Pyformlang: An Educational Library for Formal Language Manipulation

Julien Romero
julien.romero@mpi-inf.mpg.de
Max Planck Institute for Informatics
Saarbruecken, Germany

ABSTRACT

Formal languages are widely studied, taught and used in computer science. However, only a small part of this domain is brought to a broader audience, and students often have no practical experience in their curriculum. In this tool paper, we introduce Pyformlang, a practical and pedagogical Python library for formal languages. Our library implements the most common algorithms of the domain, accessible by an easy-to-use interface. The code is written exclusively in Python3, with a clear structure, so as to allow students to play and learn with it.

CCS CONCEPTS

• **Theory of computation → Grammars and context-free languages**; **Regular languages**; • **Applied computing → Education**; • **Software and its engineering → Software libraries and repositories**.

KEYWORDS

formal languages, library, python, education

ACM Reference Format:
Julien Romero. 2021. Pyformlang: An Educational Library for Formal Language Manipulation. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432464

1 INTRODUCTION

The range of applications of formal languages is infinite: They include information extraction with regular expressions for knowledge base construction as in Yago [24] or Quasimodo [21], code parsing with context-free grammars, video game artificial intelligence using finite-state automata [16], or even query rewriting using complex manipulations of context-free grammars and regular expressions as in [20].

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432464

Formal languages are part of most computer science programs, and many students learn about it early in their studies. For example, in France, this subject is the first topic taught in "classe préparatoire"[1] (two first years of university for engineer students).

Although formal languages are quite old, teaching this discipline is still a complicated task. The reason is that traditional teaching methods involve much mathematics: Students must learn the proofs and the algorithms to be able to reproduce them on paper. Therefore, they generally have issues visualising the concept introduced, and some authors suggested that more interactive methods should be used [14, 15]. Some textbooks [11] also contain many exercises with no solution. An automatic tool to manipulate formal language can come in handy when the teacher does not have enough time to go through all exercises, or when the student is self-taught. Moreover, more interactive and easy-to-use methods to manipulate formal languages open the field of project-based learning: Having a tool that is close to the course material and that is of practical usage would help students apply the concepts learnt in class.

Besides, when teaching formal languages applications, one generally gives a brief overview of the core concepts without entering into details. However, some students might not have learnt about formal languages before. So, to better understand the more advanced and specialised algorithms, specific tools that can decompose them are very useful. For example, regular expressions are often used in Information Extraction. Their equivalent representation using finite-state automata can be more comfortable in some situation, as it is more visual. However, the transition between the two representations is quite rare in libraries. In particular, it is not possible to obtain directly a finite-state automaton from a Python regular expression (mainly because Python regular expressions are strictly more expressive than regular expressions used in formal language theory).

Python has become one of the most popular programming languages over the past few years (https://insights.stackoverflow.com/survey/2019), and particularly among students. However, there are few tools to manipulate formal languages with Python.

In this paper, we present Pyformlang, a Python3 library for manipulating formal languages. In Section 2, we will discuss other libraries for manipulating formal languages. Then, in Section 3, we introduce Pyformlang, its components and how to use them.

2 PREVIOUS WORK

We survey libraries that allow students to learn formal languages.

The most used part of formal languages is surely regular expressions. Most programming languages implement them natively.

[1] https://eduscol.education.fr/sti/textes/programme-option-informatique-cpge-mp

Text processing is the primary use case. In Python, the *re* library provides many possibilities to use regular expressions. However, it is restricted to regular expressions, and therefore it lacks more advanced formal language manipulation tools. For example, finite-state automata are missing from the standard library. Thus, one cannot transform a Python regular expression into its equivalent finite-state automaton (which does not always exists, as Python regular expressions are strictly more general than formal language theory ones). More operations like concatenation, union or Kleene stars are not available at the level of regular expressions.

For parsing natural language, the most common tools are context-free grammars. Some libraries are specialised in language parsing, such as Lark (https://pypi.org/project/lark-parser/) or more generally NLTK [13]. However, they are focused on parsing natural language, make no link with the rest of the theory, and often rely on an external file format (.lark for instance). As an example, none of the existing libraries allows the intersection between a context-free grammar and a regular expression or a finite-state automaton (which is also a context-free language).

FAdo [17] is a library that focuses mainly on finite-state automata, with few other functionalities. The language of FAdo is Python2, a deprecated language (https://www.python.org/dev/peps/pep-0373/) which is not compatible with Python3. The library implements regular expressions, deterministic, non-deterministic and general automata and the transformations to go from one to the other. One can also perform standard operations such as union, intersection, concatenation or emptiness. However, FAdo does not go further than finite-state automata. For example, there is no link with context-free grammars.

JFLAP [18] is a tool written in Java to manipulate formal languages and it is the most advanced tool available. It implements regular expressions, deterministic and non-deterministic automata, context-free grammars, push-down automata, and Turing Machines. A user can also perform transformations between these structures. Besides, it provides roughly the same basic operations as FAdo for manipulating finite-state automata. However, many advanced operations are missing: For example, one cannot compute the intersection of context-free grammar and a regular expression. Furthermore, the usage of Java can be a limiting factor for fresh computer science students, and it makes prototyping harder. Finally, JFLAP is a graphical tool that does not provide an easy-to-access programmatic API for students. Therefore, it cannot be used outside its graphical interface and, in particular, it cannot be easily integrated into other applications.

Forlan [22, 23] is a toolkit designed to run experiments with formal languages. It is written in Standard ML, a barely used language. Forlan comes with JForlan, a graphical editor for automata and trees.

OpenFST [3] is a library to manipulate weighted finite-state transducers written in C++. Some Python wrappers were also implemented. OpenFST does not support operations beyond the realm of transducers.

Vaucanson [6] is an open-source C++ platform dedicated to the manipulation of finite weighted automata. The usage of C++ makes it very fast. It has bindings for Python3, but it cannot be installed directly from PyPi, making its usage more complex. The content on automata and transducers is very advanced (going beyond a first class on formal languages), but there is no link with non-regular languages.

Some other tools exist to manipulate formal languages but they are either no longer available or very specialised. For example, Covenant [7] specialises in the intersection of context-free grammars (which are not necessary context-free) but is no longer accessible.

3 PYFORMLANG

Pyformlang is a Python3 library specialised in formal language manipulation. It is freely available on PyPI (https://pypi.org/project/pyformlang/) and the source code is on Github (https://github.com/Auns iels/pyformlang). The full documentation can also be accessed on ReadTheDocs (https://pyformlang.readthedocs.io).

We chose to implement Pyformlang in Python3 in order to make it easily accessible. The implementation of each algorithm follows the one presented in the relevant textbooks. This way, a student can easily follow every detail of the lectures in Pyformlang. Unless otherwise mentioned, we used Hopcroft [11] as the source of most algorithms, since it is the most popular textbook for formal languages.

Pyformlang is composed of six modules. In Section 3.1, we present how regular expressions are handled. Then, we introduced their equivalent representation using finite state automata in Section 3.2. Next, we present finite-state transducers in Section 3.3. In Section 3.4, we take a look to context-free grammars before treating the equivalent class of push-down automata in Section 3.5. Finally, we talk about indexed grammars in Section 3.6.

3.1 Regular Expressions

All courses on formal languages generally start with regular expressions. Pyformlang implements the operators of textbooks, which deviate slightly from the operators in Python.

- The concatenation can be represented either by a space or a dot (.)
- The union is represented either by | or +
- The Kleene star is represented by *
- The epsilon symbol can either be *epsilon* or $

It is also possible to use parentheses. All symbols except the space, ., |, +, *, (,), *epsilon* and $ can be part of the alphabet. All other common regex operators (such as [])) are syntactic sugar that can be reduced to our operators.

We deviate in one important point from the standard implementation of regular expressions. Usually, the alphabet consists of all single characters (minus special ones), thus creating a concatenation when encountering consecutive characters. In Pyformlang, we consider that consecutive characters are a single symbol. We wanted to have a more general approach not bound to text processing. As an example, in normal Python, *na** will be interpreted as all words starting by *n* and ending by an indeterminate amount of *a* (like *naaa*). In Pyformlang it represents all strings composed of zero or more repetitions of the letter *na* (like *nanananana*). This deviation allows expressing a wider variety of symbols in our alphabet, and thus more words (i.e. combinations of alphabet symbols) in a regular language. Still, our library also allows the standard semantics though a wrapper of Python regular expressions.

As most users are familiar with the implementation in the Python standard library (which follows Perl's standard), our library can transform a Python regular expression into a regular expression compatible with our implementation. This transformation gives access to a more diversified function set for manipulating regular expressions and also makes it possible to have additional representations (such as finite automata, see Section 3.2) that are missing from the standard library. The transformation modifies the initial regular expression to reduce it to the fundamental operators presented above. Then, our parser can take the transformed regular expression and turn it into the internal representation.

As is the case in most systems, we used a tree structure to represent the regular expressions, where each node is an operator, and each leaf is a symbol in the alphabet. This tree structure is often presented to explain how regular expression work and Pyformlang is able to print it.

Pyformlang contains the fundamental transformations on regular expressions, which produce again regular expressions: concatenation, Kleene star and unions. The transformation of a regular expression into an equivalent non-deterministic automaton with epsilon transitions gives access to additional operations.

Example 3.1. We show here an example of how to use the regular expressions in our library:

```
from pyformlang.regular_expression import Regex

regex = Regex("abc|d")
# Check if the symbol "abc" is accepted
regex.accepts(["abc"])  # True
# Check if the word composed of the symbols
# "a", "b" and "c" is accepted
regex.accepts(["a", "b", "c"])  # False
# Check if the symbol "d" is accepted
regex.accepts(["d"])  # True

regex1 = Regex("a b")
regex_concat = regex.concatenate(regex1)
regex_concat.accepts(["d", "a", "b"])  # True

print(regex_concat.get_tree_str())
# Operator(Concatenation)
#   Operator(Union)
#     Symbol(abc)
#     Symbol(d)
#   Operator(Concatenation)
#     Symbol(a)
#     Symbol(b)

# Give the equivalent finite-state automaton
equivalent_enfa = regex_concat.to_epsilon_nfa()
equivalent_enfa.accepts(["d", "a", "b"])  # True

# Python regular expressions wrapper
from pyformlang.regular_expression import PythonRegex

p_regex = PythonRegex("a+[cd]")
p_regex.accepts(["a", "a", "d"])  # True
# As the alphabet is composed of single characters, one
# could also write
p_regex.accepts("aad")  # True
p_regex.accepts(["d"])  # False
```

Listing 1: Regular Expression Example

3.2 Finite-State Automata

Finite-state automata are an equivalent representation of regular expressions. They have the advantage of being more visual and many algorithms (such as the intersection with a context-free grammar) rely on them directly. Pyformlang contains the three main types of non-weighted finite-state automata: deterministic, non-deterministic and non-deterministic with epsilon transitions.

All possible transformations are implemented. First, it is possible to transform any non-deterministic finite-state automaton with epsilon transitions into a non-deterministic finite without epsilon transitions. Then, it is possible to turn a non-deterministic finite automaton into a deterministic one. Besides, as finite-state automata and regular expressions are equivalent, we offer the possibility to go from one to the other.

We implemented the fundamental operations on automata:

- Get the complementary, the reverse and the Kleene star of an automaton.
- Get the difference, the concatenation and the intersection between two automata.
- Check if an automaton is deterministic.
- Check if an automaton is acyclic.
- Check if an automaton produces the empty language.
- Minimize an automaton using the Hopcroft's minimization algorithm for deterministic automata [10, 25].
- Check if two automata are equivalent.

Internally, the automaton is implemented using dictionaries and these dictionaries are accessible to the user who wants to manipulate them.

An advantage of finite-state automata is that they offer a nice visual representation (at least for smaller automata). Fundamentally, a finite-state automaton can be represented as a directed graph with two kinds of special nodes: Starting nodes and final nodes. We offer the possibility to turn a finite automaton into a NetworkX [9] MultiDiGraph. This library is the most popular one for graph manipulation in Python. Besides, the user can also save the finite-automaton into a dot file (https://graphviz.org/doc/info/lang.html) to print it in a GUI. An example is given in Figure 1.

Example 3.2. We show here an example of how to use the finite automata in our library:

```
from pyformlang.finite_automaton import EpsilonNFA

enfa = EpsilonNFA()
enfa.add_transitions(
    [(0, "abc", 1), (0, "d", 1), (0, "epsilon", 2)])
enfa.add_start_state(0)
enfa.add_final_state(1)
enfa.is_deterministic()  # False

dfa = enfa.to_deterministic()
dfa.is_deterministic()  # True
dfa.is_equivalent_to(enfa)  # True

enfa.is_acyclic()  # True
enfa.is_empty()  # False
enfa.accepts(["abc", "epsilon"])  # True
enfa.accepts(["epsilon"])  # False

enfa2 = EpsilonNFA()
enfa2.add_transition(0, "d", 1)
```

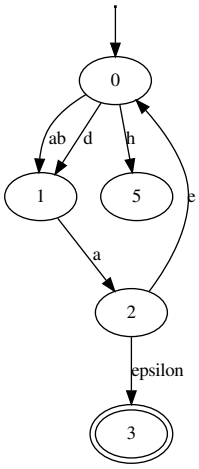

Figure 1: Visualisation of a Finite-State Automaton

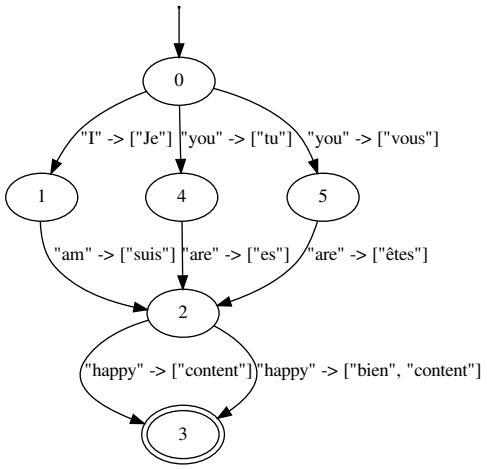

Figure 2: A Finite-State Transducer

```
enfa2.add_final_state(1)
enfa2.add_start_state(0)
enfa_inter = enfa.get_intersection(enfa2)
enfa_inter.accepts(["abc"])  # False
enfa_inter.accepts(["d"])  # True
```

Listing 2: Finite-State Automata Example

3.3 Finite-State Transducer

Finite-state transducers look like finite-state automata. The main difference is that they take as input a word and translate it into another word. Finite-state transducers are more rarely introduced in the first class on formal languages, but rather in more advanced lectures such as natural language processing [12]. Pyformlang implements non-weighted finite-state transducers and operators on them: the concatenation, the union and the Kleene star. Besides, we offer an intersection function to intersect a finite-state transducer with an indexed grammar (see Section 3.6).

Just like finite-state automata, it is possible to turn a finite-state transducer into a NetworkX graph and to save it into a dot file. Figure 2 shows the finite-state transducer used in Example 3.3.

Example 3.3. We show here an example of how to use the finite-state transducers in our library:

```
from pyformlang.fst import FST

fst = FST()
fst.add_transitions(
    [(0, "I", 1, ["Je"]), (1, "am", 2, ["suis"]),
    (2, "happy", 3, ["content"]),
    (2, "happy", 3, ["bien", "content"]),
    (0, "you", 4, ["tu"]), (4, "are", 2, ["es"]),
    (0, "you", 5, ["vous"]), (5, "are", 2, ["etes"])])
fst.add_start_state(0)
fst.add_final_state(3)
list(fst.translate(["you", "are", "happy"]))
# [['vous', 'etes', 'bien', 'content'],
#  ['vous', 'etes', 'content'],
#  ['tu', 'es', 'bien', 'content'],
#  ['tu', 'es', 'content']]
```

Listing 3: Finite-State Transducer Example

3.4 Context-Free Grammars

Context-free languages are strictly more powerful than regular languages, i.e. all regular expressions can be expressed as a context-free grammar, but the inverse is not true. Pyformlang implements context-free grammars and the essential operations on them. One can construct a context-free grammar in two different ways. The first one consists of using internal representation objects to represent variables, terminals and production rules. This initialisation process can be quite wordy, and in most cases, it is not necessary. However, it is close to textbooks representations and allows a better understanding of context-free grammars. Besides, it is easier to use in a computer program. The other way, used by most libraries, uses a string representation of the context-free grammar. Example 3.4 shows how this construction looks.

As many algorithms use the Chomsky Normal Form (CNF), our library offers the possibility to transform a context-free grammar into its CNF.

Context-free grammars are equivalent to push-down automata. So, our library allows transforming a context-free grammar into its equivalent push-down automaton accepting by empty stack.

Our context-free grammar implementation also contains several operations linked to the closure properties: The concatenation, the union, the (positive) closure, the reversal and the substitution of terminals by another context-free grammar. In general, the intersection of two context-free grammars is not a context-free grammar, except if one of them is a regular expression or a finite automaton. Hopcroft [11] presents an algorithm that first transforms the grammar into an equivalent push-down automaton accepting by empty stack, and then into an equivalent push-down automaton accepting by final state. Next, it performs the intersection with the regular expression (or finite automaton) and transforms the resulting push-down automaton back into a context-free grammar (by transiting through a push-down automaton accepting by empty stack). However, this solution is costly, and we preferred the solution given by Bar-Hillel [4, 5], which does not require push-down automata and is much more efficient. The algorithm of Bar-Hillel

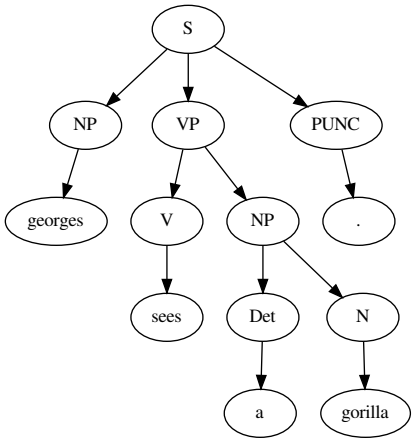

Figure 3: Parsing Tree

directly creates new non-terminals and the transitions between them from the context-free grammar and the regular expression.

We implemented various other operations, such as:

- Checking if a given context-free grammar produces a word
- Checking if a word is part of the language generated by the grammar
- Generate words in the grammar (through a generator as the language associated with the grammar is not necessarily finite)
- Checking whether a grammar is finite or not

Finally, we also added parsers such as the recursive decent parser (that might not terminate for some left-recursive grammars) and the LL(1) parser [2] (that only works for some types of grammars), which allows us to obtain parsing trees and derivations. Figure 3 shows an example of a parsing tree from the grammar defined in Example 3.4.

Example 3.4. We show here an example of how to use context-free grammars in our library:

```
from pyformlang.cfg import CFG
from pyformlang.cfg.llone_parser import LLOneParser
from pyformlang.regular_expression import Regex

cfg = CFG.from_text("""
    S -> NP VP PUNC
    PUNC -> . | !
    VP -> V NP
    V -> buys | touches | sees
    NP -> georges | jacques | leo | Det N
    Det -> a | an | the
    N -> gorilla | dog | carrots""")
regex = Regex("georges touches (a|an) (dog|gorilla) !")

cfg_inter = cfg.intersection(regex)
cfg_inter.is_empty()  # False
cfg_inter.is_finite()  # True
cfg_inter.contains(["georges", "sees",
                    "a", "gorilla", "."])  # False
cfg_inter.contains(["georges", "touches",
                    "a", "gorilla", "!"])  # True

```

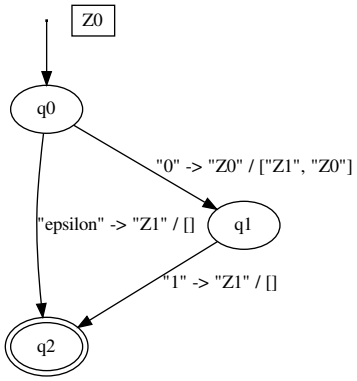

Figure 4: Visualisation of a Push-Down Automaton

```
cfg_inter.is_normal_form()  # False
cnf = cfg.to_normal_form()
cnf.is_normal_form()  # True

llone_parser = LLOneParser(cfg)
parse_tree = llone_parser.get_llone_parse_tree(
    ["georges", "sees", "a", "gorilla", "."])
parse_tree.get_leftmost_derivation(),
# [[Variable("S")],
#  [Variable("NP"), Variable("VP"), Variable("PUNC")],
#  ...,
#  [Terminal("georges"), Terminal("sees"),
#   Terminal("a"), Terminal("gorilla"), Terminal(".")]]
```

Listing 4: Context-Free Grammar Example

3.5 Push-Down Automata

Push-down automata are an equivalent representation of context-free grammars. They have a more visual representation as they look like a finite-state automaton with a stack. Pyformlang implements push-down automata accepting by final state or by empty stack. It also adds the possibility to go from one acceptance to the other. As context-free grammars are equivalent to push-down automata accepting by empty stack, we implemented the transformation to go from one to the other. Push-down automata have the same closure properties as context-free grammars. We implemented the intersection with a regular language using the native algorithm in Hopcroft [11].

Just like finite-state automata and finite-state transducers, push-down automata can be visualised as a graph with NetworkX. Figure 4 shows the push-down automaton used in Example 3.5.

Example 3.5. We show here an example of how to use the push-down automata in our library:

```
from pyformlang.pda import PDA

pda = PDA()
pda.add_transitions(
    [
        ("q0", "0", "Z0", "q1", ("Z1", "Z0")),
        ("q1", "1", "Z1", "q2", []),
        ("q0", "epsilon", "Z1", "q2", [])
    ]
)
```

```
pda.set_start_state("q0")
pda.set_start_stack_symbol("Z0")
pda.add_final_state("q2")

pda_final_state = pda.to_final_state()
cfg = pda.to_empty_stack().to_cfg()
cfg.contains(["0", "1"])  # True
```

Listing 5: Context-Free Grammar Example

3.6 Indexed Grammar

Aho [1] introduced indexed grammars, and they are an excellent example of non-context-free grammars. For instance, they have application in natural language processing [8]. To the best of our knowledge, no library provides an implementation for this class of grammars. As indexed grammars are less constrained, they do not allow as many operations as context-free grammars: They can represent more languages but the languages are harder to manipulate. Still, Pyformlang implements essential functions such as checking if the grammar is empty or intersecting with a regular expression or a finite automaton. Indeed, indexed grammars are stable by this last operation. However, the complexity of the operations is in general exponential, so we observe long computation times.

As indexed grammars are not well documented, we give here more explanations about how they work. Indexed grammars [1] include the context-free grammars, but are strictly less expressive than context-sensitive grammars.

They generate the class of indexed languages, which contains all context-free languages. A nice feature of this class of languages is that it conserves closure properties and decidability results. In addition to the set of terminals and non-terminals from the context-free grammars, the indexed grammars introduce the set of *index symbols*. Following [11], an indexed grammar is a 5-tuple (N, T, I, P, S) where:

- N is a set of variables or non-terminal symbols
- T is a set of terminal symbols
- I is a set of index symbols
- $S \in N$ is the start symbol, and
- P is a finite set of productions.

Each production in P takes one of the following forms:

$$A[\sigma] \to a \qquad \text{(end rule)}$$
$$A[\sigma] \to B[\sigma]C[\sigma] \qquad \text{(duplication rule)}$$
$$A[\sigma] \to B[f\sigma] \qquad \text{(production rule)}$$
$$A[f\sigma] \to B[\sigma] \qquad \text{(consumption rule)}$$

Here, A, B, and C are non-terminals, a is a terminal, and f is an index symbol. The part in brackets $[\ldots]$ is the so-called stack. The left-most symbol is the top of the stack. σ is a special character that stands for the rest of the stack. For example, the duplication rule states that a non-terminal with a certain stack gives rise to two other non-terminals that each carry the same symbols on the stack.

Example 3.6. We show here an example of how to use indexed grammars in our library:

```
from pyformlang.indexed_grammar import Rules,
    ConsumptionRule, EndRule, ProductionRule,
    DuplicationRule, IndexedGrammar
from pyformlang.regular_expression import Regex

# Generate a single word: a b
all_rules = [ProductionRule("S", "D", "f"),
             DuplicationRule("D", "A", "B"),
             ConsumptionRule("f", "A", "Afinal"),
             ConsumptionRule("f", "B", "Bfinal"),
             EndRule("Afinal", "a"),
             EndRule("Bfinal", "b")]
indexed_grammar = IndexedGrammar(Rules(all_rules))
indexed_grammar.is_empty()  # False
i_inter = indexed_grammar.intersection(Regex("a.b"))
i_inter.is_empty()  # False
```

Listing 6: Indexed-Grammar Example

4 CONCLUSION

In this tool paper, we introduced Pyformlang, a modern Python3 library to manipulate formal languages, especially tailored for practical and pedagogical purposes. It contains the main functionalities to manipulate regular expressions, finite state automata, finite-state transducers, context-free grammars, push-down automata and indexed grammars. Pyformlang can easily be integrated into an automatic rating system such as VPL [19] in Moodle thanks to *pip*. The code is fully documented and is open-sourced on GitHub (https://github.com/Aunsiels/pyformlang).

The next steps for Pyformlang will be centred around two directions. The first one is to keep developing the core content, i.e. adding advanced functionalities, algorithms and data structures. The second direction is the creation of advanced visualisation tools around the library, like it can be seen in other tools.

We hope that our library will help students, lecturers and practitioners to discover and exploit the beauty of formal languages.

Acknowledgements. Partially supported by the grants ANR-16-CE23-0007-01 ("DICOS") and ANR-18-CE23-0003-02 ("CQFD").

We thank Fabian Suchanek and Nicoleta Preda for their feedback, and the reviewers for their constructive reviews.

REFERENCES

[1] Alfred V Aho. 1968. Indexed grammars—an extension of context-free grammars. *Journal of the ACM (JACM)* 15, 4 (1968), 647–671.
[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.
[3] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *International Conference on Implementation and Application of Automata*. Springer, 11–23.
[4] Y. Ba-Hillel, M. Prles, and E. Shamir. 1965. On formal properties of simple phrase structure grammars. Z. Phonetik, Sprachwissen. Komm. 15 (I961), 143-172. *Y. Bar-Hillel, Language and Information, Addison-Wesley, Reading, Mass* (1965), 116–150.
[5] Richard Beigel and William Gasarch. .. A Proof that the intersection of a context-free language and a regular language is Context-Free Which Does Not use push-down automata. *http://www.cs.umd.edu/~gasarch/BLOGPAPERS/cfg.pdf* (.).
[6] Akim Demaille, Alexandre Duret-Lutz, Sylvain Lombardy, and Jacques Sakarovitch. 2013. Implementation Concepts in Vaucanson 2. In *Proceedings of Implementation and Application of Automata, 18th International Conference (CIAA'13) (Lecture Notes in Computer Science, Vol. 7982)*, Stavros Konstantinidis (Ed.). Springer, Halifax, NS, Canada, 122–133. https://doi.org/10.1007/978-3-642-39274-0_12
[7] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. 2015. A tool for intersecting context-free grammars and its applications. In *NASA Formal Methods Symposium*. Springer, 422–428.
[8] Gerald Gazdar. 1988. Applicability of indexed grammars to natural languages. In *Natural language parsing and linguistic theories*. Springer, 69–94.

[9] Aric Hagberg, Dan Schult, Pieter Swart, D Conway, L Séguin-Charbonneau, C Ellison, B Edwards, and J Torrents. 2013. Networkx. *URL http://networkx. github. io/index. html* (2013).
[10] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*. Elsevier, 189–196.
[11] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.
[12] Dan Jurafsky. 2000. *Speech & language processing*. Pearson Education India.
[13] Edward Loper and Steven Bird. 2002. NLTK: the natural language toolkit. *arXiv preprint cs/0205028* (2002).
[14] Mostafa Mohammed, Clifford A. Shaffer, and Susan H. Rodger. 2019. Using Interactive Visualization and Programmed Instruction to Teach Formal Languages. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 1263. https://doi.org/10.1145/3287324.3293795
[15] Mostafa Kamel Osman Mohammed. 2020. Teaching Formal Languages through Visualizations, Simulators, Auto-Graded Exercises, and Programmed Instruction. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1429. https://doi.org/10.1145/3328778.3372711
[16] Steven Rabin. 2015. *Game AI pro 2: collected wisdom of game AI professionals*. AK Peters/CRC Press.
[17] Rogério Reis and Nelma Moreira. 2002. FAdo: tools for finite automata and regular expressions manipulation. *https://www.dcc.fc.up.pt/ nam/publica/dcc-2002-2.pdf* (2002).
[18] Susan H Rodger and Thomas W Finley. 2006. *JFLAP: an interactive formal languages and automata package*. Jones & Bartlett Learning.
[19] Juan Carlos Rodríguez-del Pino, Enrique Rubio Royo, and Zenón Hernández Figueroa. 2012. A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. (2012).
[20] Julien Romero, Nicoleta Preda, Antoine Amarilli, and Fabian Suchanek. 2020. Equivalent Rewritings on Path Views with Binding Patterns. arXiv:2003.07316 [cs.DB] Extended version with proofs. https://arxiv.org/abs/2003.07316.
[21] Julien Romero, Simon Razniewski, Koninika Pal, Jeff Z. Pan, Archit Sakhadeo, and Gerhard Weikum. 2019. Commonsense Properties from Query Logs and Question Answering Forums. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management - CIKM '19* (2019). https://doi.org/10.1145/3357384.3357955
[22] Alley Stoughton. 2008. Experimenting with formal languages using Forlan. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education*. 41–50.
[23] Alley Stoughton. 2016. Formal Language Theory: Integrating Experimentation and Proof.
[24] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*. 697–706.
[25] Hang Zhou. 2009. Implementation of the Hopcroft's Algorithm. *https://www.irif.fr/ carton//Enseignement/Complexite/ENS/Redaction/2009-2010/hang.zhou.pdf* (2009).