4):
2     if isinstance(s, unicode):
3         s = s.encode(_encoding, 'backslashreplace')
4     return re.sub('(?!m)^(?!$)', indent * ' ', s)

```

Figure 4.6: Examples of an ad hoc parser using regular expressions.

Overlap Between Categories

Some functional-based subsets overlap significantly. Figure 4.7 shows this intersection. *Test Methods* often overlaps with *Literal Inputs*, probably due to the nature of test methods using fixed strings as test input. *Regex Using* is also often used in combination with *File Inputs* or *Parsing Methods* due to the variability of the input string and need for pattern extraction, but less in test methods because the input strings are already known and therefore there is less need for complex pattern matching.

4.2 Frequency

To know how common ad hoc parsers are in the wild, we looked at the number of projects that contain at least one ad hoc parser. Out of 1710 projects, 1285 contain at least one ad hoc parser, meaning that about 75% of projects use ad hoc parsing. Comparing the projects that contain parsers with those that do not, we found that projects without

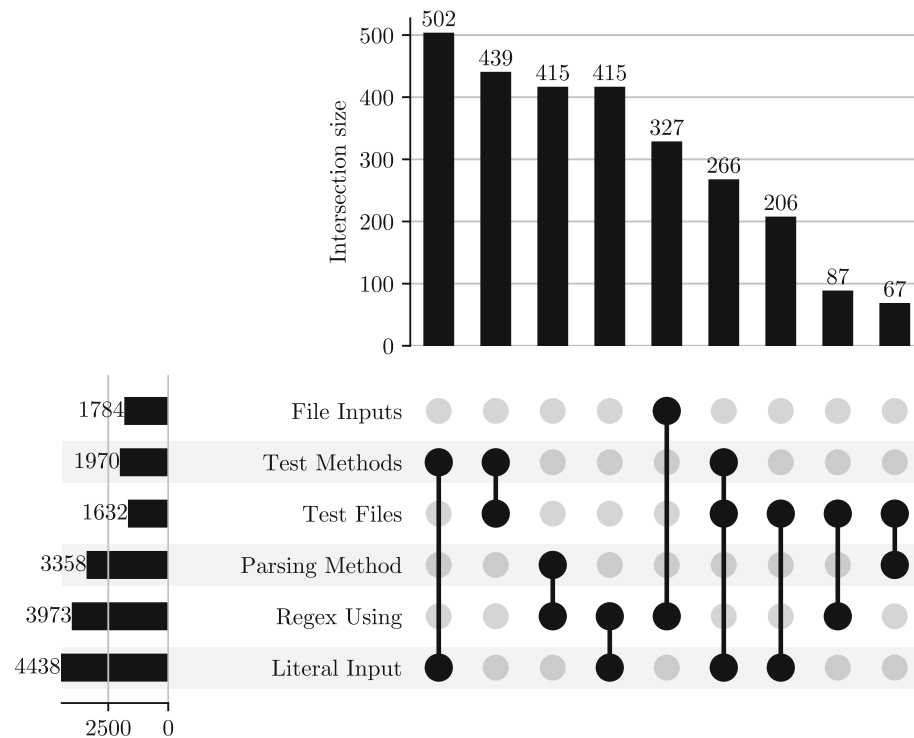


Figure 4.7: Distribution of input sources

parsers are much smaller in terms of both the number of files and the number of AST nodes (Table 4.1). This may be because smaller projects are simpler and may not need ad hoc parsing because their functionality is limited.

4.3 Location

We wanted to analyze where ad hoc parsers are located within the method. So we divided the method into four equal sections: Start, Earlier Section, Later Section, and Ending. Then we looked at which section of each method contained at least one parsing task. One might think that the parsing component of a function would typically be at the beginning, validating and transforming inputs before passing them on to the rest of the program, which is not confirmed by looking at the results. 48.6% of the methods contain parsing tasks in the first quarter (Beginning), 61.9% in the earlier section. 45.4% in the later part and 20% in the end (Figure 4.8). So, although there might be some tendency for using a parser in earlier sections of a method, parsing tasks

Table 4.1: Comparison of Projects With and Without Parsers

	No Parser		At Least One Parser	
	files	ast nodes	files	ast nodes
count	425	425	1,285	1,285
mean	9	72,972	62	2,491,491
std	18	379,478	169	10,417,270
min	1	36	1	208
25%	2	2,762	6	29,475
50%	4	9,071	18	165,130
75%	10	34,664	47	908,894
max	290	6,560,250	2,464	158,823,900

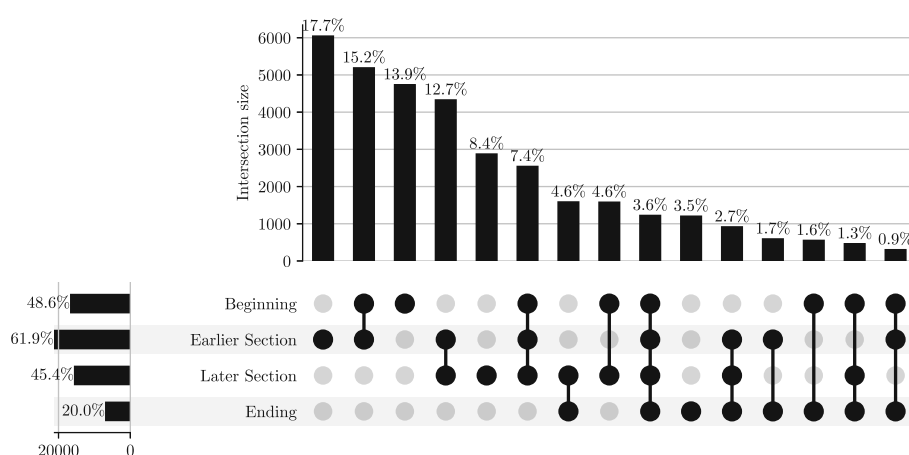


Figure 4.8: Positioning of parsers inside Methods is rather dense. It is uncommon to pause the parsing task and finish it in later sections.

occur in all sections. This supports the idea that shotgun parsing — the mixing of parsing and business logic — is a real phenomenon [22, 29]. But although parsers can be found throughout in the whole method it seems that they are rather compact, i.e., it is not common to start a parsing task at the beginning of a method and continue it at the end, which is done by only 1.6% of all parsers, which indicates that developers prefer to encapsulate parser logic within small code block rather than spreading it across the whole method.

The standard deviation of the positions of a parser gives us insight into the scatter of the parser relative to the size of the function: Half of the parsers have a standard deviation less than 0.14 and 75% of the parsers less than 0.21 (Table 4.2). This means that ad hoc parsers are rather compact and less scattered across methods, which leads to better

maintainability of ad hoc parsers.

Table 4.2: Summary Statistics for Scattering Metrics

Statistic	Range	StdDev	MAD
Count	34,915	34,915	34,915
Mean	0.34	0.15	0.15
Standard Deviation (Std)	0.24	0.10	0.10
Minimum (Min)	0.00	0.00	0.00
25th Percentile (Q1)	0.14	0.06	0.07
Median (Q2)	0.31	0.14	0.14
75th Percentile (Q3)	0.50	0.21	0.22
Maximum (Max)	1.00	0.48	0.48

4.4 Size

This section contains a discussion about the size of ad hoc parsers. By definition, they are small snippets of code, but besides the average line of code and number of expressions, we also want to analyze if ad hoc parsers regularly use temporary variables to store intermediate results or if they might prefer method chaining. Ad hoc parsers might be syntactically compact but packed with complex functionality in a small space. Analyzing the size and structure can provide insights into the readability and maintainability of ad hoc parsers.

To measure structural complexity, we looked at cyclomatic complexity, method chaining, function calls, and temporary variables. Cyclomatic complexity is expressed as the number of linearly independent paths in a program. A higher cyclomatic complexity indicates a code that is harder to understand, maintain, and hence has a higher probability of errors [21]. The method chaining is calculated by summing up all method chaining in the parser. It is only counted when consecutive method calls are applied to the result of a previous method. An example is given in Table 4.3.

$$\text{Chaining Count} = \text{Number of Consecutive Method Calls} - 1$$

The median size of 4 lines of code supports the characterization of ad hoc parsers as small code snippets. 19.9% of all parsers are even one-liners, while the existence of larger parsers also suggests that some contain complex implementations.

On average, parsers take up 1.37% of the project code, which confirms that ad hoc parsers are small components.

We found that parsers have a moderate cyclomatic complexity (median=2), but there are extreme outliers (max=182), which suggests the existence of mega parsers with

Table 4.3: Method Chaining Examples

Code Snippet	Chaining Count	Explanation
<code>input.split()</code>	0	Single method call
<code>input.strip().split()</code>	1	Two consecutive calls $\Rightarrow 2 - 1 = 1$
<code>data.filter().map().reduce()</code>	2	Three consecutive calls $\Rightarrow 3 - 1 = 2$

deeply nested logic. These outliers might represent technical debt that could benefit from refactoring into smaller components. Method chaining is used rather sparsely. 90% of the parsers do not use method chaining at all. Parsers call an average of 4.21 functions, while file input processing parsers call slightly more functions (mean = 5.95). We applied the Pearson correlation [4] to determine the linear relationships of our metrics (Table 4.4). There is a strong correlation with the use of temporary variables ($\rho=0.80$), which confirms that procedural coding predominates.

Table 4.4: Pearson Correlation Matrix of Parser Implementation Characteristics

	Temp Vars	Func Calls	Cyclo Compl	Meth Chain
Temp Vars	1.00	0.80	0.60	0.11
Func Calls	0.80	1.00	0.78	0.27
Cyclo Compl	0.60	0.78	1.00	0.11
Meth Chain	0.11	0.27	0.11	1.00

Temp Vars: Temporary Variables, Func Call: Function Calls, Cyclo Compl: Cyclomatic Complexity, Meth Chain: Method Chaining

This confirms that ad hoc parsers are predominantly small and also simple. Although there are more complex and functionally dense parsers, in general ad hoc parsers prefer to follow procedural patterns and favour explicit states (temporary variables) and modular partitioning (separate function calls) over chained method calls.

RQ1 Where do we find ad hoc parsers in Python projects?

Ad hoc parsers are widespread in Python projects and are found in an average of 75.15% of all projects. This high prevalence emphasises the popular use of ad hoc parsers. Contrary to the expectation that parsers are mainly used at the beginning of a method (for example, for input validation), there is a broad distribution across the entire body of the method. The size and complexity of ad hoc parsers may vary, but in most cases they are relatively small and simple or even single-line.

4.5 Input Sources

To investigate the input sources of ad hoc parsers, we distinguished between two kinds of sources: the programmatic source, i.e., how the input source is accessed (method argument, function call, dictionary lookup, etc.) and the actual source of the input (file input, command-line argument, config, etc.). The examples in Table 4.5 show the different types of input sources we have examined.

Table 4.5: Examples of different input sources

Input Source	Example
Config	<code>updateGameListInterval = config.get('Scheduler', '')</code>
Literal	<code>usage = "%prog [options] name=value ..."</code>
Command-line	<code>expected_help = "\n" + sys.argv[0] + ':'</code>
File input	<code>firstline = file.readline()</code>
Database query	<code>philo_id = cursor.fetchone()[0]</code>
Environment variable	<code>no_proxy = os.environ.get('no_proxy', '')</code>
Shell command output	<code>result = subprocess .run(cmd, shell = True, stdout = subprocess.PIPE) .stdout .decode()</code>
Network request	<code>scpd_body = requests .get(self.base_url + self.scpd_url, timeout = 10) .text</code>

Figure 4.9 shows the distribution of the input variable source. 43.12% of the parsers receive their input directly as a method argument, 26.26% via a function call, and 12.01% as a literal. In rare cases, the input variable is assigned values through variable assignments (5.41%) and dictionary lookups (3.57%).

The most common source of input is method arguments (65.84%). Figure 4.10 shows how programmatic sources are related to actual sources. We observe that a large proportion of method arguments are not directly used as input, but also as function calls. Figure shows an example where the actual input source is passed as a method argument, but processed before being used as an input for ad hoc parsing. 25.45% of the sources are literals being parsed, and 4.8% is text originating from file reads. There is still a large group of unknown sources, which are mostly generated by function calls, of which we can not know the exact source of the return value. Since our analysis is based on the dataset of parsing code, we don't have access to custom-written methods and therefore cannot compute the input source. Also, we do not know where the parsing function is called, and therefore, we do not know the actual input source of the method arguments.

The largest input source for parsers in test methods is literals (37.99%), suggesting that ad hoc parsers are often used in test methods to prepare literals to be tested.

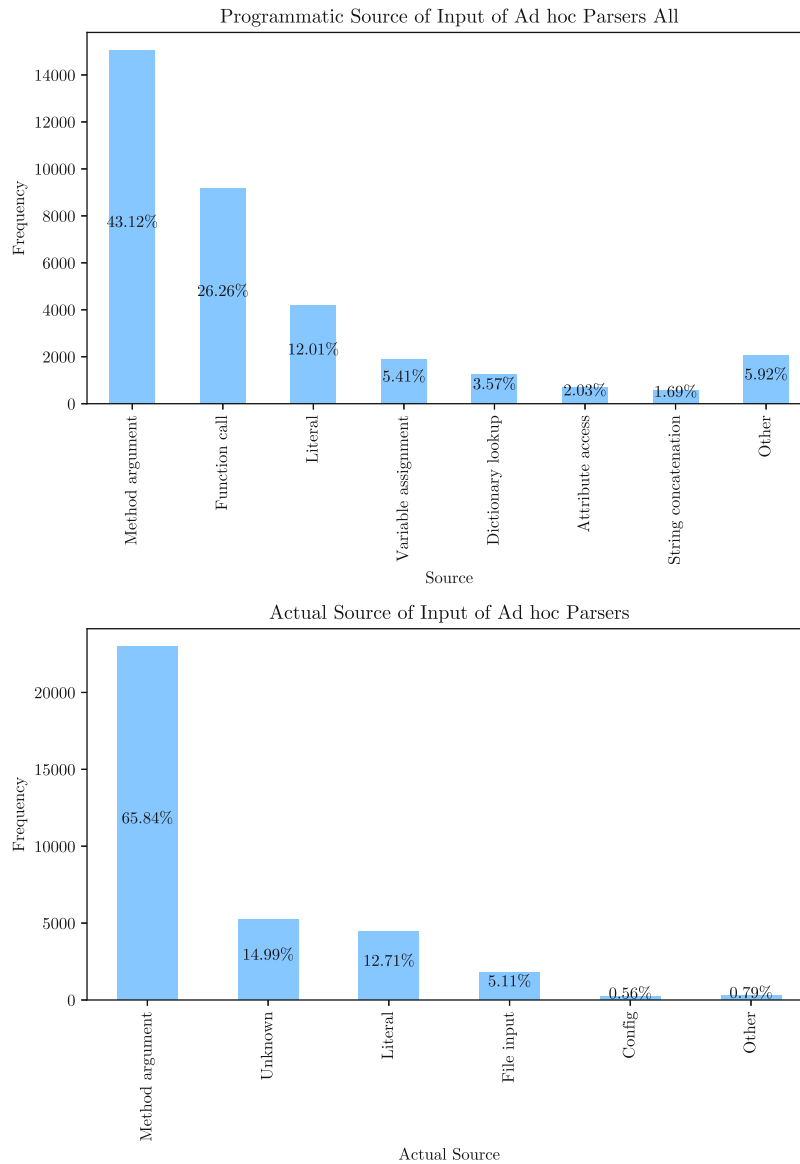


Figure 4.9: Examples of different input sources.

4.6 Function Calls

In this section, we investigate the usage of function calls in ad hoc parsers. An understanding of how functions are used provides insight into parsing patterns. Table 4.6 lists

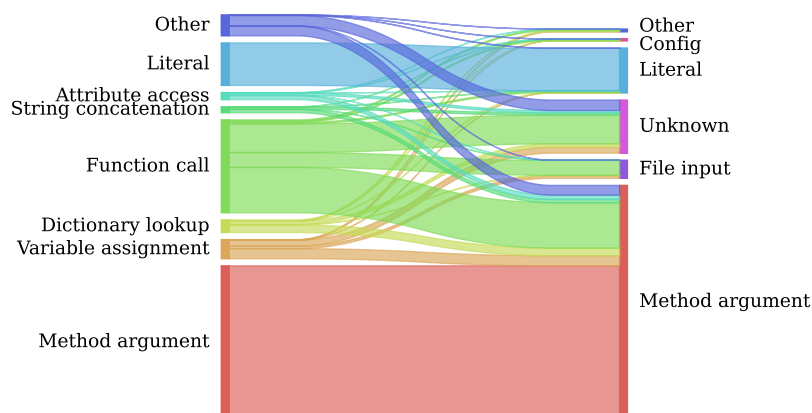


Figure 4.10: Sankey diagram, showing the connection between programmatic source and actual source of the input.

the top 20 functions used in ad hoc parsers.

Ad hoc parsers typically employ several categories of functions:

- **String manipulation functions** (e.g., `split`, `replace`, `strip`) for tokenizing and transforming text
- **Type conversion functions** (e.g., `int`, `float`, `str`) for transforming parsed strings into appropriate data types
- **Validation functions** (e.g., `startswith`, `isinstance`, `len`) for checking input validity
- **Collection operations** (e.g., `join`, `map`, slicing operations) for manipulating parsed data
- **I/O functions** (e.g., `read`, `get`) for retrieving input data

To analyze function usage patterns, we categorized the function calls within ad hoc parsers by name, frequency, and position within the parsers.

The top 5 functions (`split`, `replace`, `int`, `encode`, `len`) account for about 34% of all function usage, suggesting that these form the core toolkit for parsing operations. `int`, `str`, and `float` together account for about 9.4% of function calls, highlighting the importance of type conversion in parsing. There are also non-parsing functions that are widely used, such as `join` (6.76%), `startswith` (5.38%), and `isinstance` (4.67%). This widespread use of `startswith` and `isinstance` demonstrates the importance of input validation in parsing, as shown in Figure 4.6.

Table 4.6: Distribution of Python functions by category and usage. The column Function % displays the relative amount of the function compared to all functions in the dataset, while the column Parser % displays the percentage of parsers using that function

Function	Category	Function %	Parser %	Projects
split	String Manipulation	11.8%	35.3%	939
replace	String Manipulation	9.4%	21.1%	699
int	Type Conversion	6.5%	13.4%	624
encode	String Manipulation	3.3%	10.9%	493
len	Inspection	3.2%	9.0%	502
find	Validation	2.6%	5.8%	321
strip	String Manipulation	2.1%	6.7%	478
sub	Regular Expressions	2.0%	3.9%	305
join	String Manipulation	2.0%	6.8%	481
format	String Manipulation	1.9%	5.8%	311
startswith	Validation	1.9%	5.4%	372
isinstance	Validation	1.7%	4.7%	302
str	Type Conversion	1.6%	4.7%	345
search	Regular Expressions	1.5%	3.8%	286
group	Collection Operations	1.4%	2.5%	223
float	Type Conversion	1.4%	3.2%	256
match	Regular Expressions	1.1%	3.0%	291
read	I/O Functions	1.1%	3.1%	294
get	Collection Operations	1.0%	3.3%	302
parse	String Manipulation	1.0%	2.8%	192
open	I/O Functions	0.9%	2.9%	286
lower	String Manipulation	0.9%	2.8%	264
append	Collection Operations	0.7%	2.2%	259
findall	Regular Expressions	0.6%	1.7%	150
endswith	Validation	0.6%	1.9%	201
loads	I/O Functions	0.6%	2.1%	239
print	I/O Functions	0.5%	1.1%	126
rfind	Validation	0.5%	1.6%	146
compile	Regular Expressions	0.5%	1.7%	179
escape	Regular Expressions	0.5%	1.0%	93

The beginning section of ad hoc parsers is characterized by the presence of setup code, including imports, configuration, and initial data retrieval, which may involve fewer function calls overall. Functions such as `get` (50.32%) and `read` (46.85%) exhibit a high proportion in this section. The earlier section is likely to be utilized for the preparation of inputs, the validation of data, or the execution of initial parsing steps. Functions such as `startswith` (38.72%), `len` (31.82%), and `sub` (27.83%) are prevalent in this section.

The majority of functions are located in the final quarter of the parser (42.23%), with the ending section often responsible for final transformations, encoding, formatting, or output generation. Further, we observed this tendency of the predominance of functions in the ending section of an ad hoc parser. `encode` (52.31%), `float` (51.90%), `replace` (48.52%), and `int` (49.69%) are notable examples for this. While functions such as `startswith` or `isinstance` are typically associated with early-stage validation, they also occur in later steps of the parser. This observation suggests that these functions are not only employed for initialization but also intermediate checks during parsing.

4.7 Regular Expressions

In this section, we look at the use of regular expressions in various ad hoc parsers. Regular expressions represent a formal method of pattern matching, which is regularly used in ad hoc parsers. Understanding how regex is applied can inform static analysis techniques and parser design practices.

Regular expressions are used by 11% of the parsers analysed. File input processing parsers show the highest number of regular expressions (18%), which is most likely due to the need for structured input processing when processing file content.

There is a strong correlation between parser size and the use of regular expressions. Larger parsers have a much higher rate of regular expressions (54%), suggesting that regular expressions become increasingly valuable as parser complexity grows.

4.7.1 Function Specialization by Category

File input parsers show heavy use of `findall` (31.02%) and `search` (25.23%), suggesting that regular expressions are used to extract specific patterns from the contents of files. Parsers in test files show an exceptionally high usage of the `split` method (37.092%), suggesting that ad hoc parsers here use regex mainly for tokenisation. Test method parsers mainly use `escape` (24.24%), `match` (26.52%), and `search` (24.24%), with no use of `split`, indicating a focus on precise pattern validation. In general, the most used regular expression functions are `sub` (26.44%), `search` (23.55%), and `match` (14.79%).

4.7.2 Positional Distribution of Regex Functions

The positional analysis of regex functions indicates special usage patterns across different sections of the parser (Table 4.7). Regex functions are mainly used in later stages of the

parser. 38.2% of all regex function calls are in the last 25% of parsers, which suggests that regex is often used for refinement or transformations of already processed data.

Table 4.7: Distribution of regex functions across parser sections

Regex Function	Beginning	Earlier Section	Later Section	Ending
compile	45 (11.1%)	64 (15.8%)	75 (18.5%)	222 (54.7%)
escape	24 (4.5%)	91 (17.1%)	183 (34.4%)	234 (44.0%)
findall	92 (19.3%)	136 (28.5%)	103 (21.6%)	146 (30.6%)
finditer	24 (16.0%)	44 (29.3%)	46 (30.7%)	36 (24.0%)
match	152 (17.1%)	197 (22.2%)	217 (24.4%)	322 (36.3%)
search	256 (18.1%)	280 (19.8%)	385 (27.2%)	493 (34.9%)
split	139 (25.6%)	90 (16.6%)	150 (27.6%)	164 (30.2%)
sub	229 (14.4%)	346 (21.8%)	340 (21.4%)	673 (42.4%)
subn	2 (28.6%)	2 (28.6%)	0 (0.0%)	3 (42.9%)
Total	963 (16.0%)	1250 (20.8%)	1499 (25.0%)	2293 (38.2%)

compile and escape show a strong tendency toward later positions, probably due to optimization of repetitive pattern use, while findall and finditer have a balanced distribution.

Only 16% of regex functions are used at the beginning, while 38.3% are used in the ending section. This indicates that regex is mostly used for final processing instead of initial tokenisation.

4.7.3 Pattern Usage Distribution of Regex Functions

To understand the use of regular expression patterns, we categorized each pattern based on its structural features. This categorization was inspired by prior research on regex feature usage by Chapman and Stolee [7]. The pattern `^[0-9]+$`, for example, contains five different types of tokens. It has the start anchor (STR) and the end anchor (END), which are specified using the caret `^` at the beginning and the dollar sign `$` at the end of a pattern, the custom character class (CCR) using pairs of brackets `[...]`. It also utilizes a range (RNG), specified by a hyphen `[0-9]` and additional repetition (ADD) expressed by using plus `+`. The full list of features and abbreviations is provided in Table 4.8. CCC is the most prevalent category, appearing in about 56% of patterns. Repetition Operators (ADD: 46.7%, KLE: 36.1%, QST: 26.6%) are widely used. Compared to the findings of Chapman and Stolee [7] we found that some features like CCC (+15.2%) and STR (+11.2%) have a much higher prevalence in ad hoc parsers, while features like KLE (-11.3%) and ANY (-10.1%) are used way less compared to the results of Chapman and Stolee [7]. This suggests that the programmers use less ambiguous regular expressions for ad hoc parsers. ANY and KLE can lead to overmatching, while CCC and ADD are more restrictive.

Table 4.8: Regex Pattern Categories with Usage Statistics

Code	Description	Example	nPatterns (%)	nProjects (%)
CCC	custom character class	[aeiou]	185 (48.1%)	91 (66.9%)
ADD	one-or-more repetition	z+	175 (45.5%)	78 (57.4%)
STR	start-of-line	^{ }	144 (37.4%)	68 (50.0%)
CG	capture group	(caught)	158 (41.0%)	67 (49.3%)
RNG	character range	[a-z]	106 (27.5%)	60 (44.1%)
KLE	zero-or-more repetition	.*	127 (33.0%)	59 (43.4%)
ANY	any non-newline char	.	93 (24.2%)	47 (34.6%)
QST	zero-or-one repetition	z?	97 (25.2%)	44 (32.4%)
END	end-of-line	\\$	67 (17.4%)	39 (28.7%)
NCCC	negated character class	[^{}qwxzf]	67 (17.4%)	37 (27.2%)
DEC	decimal digits	\d	58 (15.1%)	35 (25.7%)
WSP	whitespace	\s	92 (23.9%)	30 (22.1%)
OR	logical or	a b	50 (13.0%)	27 (19.9%)
WRD	word characters	\w	33 (8.6%)	21 (15.4%)
LZY	lazy repetition	z+?	33 (8.6%)	16 (11.8%)
SNG	exact repetition	z\{8\}	18 (4.7%)	13 (9.6%)
NCG	non-capturing group	(?:b)	15 (3.9%)	13 (9.6%)
NWRD	non-word chars	\W	9 (2.3%)	8 (5.9%)
NWSP	non-whitespace	\S	10 (2.6%)	6 (4.4%)
OPT	options wrapper	(?i)	16 (4.2%)	6 (4.4%)
NLKA	negative lookahead	a(?!yz)	5 (1.3%)	5 (3.7%)
LKB	positive lookbehind	(?<=a)bc	4 (1.0%)	4 (2.9%)
NLKB	negative lookbehind	(?<!x)yz	3 (0.8%)	3 (2.2%)
LWB	at least n repetitions	z{15,}	3 (0.8%)	3 (2.2%)
PNG	named capture group	(?P<name>x)	7 (1.8%)	3 (2.2%)
DBB	bounded repetition	z{3,8}	3 (0.8%)	3 (2.2%)
LKA	positive lookahead	a(?:=bc)	2 (0.5%)	2 (1.5%)
WNW	word boundary	\b	6 (1.6%)	2 (1.5%)
ENDZ	absolute end of string	\Z	2 (0.5%)	1 (0.7%)

Note: Percentages calculated relative to total patterns (385) and projects (136).

4.8 Loops

Parsers employ various looping constructs to iterate over the input string. In this section, we examine these constructs. Therefore, we categorized loops based on type (`for`, `while`), nesting depth, and bound complexity. Additionally, we also investigated the use of list comprehensions and sequence operations (`map`, `index`, `enumerate`, `zip`, `filter`) as alternatives to explicit loops.

Explicit loops occur relatively often. 26.58% of our analyzed ad hoc parsers had explicit loops (24.46% contain `for` statements, and 2.9% contain `while` statements). Parsers with file processing tasks have the highest prevalence of explicit loops (47.59%). This may result from incremental processing in file processing tasks, where files are read line

by line.

The nesting depth of loops is rather shallow. 73.42% of loops have no nesting. 22.63% have a single level of nesting, 3% have a nesting depth of 2 and fewer than 1% have a nesting ≥ 3 .

We defined three categories of loop bounds: 1. Constant (4.1%): Fixed iteration count. 2. Linear on input (48.2%): Directly proportional to input size. 3. Complex (47.8%): Dependent on conditions beyond simple input length.

The prevalence of shallow nesting and linear bound suggests that ad hoc parsers generally favor simple, straightforward iteration strategies. However, there is a significant presence of complex loop bounds, indicating that many parsers involve a non-trivial iteration logic.

The variation in loop usage across different parser categories (e.g, file input vs. regex-based parsing) indicates that the nature of the parsing task influences the loop structure and complexity.

4.9 Error Handling

This section deals with error handling within the parsers. The results include patterns of how parsers handle exceptions. Table 4.9 lists the occurrence of exception handling across the datasets. Only 14.0% of parsers contain explicit exception handling using try blocks. 13.4% contain except blocks, and only 1.0% implement a finally clause. 4.1% explicitly raise an exception.

Table 4.9: Exception Handling Statistics Across Different Code Categories

Category	try	except	finally	explicit raise
All	14.0%	13.4%	1.0%	4.1%
Very-Large	39.4%	39.4%	8.6%	19.9%
Large	38.3%	37.2%	3.0%	15.8%
Medium	27.7%	26.4%	2.0%	6.1%
Small	11.5%	10.8%	0.7%	2.5%
One Liners	0.0%	0.0%	0.0%	0.0%
Test Files	8.7%	7.4%	1.9%	1.6%
Test Methods	5.3%	3.5%	1.9%	0.3%
File Inputs	30.0%	27.7%	3.9%	7.9%
Regex using	15.8%	15.0%	2.4%	7.1%
Parsing Method	13.3%	13.2%	0.3%	5.4%
Possible Throwing	18.4%	17.2%	2.0%	5.7%

Parsers that read their input from files show the highest usage (30.0%) of error handling. This significantly higher value draws attention to the need to act on the I/O exception handling of files. Test methods, on the other hand, show relatively low exception handling (only 5.4% of parsers in test methods use try blocks) but have a relatively high proportion

of `finally` clauses (1.9%) relative to general exception handling, which is presumably due to cleanup operations of the test environments.

There is a strong correlation between the size of parsers and exception handling. Ultra-large parsers show the highest proportion of exception handling (38.3% of try blocks and exception blocks), 8.6% use finally blocks, and 19.9% explicitly throw exceptions. The large parser set shows a similarly high number of try blocks (38.3%) with a noticeably lower use of finally clauses (3%). Medium parsers show a moderate use of try blocks (27.7%), while small parsers show a significantly lower number (11.5%). 29.1% of the parsers are using functions that possibly raise exceptions; however, only 18.4% of these exception-throwing parsers include explicit error handling, which means that 81.6% of the parsers that could throw exceptions are not handled locally but are propagated further.

This analysis shows that most parsers lack exception handling. The limited implementation of exception handling combined with a relatively high number of parsers that may throw unhandled exceptions indicates that many ad hoc parsers may be susceptible to errors occurring when handling invalid inputs. This finding emphasizes the importance of robust error-handling policies for parser design, especially for parsers with untrusted input.

RQ2 What are the syntactic characteristics of ad hoc parsers?

The input source of ad hoc parsers is mainly method arguments, 12.71% of the parsers process literals. 5.11% of the parsers obtain the text to be processed from files. The most frequently used functions are `split`, `replace`, `int`, `encode`, `len`. In addition to the parsing functions, functions for type conversion (`int`, `str`, `float`) or for input validation (`startswith`, `isinstance`) are frequently used. Regular expressions are used by 11% of all parsers and even more by larger ad hoc parsers. Regular expressions are mainly used in later phases of parsing for refinement or transformation of already processed data. The simple nature of ad hoc parsers is also reflected in the use of loop constructs. ad hoc parsers favour explicit `for`-loops with low nesting depth. Only 13.98% of ad hoc parsers handle exceptions locally. 81.6% of parsers that could potentially throw exceptions have no local error handling.

These results show that ad hoc parsers in Python typically are compact, procedural structures and contain limited exception handling. Ad hoc parsers often use string manipulation functions and also regular expressions. Different variations in size, complexity, and exception handling indicate different parsing requirements and potential robustness issues.

4.10 Threats to Validity

This section covers the discussion about internal and external validity.

4.10.1 Internal Validity

Identification of string variables: We inferred string variables based on heuristics and name conventions. This could lead to misclassifications.

False positive Parser: Like shown in Table 3.4, 14% of the ad hoc parsers were falsely identified as such.

Boa AST limitation: The AST of Boa does not cover all code constructs (e.g., slicing, variable-positional and variable-keyword arguments, lambda expressions inside list comprehensions). To mitigate this, we tried to infer these constructs where it was possible. Due to missing information, some code snippets were not compilable and therefore excluded from the study.

4.10.2 External Validity

Scope of the dataset: Our thesis is limited by the scope of our test set. Due to a bug in Boa and time limitation, it was not possible to generate an ad hoc parser dataset for all the 102.424 projects on Boa.

Programming language: The results of this study only apply to Python projects and ad hoc parser code snippets that can be compiled with Python 3.13. However, due to our modular structure and due to using Boa with its own AST, this research can be extended to other programming languages supported by Boa with few adaptations.

4.10.3 Construct Validity

Definition of ad hoc parser: Our definition of ad hoc parser (Section 3.2.2) is very broad. Our detection is based on the predefined list of methods that typically perform string operations (Table 3.2). Due to the fact that Python is dynamically typed, we cannot assure that these functions are, in fact, performed on strings.

Recall issues: As described in 3.2.4, due to the low prevalence of true positives in the sample size, the calculation of the recall was not possible.

Conclusion and Future Work

This thesis sheds light on the usage of ad hoc parsers in Python. We have successfully identified ad hoc parsers and analyzed them. We have shown that ad hoc parsers are

- **Widespread:** We found ad hoc parsers in the majority of projects.
- **Small:** With a median size of 4 lines of code ad hoc parsers are rather small.
- **Compact:** Ad hoc parsing is mixed with business logic, but is compact in itself. Meaning it is unlikely to start a parsing process at the beginning of a method, pausing it for other logic, and then continuing the parsing process later on.
- **Simple:** Most ad hoc parsers have a low cyclomatic complexity, prefer a straight-forward control flow, and use low nesting.
- **Procedural:** Ad hoc tend to be written in a procedural style. They use temporary variables and prefer separate function calls over method chaining.
- **Error prone:** Exception handling in ad hoc parsers is rare. 80% of possibly error-throwing parsers have no exception handling, making them vulnerable to runtime errors.

Now we know that ad hoc parsers often use string manipulating functions (e.g., `split`, `split` or `strip`), type conversion functions (e.g., `int`, `float`, `str`), and validation functions (e.g., `int`, `float`, `str`). We also found that 10% of ad hoc parsers use regular expressions, which are rather used at the end of ad hoc parsing. The top regular expressions features used in ad hoc parsers are custom character classes (`[aeiou]`), one-or-more repetition (`z+`), start of line (`^{\}`), and group capturing (`(caught)`).

In addition to these results, we provide an infrastructure for future work. We have created a dataset containing 34,925 snippets of real-world ad hoc parsers. This dataset can be utilized for future research. Our Boa script for generating this dataset can be adapted for other programming languages that are supported by Boa, like Java or Kotlin. As we are one of the first to work extensively with Boas program analysis tools, like graph

traversal in the Python dataset, we could identify some issues and limitations that were not documented until now. We informed the developers of Boa and thus contributed to further development and improvement of this framework.

We found that projects not containing ad hoc parsers are smaller in terms of file size and AST nodes compared to projects containing ad hoc parsers. More precise analysis could give insights into why ad hoc parsers are not needed in these projects. The dataset and result provided can be used to train machine learning models to identify ad hoc parsers. Future research could also go on to more precisely analyze input sources of ad hoc parsers. Due to time constraints, we were not able to trace the source of the input outside of the method that contains the ad hoc parsers. With Boa closing open issues, the study can be reproduced to generate a larger dataset covering more ad hoc parsers. Further studies might investigate the chronological code change of ad hoc parsers. By looking at bug-fix commits, this could give insights into whether ad hoc parsers tend to be buggy or not.

List of Figures

2.1	Boa code snippet	4
2.2	Dependency Graphs	6
3.1	Workflow of detecting ad hoc parsers	10
3.2	Ad hoc parser example with CFG	10
3.3	Workflow of Parser Slicing	11
3.4	Docstring example	12
3.5	String identifying regular expression	12
3.6	Workflow of parser set generation	16
3.7	Comparison of ad hoc parser snippet	19
4.1	Ad hoc parser: File input	24
4.2	Ad hoc parser: Test method Input	24
4.3	Ad hoc parser: Test file	24
4.4	Ad hoc parser: Parsing method	25
4.5	Ad hoc parser: Literal input	25
4.6	Ad hoc parser: Regex using	25
4.7	Distribution of input sources	26
4.8	Parser Positioning	27
4.9	Examples of different input sources	31
4.10	Input Sources: Sankey Diagram	32

List of Tables

2.1	Summary Statistics of the Dataset 2022 February/Python	3
3.1	Return type calculation	13
3.2	Methods divided into categories	15
3.3	Examples of not correctly compiled Boa AST	20
3.4	False positive instances	21
4.1	Comparison of Projects With and Without Parsers	27
4.2	Summary Statistics for Scattering Metrics	28
4.3	Method Chaining Examples	29
4.4	Pearson Correlation Matrix of Parser Implementation Characteristics . .	29
4.5	Examples of different input sources	30
4.6	Function Metrics	33
4.7	Distribution of regex functions across parser sections	35
4.8	Regex Pattern Categories with Usage Statistics	36
4.9	Exception Handling	37

List of Algorithms

3.1 Parser-Detection 17

Bibliography

- [1] Nour AlAbwaini, Amal Aldaàje, Tamara Jaber, Mohammad Abdallah, and Abdelfatah Tamimi. Using program slicing to detect the dead code. In *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, pages 230–233, 2018. doi: 10.1109/CSIT.2018.8486334. URL <https://doi.org/10.1109/CSIT.2018.8486334>.
- [2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. doi: 10.1145/390013.808479. URL <https://doi.org/10.1145/390013.808479>.
- [3] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. How developers use exception handling in java? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 516–519, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2903500. URL <https://doi.org/10.1145/2901739.2903500>.
- [4] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. *Pearson Correlation Coefficient*, pages 1–4. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00296-0. doi: 10.1007/978-3-642-00296-0_5. URL https://doi.org/10.1007/978-3-642-00296-0_5.
- [5] Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca bx Shapiro, and Anna Shubina. Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier. *IEEE Security Privacy*, 12(1):83–87, 2014. doi: 10.1109/MSP.2014.1. URL <https://doi.org/10.1109/MSP.2014.1>.
- [6] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. Chapter 4 - source code analysis and instrumentation. In João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz, editors, *Embedded Computing for High Performance*, pages 99–135. Morgan Kaufmann, Boston, 2017. ISBN 978-0-12-804189-5. doi: <https://doi.org/10.1016/B978-0-12-804189-5.00004-1>. URL <https://www.sciencedirect.com/science/article/pii/B9780128041895000041>.

- [7] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 282–293, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931073. URL <https://doi.org/10.1145/2931037.2931073>.
- [8] Robert Dyer and Jigyasa Chauhan. An Exploratory Study on the Predominant Programming Paradigms in Python Code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, nov 2022. doi: 10.1145/3540250.3549158. URL <https://doi.org/10.1145/3540250.3549158>.
- [9] Robert Dyer, Hoan Nguyen, Hridesh Rajan, and Nguyen Tien. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering (San Francisco, CA) (ICSE’13)*, pages 422–431, 2013. ISBN 978-1-4673-3073-2. doi: 10.1109/ICSE.2013.6606588. URL <https://doi.org/10.1109/ICSE.2013.6606588>.
- [10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3): 319–349, jul 1987. ISSN 0164-0925. doi: 10.1145/24039.24041. URL <https://doi.org/10.1145/24039.24041>.
- [11] Samuel W Flint, Ali M Keshk, Robert Dyer, and Hamid Bagheri. How do developers use type inference: an exploratory study in kotlin. *Empirical Software Engineering*, 30(2):1–29, 2025. doi: 10.1007/s10664-024-10585-y. URL <https://doi.org/10.1007/s10664-024-10585-y>.
- [12] Miles Frantz, Ya Xiao, Tanmoy Sarkar Pias, Na Meng, and Danfeng Yao. Methods and benchmark for detecting cryptographic api misuses in python. *IEEE Transactions on Software Engineering*, 50(5):1118–1129, 2024. doi: 10.1109/TSE.2024.3377182. URL <https://doi.org/10.1109/TSE.2024.3377182>.
- [13] Keith Gallagher and James Lyle. Using program slicing in software maintenance. *Software Engineering, IEEE Transactions on*, 17:751–761, 09 1991. doi: 10.1109/32.83912. URL <https://doi.org/10.1109/32.83912>.
- [14] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. Prevalence of confusing code in software projects: atoms of confusion in the wild. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR ’18, page 281–291, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196432. URL <https://doi.org/10.1145/3196398.3196432>.
- [15] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software

- engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8), December 2024. ISSN 1049-331X. doi: 10.1145/3695988. URL <https://doi.org/10.1145/3695988>.
- [16] Qing Huang, Zhou Zou, Zhenchang Xing, Zhenkang Zuo, Xiwei Xu, and Qinghua Lu. Ai chain on large language model for unsupervised control flow graph generation for statically-typed partial code, 2023. URL <https://doi.org/10.48550/arXiv.2306.00757>.
 - [17] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. Examining programmer practices for locally handling exceptions. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 484–487, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2903497. URL <https://doi.org/10.1145/2901739.2903497>.
 - [18] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 207–218, New York, NY, USA, 1981. Association for Computing Machinery. ISBN 089791029X. doi: 10.1145/567532.567555. URL <https://doi.org/10.1145/567532.567555>.
 - [19] Ivano Malavolta, Kishan Nirghin, Gian Luca Scoccia, Simone Romano, Salvatore Lombardi, Giuseppe Scanniello, and Patricia Lago. Javascript dead code identification, elimination, and empirical assessment. *IEEE Transactions on Software Engineering*, 49(7):3692–3714, 2023. doi: 10.1109/TSE.2023.3267848. URL <https://doi.org/10.1109/TSE.2023.3267848>.
 - [20] Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K. Lahiri, and Isil Dillig. Demystifying loops in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 262–274, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3416626. URL <https://doi.org/10.1145/3324884.3416626>.
 - [21] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837. URL <https://doi.org/10.1109/TSE.1976.233837>.
 - [22] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 45–52, 2016. doi: 10.1109/SecDev.2016.019. URL <https://doi.org/10.1109/SecDev.2016.019>.
 - [23] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in java projects: an empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 500–503, New York, NY, USA, 2016. Association for Computing Machinery. ISBN

9781450341868. doi: 10.1145/2901739.2903499. URL <https://doi.org/10.1145/2901739.2903499>.

- [24] Yun Peng, Yu Zhang, and Mingzhe Hu. An empirical study for common language features used in python projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 24–35, 2021. doi: 10.1109/SANER50967.2021.00012. URL <https://doi.org/10.1109/SANER50967.2021.00012>.
- [25] Michael Schröder and Jürgen Cito. Grammars for free: toward grammar inference for ad hoc parsers. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER '22*, page 41–45, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392242. doi: 10.1145/3510455.3512787. URL <https://doi.org/10.1145/3510455.3512787>.
- [26] Michael Schröder, Marc Goritschnig, and Jürgen Cito. An Exploratory Study of Ad Hoc Parsers in Python, 2023. URL <https://doi.org/10.48550/arXiv.2304.09733>. This is a report registered at MSR 2023.
- [27] Florian Sihler, Lukas Pietzschmann, Raphael Straub, Matthias Tichy, Andor Diera, and Abdelhalim Dahou. On the anatomy of real-world r code for static analysis, 2024. URL <https://doi.org/10.1145/3643991.3644911>.
- [28] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers Security*, 81:123–147, 2019. ISSN 0167-4048. doi: 10.1016/j.cose.2018.11.001. URL <https://doi.org/10.1016/j.cose.2018.11.001>.
- [29] Katherine Underwood and Michael E. Locasto. In search of shotgun parsers in android applications. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 140–155, 2016. doi: 10.1109/SPW.2016.41. URL <https://doi.org/10.1109/SPW.2016.41>.
- [30] Mark Weiser. Program Slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010248. URL <https://doi.org/10.1109/TSE.1984.5010248>.
- [31] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005. ISSN 0163-5948. doi: 10.1145/1050849.1050865. URL <https://doi.org/10.1145/1050849.1050865>.
- [32] Zhaogui Xu, Ju Qian, Lin Chen, Zhifei Chen, and Baowen Xu. Static slicing for python first-class objects. In *2013 13th International Conference on Quality Software*, pages 117–124, 2013. doi: 10.1109/QSIC.2013.50. URL <https://doi.org/10.1109/QSIC.2013.50>.

- [33] Yi Yang, Ana Milanova, and Martin Hirzel. Complex python features in the wild. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 282–293, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528467. URL <https://doi.org/10.1145/3524842.3528467>.