pyformlang Documentation

Release 0.1.3

Julien Romero

Jul 27, 2021

CONTENTS:

CHAPTER

ONE

INSTALLATION

1.1 Stable release

To install Pyformlang, run this command in your terminal:

```
$ pip3 install pyformlang
```

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for Pyformlang can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/Aunsiels/pyformlang
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

TWO

USAGE

2.1 Regular expression

Pyformlang's Regex class implements the operators of textbooks, which deviate slightly from the operators in Python. For a representation closer to Python one, please use the PythonRegex class.

- 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 the previous operators. The class PythonRegex implements some of them natively. Another main difference is that the alphabet is not reduced to single characters as it is the case in Python. For example, *python* is a single symbol in Pyformlang, whereas it is the concatenation of six symbols in regular Python.

All special characters except epsilon can be escaped with a backslash (double backslash \\ in strings).

```
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)
```

(continues on next page)

(continued from previous page)

```
#   Symbol(a)
#   Symbol(b)

# Give the equivalent finite-state automaton
regex_concat.to_epsilon_nfa()

# 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
```

2.2 Finite Automata

Pyformlang contains several finite automata, all of them being equivalent in the languages they can describe. In general, the states have to be represented by a *pyformlang.finite_automaton.State* object and the symbols by a *pyformlang.finite_automaton.Symbol*. When the class is not ambiguous, raw values can also be used. In addition, epsilon transitions are elements of the class: *pyformlang.finite_automaton.Epsilon*.

The finite-state automata are compatible with NetworkX. The function *to_networkx* gives a MultiDiGraph representing the automaton. Besides, the function *write_as_dot* allows to save the automaton into a dot file, from which one can get a visual representation.

2.2.1 Deterministic Automata

These represent deterministic automata, i.e. with only one possible next state possible at each stage and no epsilon transitions. Here, we give an example with explicit States and Symbols.

```
from pyformlang.finite_automaton import DeterministicFiniteAutomaton
from pyformlang.finite_automaton import State
from pyformlang.finite_automaton import Symbol

# Declaration of the DFA
dfa = DeterministicFiniteAutomaton()

# Creation of the states
state0 = State(0)
state1 = State(1)
state2 = State(2)
state3 = State(3)

# Creation of the symbols
symb_a = Symbol("a")
symb_b = Symbol("b")
symb_c = Symbol("c")
symb_d = Symbol("d")
```

(continues on next page)

(continued from previous page)

```
# Add a start state
dfa.add_start_state(state0)

# Add two final states
dfa.add_final_state(state2)
dfa.add_final_state(state3)

# Create transitions
dfa.add_transition(state0, symb_a, state1)
dfa.add_transition(state1, symb_b, state1)
dfa.add_transition(state1, symb_c, state2)
dfa.add_transition(state1, symb_d, state3)

# Check if a word is accepted
dfa.accepts([symb_a, symb_b, symb_c])
```

2.2.2 Non Deterministic Automata

The representation of non deterministic automata, i.e. automata with possibly several next states at each stage but no epsilon transitions. Here, we give an example with explicit States and Symbols.

```
from pyformlang.finite_automaton import NondeterministicFiniteAutomaton
from pyformlang.finite_automaton import State
from pyformlang.finite_automaton import Symbol

# Definition of the NFA
nfa = NondeterministicFiniteAutomaton()

# Declare the states
state0 = State(0)
state1 = State(1)
state2 = State(2)
state3 = State(3)
state4 = State(4)

# Declare the symbols
symb_a = Symbol("a")
symb_b = Symbol("b")
symb_c = Symbol("c")
symb_d = Symbol("d")

# Add a start state
nfa.add_start_state(state0)
# Add a final state
nfa.add_final_state(state4)
nfa.add_final_state(state3)
# Add the transitions
nfa.add_transition(state0, symb_a, state1)
nfa.add_transition(state1, symb_b, state1)
nfa.add_transition(state1, symb_c, state2)
```

(continues on next page)

(continued from previous page)

```
nfa.add_transition(state1, symb_d, state3)
nfa.add_transition(state1, symb_c, state4)
nfa.add_transition(state1, symb_b, state4)

# Check if a word is accepted
nfa.accepts([symb_a, symb_b, symb_c])

# Check if a NFA is deterministic
nfa.is_deterministic() # False

# Get the equivalent deterministic automaton
dfa = nfa.to_deterministic()
```

2.2.3 Epsilon Non Deterministic Automata

It represents a non deterministic automaton where epsilon transitions are allowed. First, we give an example with explicit States and Symbols.

```
from pyformlang.finite_automaton import EpsilonNFA, State, Symbol, Epsilon

# Declaration of the symbols and the states
epsilon = Epsilon()
plus = Symbol("+")
minus = Symbol("-")
point = Symbol(".")
digits = [Symbol(x) for x in range(10)]
states = [State("q" + str(x)) for x in range(6)]

# Creation of the Epsilon NFA
enfa = EpsilonNFA()
enfa.add_start_state(states[0])
enfa.add_final_state(states[5])
enfa.add_transition(states[0], epsilon, states[1])
enfa.add_transition(states[0], plus, states[1])
enfa.add_transition(states[0], minus, states[1])
for digit in digits:
    enfa.add_transition(states[1], digit, states[1])
    enfa.add_transition(states[1], digit, states[4])
    enfa.add_transition(states[2], digit, states[3])
    enfa.add_transition(states[3], digit, states[3])
enfa.add_transition(states[1], point, states[2])
enfa.add_transition(states[4], point, states[3])
enfa.add_transition(states[3], epsilon, states[5])

# Checks if a word is accepted
enfa.accepts([plus, digits[1], point, digits[9]])
```

Here, we present an example where the State and Symbols are implicit. The construction becomes much simpler.

```
from pyformlang.finite_automaton import EpsilonNFA
```

(continues on next page)

(continued from previous page)

```
# Initialize the automaton
enfa = EpsilonNFA()

# We can add multiple transitions at once
enfa.add_transitions(
   [(0, "abc", 1), (0, "d", 1), (0, "epsilon", 2)])

# We add start and final states
enfa.add_start_state(0)
enfa.add_final_state(1)

# We check if the automaton is deterministic
enfa.is_deterministic()  # False

# We make the automaton deterministic
dfa = enfa.to_deterministic()

# We can check that the new automaton is deterministic and equivalent to
# the original one
dfa.is_deterministic()  # True
dfa.is_equivalent_to(enfa)  # True

# We check if the automaton is acyclic
enfa.is_acyclic()  # True

# We check if the automaton generates the empty language
enfa.is_empty()  # False

# We check if some words are accepted. A word must be an iterable of
# symbols.
enfa.accepts(["abc", "epsilon"])  # True
enfa.accepts(["epsilon"])  # False

# We create a new automaton
enfa2 = EpsilonNFA()
enfa2.add_transition(0, "d", 1)
enfa2.add_final_state(1)
enfa2.add_start_state(0)

# We take the intersection of the two automata
enfa_inter = enfa.get_intersection(enfa2)
enfa_inter.accepts(["abc"])  # False
enfa_inter.accepts(["d"])  # True
```

2.3 Regex and Finite Automaton

As regex and finite automaton are equivalent, one can turn one into the other.

```
from pyformlang.finite_automaton import EpsilonNFA, State, Symbol, Epsilon

enfa = EpsilonNFA()
state0 = State(0)
state1 = State(1)
symb_a = Symbol("0")
symb_b = Symbol("1")
enfa.add_start_state(state0)
enfa.add_final_state(state1)
enfa.add_transition(state0, symb_a, state0)
enfa.add_transition(state1, symb_b, state0)
enfa.add_transition(state1, symb_b, state1)

# Turn a finite automaton into a regex...
regex = enfa.to_regex()
# And turn it back into an epsilon non deterministic automaton
enfa2 = regex.to_epsilon_nfa()
```

2.4 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. 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.

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.

```
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']]
```

2.5 Context-Free Grammar

We represent here context-free grammars. Like finite automata, one needs to use the classes *pyformlang.cfg.Variable* and *pyformlang.cfg.Terminal* to represent variables and terminals. The productions need to be represented as *pyformlang.cfg.Production*. In addition, epsilon terminals are members of *pyformlang.cfg.Epsilon*. This representation removes ambiguities, but is quite verbose.

```
from pyformlang.cfg import Production, Variable, Terminal, CFG, Epsilon

# Creation of variables
var_useless = Variable("USELESS")
var_S = Variable("S")
var_B = Variable("B")

# Creation of terminals
ter_a = Terminal("a")
ter_b = Terminal("b")
ter_c = Terminal("c")

# Creation of productions
p0 = Production(var_S, [ter_a, var_S, var_B])
p1 = Production(var_useless, [ter_a, var_S, var_B])
p2 = Production(var_S, [var_useless])
p4 = Production(var_B, [ter_b])
p5 = Production(var_useless, [])

# Creation of the CFG
cfg = CFG({var_useless, var_S}, {ter_a, ter_b}, var_S, {p0, p1, p2, p4, p5})

# Check for containment
cfg.contains([Epsilon()])
cfg.contains([ter_a, ter_b])
```

Pyformlang also includes an easier-to-use representation, similar to what is done in other libraries such as NLTK. The variables are represented by a string starting with a capital letter. All other strings are terminals. Note also that the variables and the terminals cannot contain spaces.

```
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
```

(continues on next page)

(continued from previous page)

```
cfg_inter.is_finite()  # True
cfg_inter.contains(["georges", "sees",
                    "a", "gorilla", "."])  # False
cfg_inter.contains(["georges", "touches",
                    "a", "gorilla", "!"])  # True

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(".")]]
```

2.6 Push-Down Automata

For a Push-Down Automata, there are there objects: *pyformlang.pda.State* which represents a state, *pyformlang.pda.Symbol* which represents a symbol and *pyformlang.pda.StackSymbol* which represents a stack symbol.

PDA can either accept by final state or by empty stack. Function are provided to transform one kind into the other.

```
from pyformlang.pda import PDA, State, StackSymbol, Symbol, Epsilon

# Declare states
q = State("#STARTTOFINAL#")
q0 = State("q0")

# Declare symbols
e = Symbol("e")
i = Symbol("i")

# Declare stack symbols
Z = StackSymbol("Z")
Z0 = StackSymbol("Z0")

# Create the PDA
pda = PDA(states={q, q0},
          input_symbols={i, e},
          stack_alphabet={Z, Z0},
          start_state=q,
          start_stack_symbol=Z0,
          final_states={q0})

# Add transitions
```

(continues on next page)

(continued from previous page)

```
pda.add_transition(q, i, Z, q, (Z, Z))
pda.add_transition(q, i, Z0, q, (Z, Z0))
pda.add_transition(q, e, Z, q, [])
pda.add_transition(q, Epsilon(), Z0, q0, [])

# Transformation to a PDA accepting by empty stack
pda_empty_stack = pda.to_empty_stack()
# Transformation to a PDA accepting by final state
pda_final_state = pda_empty_stack.to_final_state()
```

As for finite automata, push-down automata can be constructed with implicit stack symbols and states:

```
from pyformlang.pda import PDA

pda = PDA()

# This function allows to add multiple transitions
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
```

2.7 CFG and PDA

As CFG and PDA are equivalent, one can turn one into the other, but needs to be careful about whether the PDA accepts on empty stack and final state. The conversions between CFG and PDA are done when the PDA is accepting by empty stack

```
from pyformlang.cfg import Production, Variable, Terminal, CFG

ter_a = Terminal("a")
ter_b = Terminal("b")
ter_c = Terminal("c")
var_S = Variable("S")
productions = {Production(var_S, [ter_a, var_S, ter_b]),
               Production(var_S, [ter_c])}
cfg = CFG(productions=productions, start_symbol=var_S)

# Convert into a PDA accepting by final state
pda_empty_stack = cfg.to_pda()
```

(continues on next page)

(continued from previous page)

```
# Go to final state
pda_final_state = pda_empty_stack.to_final_state()
# Go back to empty stack, necessary to transform into a CFG
pda_empty_stack = pda_final_state.to_empty_stack()
# Transform the PDA into a CFG
cfg = pda_empty_stack.to_cfg()
```

2.8 Indexed Grammars

Indexed grammars are grammars which have a stack which can be duplicated. In an indexed grammar, rules can take 4 forms (sigma is the stack):

- *EndRule*: This simply turns a Variable into a terminal, for example A[sigma]->a
- *ProductionRule*: We push something on the stack, for example A[sigma]->B[f sigma]
- *ConsumptionRule*: We consume something from the stack, for example A[f sigma] -> C[sigma]
- *DuplicationRule*: We duplicate the stack, for example A[sigma] -> B[sigma] C[sigma]

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

l_rules = []

# Initialization rules
l_rules.append(ProductionRule("S", "Cinit", "end"))
l_rules.append(ProductionRule("Cinit", "C", "b"))
l_rules.append(ConsumptionRule("end", "C", "T"))
l_rules.append(EndRule("T", "epsilon"))

# C[cm sigma] -> cm C[sigma]
l_rules.append(ConsumptionRule("cm", "C", "B0"))
l_rules.append(DuplicationRule("B0", "A0", "C"))
l_rules.append(EndRule("A0", "cm"))

rules = Rules(l_rules)
i_grammar = IndexedGrammar(rules)
self.assertTrue(i_grammar.is_empty())
```

CHAPTER

THREE

MODULES

3.1 Regular Expression

3.1.1 pyformlang.regular_expression

This module deals with regular expression.

By default, this module does not use the standard way to write regular expressions. Please read the documentation of Regex for more information.

Available Classes

Regex A regular expression

PythonRegex A regular expression closer to Python format

MisformedRegexError An error occurring when the input regex is incorrect

exception pyformlang.regular_expression.**MisformedRegexError**(*message: str*, *regex: str*)
Error for misformed regex

class pyformlang.regular_expression.**PythonRegex**(*python_regex*)
Represents a regular expression as used in Python.

It adds the following features to the basic regex:

- Set of characters with [] (no inverse with [^...])
- positive closure +
- . for all printable characters
- ? for optional character/group
- Shortcuts: d, s, w

Parameters **python_regex** (*str*) – The regex represented as a string or a compiled regex (re.compile(...))

Raises *MisformedRegexError* – If the regular expression is misformed.

Examples

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
```

class pyformlang.regular_expression.**Regex**(*regex*)

Represents a regular expression

Pyformlang implements the operators of textbooks, which deviate slightly from the operators in Python. For a representation closer to Python one, please use *PythonRegex*

- 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 the previous operators. Another main difference is that the alphabet is not reduced to single characters as it is the case in Python. For example, "python" is a single symbol in Pyformlang, whereas it is the concatenation of six symbols in regular Python.

All special characters except epsilon can be escaped with a backslash (double backslash in strings).

Parameters **regex** (*str*) – The regex represented as a string

Raises *MisformedRegexError* – If the regular expression is misformed.

Examples

```
>>> 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

```
>>> regex_concat.to_epsilon_nfa()
```

accepts(*word: Iterable[str]*) → bool

Check if a word matches (completely) the regex

Parameters **word** (*iterable of str*) – The word to check

Returns **is_accepted** – Whether the word is recognized or not

Return type bool

Examples

```
>>> regex = Regex("abc|d")
```

Check if the symbol "abc" is accepted

```
>>> regex.accepts(["abc"])
True
```

concatenate(*other:* pyformlang.regular_expression.regex.Regex) → *pyformlang.regular_expression.regex.Regex*

Concatenates a regular expression with an other one

Equivalent to:

```
>>> regex0 + regex1
```

Parameters **other** (*Regex*) – The other regex

Returns **regex** – The concatenation of the two regex

Return type *Regex*

Examples

```
>>> regex0 = Regex("a b")
>>> regex1 = Regex("c")
>>> regex_union = regex0.concatenate(regex1)
>>> regex_union.accepts(["a", "b"])
False
>>> regex_union.accepts(["a", "b", "c"])
True
```

Or equivalently:

```
>>> regex_union = regex0 + regex1
>>> regex_union.accepts(["a", "b", "c"])
True
```

classmethod from_python_regex(*regex*)

Creates a regex from a string using the python way to write it.

Careful: Not everything is implemented, check PythonRegex class documentation for more details.

It is equivalent to calling PythonRegex constructor directly.

Parameters **regex** (*str*) – The regex given as a string or compile regex

Returns **python_regex** – The regex

Return type *PythonRegex*

Examples

```
>>> Regex.from_python_regex("a+[cd]")
```

from_string(*regex_str: str*)

Construct a regex from a string. For internal usage.

Equivalent to the constructor of Regex

Parameters **regex_str** (*str*) – The string representation of the regex

Returns **regex** – The regex

Return type *Regex*

Examples

```
>>> regex.from_string("a b c")
```

, which is equivalent to:

```
>>> Regex("a b c")
```

get_number_operators() → int

Gives the number of operators in the regex

Returns **n_operators** – The number of operators in the regex

Return type int

Examples

```
>>> regex = Regex("a|b*")
>>> regex.get_number_operators()
2
```

The two operators are "|" and "*".

get_number_symbols() → int
Gives the number of symbols in the regex

Returns **n_symbols** – The number of symbols in the regex

Return type int

Examples

```
>>> regex = Regex("a|b*")
>>> regex.get_number_symbols()
2
```

The two symbols are "a" and "b".

get_tree_str(*depth: int = 0*) → str
Get a string representation of the tree behind the regex

Parameters **depth** (*int*) – The current depth, 0 by default

Returns **representation** – The tree representation

Return type str

Examples

```
>>> regex = Regex("abc|d*")
>>> print(regex.get_tree_str())
Operator(Union)
 Symbol(abc)
 Operator(Kleene Star)
  Symbol(d)
```

kleene_star() → *pyformlang.regular_expression.regex.Regex*
Makes the kleene star of the current regex

Returns **regex** – The kleene star of the current regex

Return type *Regex*

Examples

```
>>> regex = Regex("a")
>>> regex_kleene = regex.kleene_star()
>>> regex_kleene.accepts([])
True
>>> regex_kleene.accepts(["a", "a", "a"])
True
```

to_cfg(*starting_symbol='S'*) → CFG

Turns the regex into a context-free grammar

Parameters **starting_symbol** (*Variable*, optional) – The starting symbol

Returns **cfg** – An equivalent context-free grammar

Return type *CFG*

Examples

```
>>> regex = Regex("(a|b)* c")
>>> my_cfg = regex.to_cfg()
>>> my_cfg.contains(["c"])
True
```

to_epsilon_nfa()

Transforms the regular expression into an epsilon NFA

Returns **enfa** – An epsilon NFA equivalent to the regex

Return type *EpsilonNFA*

Examples

```
>>> regex = Regex("abc|d")
>>> regex.to_epsilon_nfa()
```

union(*other:* pyformlang.regular_expression.regex.Regex) → *pyformlang.regular_expression.regex.Regex*

Makes the union with another regex

Equivalent to:

```
>>> regex0 or regex1
```

Parameters **other** (*Regex*) – The other regex

Returns **regex** – The union of the two regex

Return type *Regex*

Examples

```
>>> regex0 = Regex("a b")
>>> regex1 = Regex("c")
>>> regex_union = regex0.union(regex1)
>>> regex_union.accepts(["a", "b"])
>>> regex_union.accepts(["c"])
```

Or equivalently:

```
>>> regex_union = regex0 or regex1
>>> regex_union.accepts(["a", "b"])
```

3.2 Finite Automaton

3.2.1 pyformlang.finite_automaton

This module deals with finite state automata.

Available Classes

FiniteAutomaton A general representation of automata. Cannot be used directly.

DeterministicFiniteAutomaton A deterministic finite automaton

NondeterministicFiniteAutomaton A non-deterministic finite automaton, without epsilon transitions

EpsilonNFA A non-deterministic finite automaton, with epsilon transitions

TransitionFunction A deterministic transition function

NondeterministicTransitionFunction A non-deterministic transition function

State A state (or node) in an automaton

Symbol A symbol (part of the alphabet) in an automaton

Epsilon The epsilon (or empty) symbol

DuplicateTransitionError An error that occurs when trying to add a non-deterministic edge to a deterministic automaton

InvalidEpsilonTransition An exception that occurs when adding an epsilon transition to a non-epsilon NFA.

class pyformlang.finite_automaton.**DeterministicFiniteAutomaton**(*states: Optional[AbstractSet[*pyformlang.finite_automaton.state
= None, *input_symbols: Optional[AbstractSet[*pyformlang.finite_automaton.sym
= None, *transition_function: Optional[*pyformlang.finite_automaton.transition_functi
= None, *start_state: Optional[*pyformlang.finite_automaton.state.State*]
= None, *final_states: Optional[AbstractSet[*pyformlang.finite_automaton.state
= None)

Represents a deterministic finite automaton

This class represents a deterministic finite automaton.

Parameters

- **states** (set of *State*, optional) – A finite set of states
- **input_symbols** (set of *Symbol*, optional) – A finite set of input symbols
- **transition_function** (*TransitionFunction*, optional) – Takes as arguments a state and an input symbol and returns a state.
- **start_state** (*State*, optional) – A start state, element of states
- **final_states** (set of *State*, optional) – A set of final or accepting states. It is a subset of states.

Examples

```
>>> dfa = DeterministicFiniteAutomaton()
```

Creates an empty deterministic finite automaton.

```
>>> dfa.add_transitions([(0, "abc", 1), (0, "d", 1)])
```

Adds two transitions to the deterministic finite automaton. One goes from the state 0 to the state 1 when reading the string "abc". The other also goes from the state 0 to the state 1 when reading the string "d".

```
>>> dfa.add_start_state(0)
```

Adds the start state, 0 here.

```
>>> dfa.add_final_state(1)
```

Adds a final state, 1 here.

```
>>> dfa.is_deterministic()
True
```

Checks if the automaton is deterministic. True here.

```
>>> dfa.accepts(["abc"])
True
```

Checks if the automaton recognize the word composed of a single letter, "abc".

accepts(*word: Iterable[*pyformlang.finite_automaton.symbol.Symbol*]*) → bool

Checks whether the dfa accepts a given word

Parameters **word** (iterable of *Symbol*) – A sequence of input symbols

Returns **is_accepted** – Whether the word is accepted or not

Return type bool

Examples

```
>>> dfa = DeterministicFiniteAutomaton()
>>> dfa.add_transitions([(0, "abc", 1), (0, "d", 1)])
>>> dfa.add_start_state(0)
>>> dfa.add_final_state(1)
>>> dfa.accepts(["abc"])
True
```

add_start_state(*state:* pyformlang.finite_automaton.state.State) → int
Set an initial state

Parameters **state** (*State*) – The new initial state

Returns **done** – 1 is correctly added

Return type int

Examples

```
>>> dfa = DeterministicFiniteAutomaton()
>>> dfa.add_start_state(0)
```

copy() → *pyformlang.finite_automaton.deterministic_finite_automaton.DeterministicFiniteAutomaton*
Copies the current DFA

Returns **enfa** – A copy of the current DFA

Return type DeterministicFiniteAutomaton

Examples

```
>>> dfa = DeterministicFiniteAutomaton()
>>> dfa.add_transitions([(0, "abc", 1), (0, "d", 1)])
>>> dfa.add_start_state(0)
>>> dfa.add_final_state(1)
>>> dfa_copy = dfa.copy()
>>> dfa.is_equivalent_to(dfa_copy)
True
```

is_deterministic() → bool
Checks whether an automaton is deterministic

Returns **is_deterministic** – Whether the automaton is deterministic

Return type bool

Examples

```
>>> dfa = DeterministicFiniteAutomaton()
>>> dfa.is_deterministic()
True
```

is_equivalent_to(*other*)
Check whether two automata are equivalent

Parameters **other** (FiniteAutomaton) – A sequence of input symbols

Returns **are_equivalent** – Whether the two automata are equivalent or not

Return type bool

Examples

```
>>> dfa = DeterministicFiniteAutomaton()
>>> dfa.add_transitions([(0, "abc", 1), (0, "d", 1)])
>>> dfa.add_start_state(0)
>>> dfa.add_final_state(1)
>>> dfa_minimal = dfa.minimize()
>>> dfa.is_equivalent_to(dfa_minimal)
True
```

minimize() → *pyformlang.finite_automaton.deterministic_finite_automaton.DeterministicFiniteAutomaton*
Minimize the current DFA

Returns **dfa** – The minimal DFA

Return type DeterministicFiniteAutomaton

Examples

```
>>> dfa = DeterministicFiniteAutomaton()
>>> dfa.add_transitions([(0, "abc", 1), (0, "d", 1)])
>>> dfa.add_start_state(0)
>>> dfa.add_final_state(1)
>>> dfa_minimal = dfa.minimize()
>>> dfa.is_equivalent_to(dfa_minimal)
True
```

remove_start_state(*state:* pyformlang.finite_automaton.state.State) → int
remove an initial state

Parameters **state** (*State*) – The new initial state

Returns **done** – 1 is correctly added

Return type int

Examples

```
>>> dfa = DeterministicFiniteAutomaton()
>>> dfa.add_start_state(0)
>>> dfa.remove_start_state(0)
```

property start_state: pyformlang.finite_automaton.state.State
The start state

to_deterministic() → *pyformlang.finite_automaton.deterministic_finite_automaton.DeterministicFiniteAutomaton*
Transforms the current automaton into a dfa. Does nothing if the automaton is already deterministic.

Returns **dfa** – A dfa equivalent to the current nfa

Return type DeterministicFiniteAutomaton

Examples

```
>>> dfa0 = DeterministicFiniteAutomaton()
>>> dfa1 = dfa.to_deterministic()
>>> dfa0.is_equivalent_to(dfa1)
True
```

exception pyformlang.finite_automaton.**DuplicateTransitionError**(*s_from:* pyformlang.finite_automaton.state.State, *symb_by:* pyformlang.finite_automaton.symbol.Symbol, *s_to:* pyformlang.finite_automaton.state.State, *s_to_old:* pyformlang.finite_automaton.state.State)
Signals a duplicated transition

Parameters

- **s_from** (*State*) – The source state
- **symb_by** (*Symbol*) – The transition symbol
- **s_to** (*State*) – The wanted new destination state
- **s_to_old** (*State*) – The old destination state

message
An error message summarising the information

Type str

class pyformlang.finite_automaton.**Epsilon**
An epsilon transition

Examples

```
>>> epsilon = Epsilon()
```

class pyformlang.finite_automaton.**EpsilonNFA**(*states: Optional[AbstractSet[pyformlang.finite_automaton.state.State]] = None*, *input_symbols: Optional[AbstractSet[pyformlang.finite_automaton.symbol.Symbol]] = None*, *transition_function: Optional[pyformlang.finite_automaton.nondeterministic_transition_function.No = None*, *start_state: Optional[AbstractSet[pyformlang.finite_automaton.state.State]] = None*, *final_states: Optional[AbstractSet[pyformlang.finite_automaton.state.State]] = None*)

Represents an epsilon NFA

Parameters

- **states** (set of *State*, optional) – A finite set of states
- **input_symbols** (set of *Symbol*, optional) – A finite set of input symbols
- **transition_function** (*NondeterministicTransitionFunction*, optional) – Takes as arguments a state and an input symbol and returns a state.
- **start_state** (set of *State*, optional) – A start state, element of states
- **final_states** (set of *State*, optional) – A set of final or accepting states. It is a subset of states.

Examples

```
>>> enfa = EpsilonNFA()
```

Creates an empty epsilon non-deterministic automaton.

```
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1), (0, "epsilon", 2)])
```

Adds three transition, one of them being an epsilon transition.

```
>>> enfa.add_start_state(0)
```

Adds a start state.

```
>>> enfa.add_final_state(1)
```

Adds a final state.

```
>>> enfa.is_deterministic()
False
```

Checks if the automaton is deterministic.

accepts(*word: Iterable[pyformlang.finite_automaton.symbol.Symbol]*) → bool

Checks whether the epsilon nfa accepts a given word

Parameters **word** (iterable of *Symbol*) – A sequence of input symbols

Returns **is_accepted** – Whether the word is accepted or not

Return type bool

Examples

```
>>> 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.accepts(["abc", "epsilon"])
True
```

```
>>> enfa.accepts(["epsilon"])
False
```

copy() → *pyformlang.finite_automaton.epsilon_nfa.EpsilonNFA*

Copies the current Epsilon NFA

Returns **enfa** – A copy of the current Epsilon NFA

Return type *EpsilonNFA*

Examples

```
>>> 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_copy = enfa.copy()
>>> enfa.is_equivalent_to(enfa_copy)
True
```

eclose(*state:* pyformlang.finite_automaton.state.State) → Set[*pyformlang.finite_automaton.state.State*]

Compute the epsilon closure of a state

Parameters **state** (*State*) – The source state

Returns **states** – The epsilon closure of the source state

Return type set of *State*

Examples

```
>>> 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.eclose(0)
{2}
```

eclose_iterable(*states: Iterable[*pyformlang.finite_automaton.state.State*]*) → Set[*pyformlang.finite_automaton.state.State*]

Compute the epsilon closure of a collection of states

Parameters **states** (iterable of *State*) – The source states

Returns **states** – The epsilon closure of the source state

Return type set of *State*

Examples

```
>>> 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.eclose_iterable([0])
{2}
```

get_complement() → *pyformlang.finite_automaton.epsilon_nfa.EpsilonNFA*

Get the complement of the current Epsilon NFA

Equivalent to:

```
>>> -automaton
```

Returns **enfa** – A complement automaton

Return type *EpsilonNFA*

Examples

```
>>> 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_complement = enfa.get_complement()
>>> enfa_complement.accepts(["epsilon"])
True
```

```
>>> enfa_complement.accepts(["abc"])
False
```

get_difference(*other:* pyformlang.finite_automaton.epsilon_nfa.EpsilonNFA) → *pyformlang.finite_automaton.epsilon_nfa.EpsilonNFA*

Compute the difference with another Epsilon NFA

Equivalent to:

```
>>> automaton0 - automaton1
```

Parameters **other** (*EpsilonNFA*) – The other Epsilon NFA

Returns **enfa** – The difference with the other epsilon NFA

Return type *EpsilonNFA*

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
>>> enfa2 = EpsilonNFA()
>>> enfa2.add_transition(0, "d", 1)
>>> enfa2.add_final_state(1)
>>> enfa2.add_start_state(0)
>>> enfa_diff = enfa.get_difference(enfa2)
>>> enfa_diff.accepts(["d"])
False
```

```
>>> enfa_diff.accepts(["abc"])
True
```

get_intersection(*other:* pyformlang.finite_automaton.epsilon_nfa.EpsilonNFA) → *pyformlang.finite_automaton.epsilon_nfa.EpsilonNFA*

Computes the intersection of two Epsilon NFAs

Equivalent to:

```
>>> automaton0 and automaton1
```

Parameters **other** (*EpsilonNFA*) – The other Epsilon NFA

Returns **enfa** – The intersection of the two Epsilon NFAs

Return type *EpsilonNFA*

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
>>> enfa2 = EpsilonNFA()
>>> enfa2.add_transition(0, "d", 1)
>>> 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
```

is_deterministic() → bool

Checks whether an automaton is deterministic

Returns **is_deterministic** – Whether the automaton is deterministic

Return type bool

Examples

Examples

```
>>> 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
```

is_empty() → bool

Checks if the language represented by the FSM is empty or not

Returns **is_empty** – Whether the language is empty or not

Return type bool

Examples

```
>>> 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_empty()
False
```

minimize() → *DeterministicFiniteAutomaton*

Minimize the current epsilon NFA

Returns **dfa** – The minimal DFA

Return type DeterministicFiniteAutomaton

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
>>> dfa_minimal = enfa.minimize()
>>> dfa_minimal.is_equivalent(enfa)
True
```

reverse() → *pyformlang.finite_automaton.epsilon_nfa.EpsilonNFA*

Compute the reversed EpsilonNFA

Equivalent to: >> ~automaton

Returns **enfa** – The reversed automaton

Return type *EpsilonNFA*

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (1, "d", 2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(2)
>>> enfa_reverse = enfa.reverse()
>>> enfa_reverse.accepts(["d", "abc"])
True
```

to_deterministic() → *DeterministicFiniteAutomaton*

Transforms the epsilon-nfa into a dfa

Returns **dfa** – A dfa equivalent to the current nfa

Return type DeterministicFiniteAutomaton

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
>>> dfa = enfa.to_deterministic()
>>> dfa.is_deterministic()
True
```

```
>>> enfa.is_equivalent_to(dfa)
True
```

to_regex() → Regex

Transforms the EpsilonNFA to a regular expression

Returns **regex** – A regular expression equivalent to the current Epsilon NFA

Return type *Regex*

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
>>> regex = enfa.to_regex()
>>> regex.accepts(["abc"])
True
```

class pyformlang.finite_automaton.**FiniteAutomaton**

Represents a general finite automaton

_states

A finite set of states

Type set of *State*, optional

_input_symbols

A finite set of input symbols

Type set of *Symbol*, optional

_transition_function

Takes as arguments a state and an input symbol and returns a state.

Type *NondeterministicTransitionFunction* , optional

_start_state

A start state, element of states

Type set of *State*, optional

_final_states

A set of final or accepting states. It is a subset of states.

Type set of *State*, optional

add_final_state(*state:* pyformlang.finite_automaton.state.State) → int
Adds a new final state

Parameters **state** (*State*) – A new final state

Returns **done** – 1 is correctly added

Return type int

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
```

add_start_state(*state:* pyformlang.finite_automaton.state.State) → int
Set an initial state

Parameters **state** (*State*) – The new initial state

Returns **done** – 1 is correctly added

Return type int

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
```

add_symbol(*symbol:* pyformlang.finite_automaton.symbol.Symbol)
Add a symbol

Parameters **symbol** (*Symbol*) – The symbol

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_symbol("a")
```

add_transition(*s_from:* pyformlang.finite_automaton.state.State, *symb_by:* pyformlang.finite_automaton.symbol.Symbol, *s_to:* pyformlang.finite_automaton.state.State) → int
Adds a transition to the nfa

Parameters

- **s_from** (*State*) – The source state
- **symb_by** (*Symbol*) – The transition symbol
- **s_to** (*State*) – The destination state

Returns **done** – Always 1

Return type int

Raises *DuplicateTransitionError* – If the transition already exists

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transition(0, "abc", 1)
```

add_transitions(*transitions_list*)

Adds several transitions to the automaton

Parameters **transitions_list** (*list of triples of (s_from, symb_by, s_to)*) – A list of all the transitions represented as triples as they would be used in add_transition

Returns **done** – Always 1

Return type int

Raises *DuplicateTransitionError* – If the transition already exists

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
```

property final_states

The final states

classmethod from_networkx(*graph*)

Import a networkx graph into an finite state automaton. The imported graph requires to have the good format, i.e. to come from the function to_networkx

Parameters **graph** – The graph representation of the automaton

Returns A epsilon nondeterministic finite automaton read from the graph

Return type enfa

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
>>> graph = enfa.to_networkx()
>>> enfa_from_nx = EpsilonNFA.from_networkx(graph)
```

get_number_transitions() → int

Gives the number of transitions

Returns **n_transitions** – The number of deterministic transitions

Return type int

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.get_number_transitions()
3
```

is_acyclic() → bool
Checks if the automaton is acyclic

Returns **is_acyclic** – Whether the automaton is acyclic or not

Return type bool

Examples

```
>>> 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_acyclic()
True
```

is_deterministic()
Checks if the automaton is deterministic

is_equivalent_to(*other*)
Checks if the current automaton is equivalent to a given one.

Parameters **other** – An other finite state automaton

Returns **is_equivalent** – Whether the two automata are equivalent or not

Return type bool

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
>>> dfa = enfa.to_deterministic()
>>> dfa.is_deterministic()
True
```

is_final_state(*state: pyformlang.finite_automaton.state.State*) → bool
Checks if a state is final

Parameters **state** (*State*) – The state to check

Returns **is_final** – Whether the state is final or not

Return type bool

Examples

```
>>> 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_final_state(1)
True
```

remove_final_state(*state:* pyformlang.finite_automaton.state.State) → int

Remove a final state

Parameters **state** (*State*) – A final state to remove

Returns **done** – 0 if it was not a final state, 1 otherwise

Return type int

Examples

```
>>> 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.remove_final_state(1)
```

remove_start_state(*state:* pyformlang.finite_automaton.state.State) → int

remove an initial state

Parameters **state** (*State*) – The new initial state

Returns **done** – 1 is correctly added

Return type int

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.remove_start_state(0)
```

remove_transition(*s_from:* pyformlang.finite_automaton.state.State, *symb_by:* pyformlang.finite_automaton.symbol.Symbol, *s_to:* pyformlang.finite_automaton.state.State) → int

Remove a transition of the nfa

Parameters

- **s_from** (*State*) – The source state
- **symb_by** (*Symbol*) – The transition symbol
- **s_to** (*State*) – The destination state

Returns **done** – 1 if the transition existed, 0 otherwise

Return type int

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transition(0, "abc", 1)
>>> enfa.remove_transition(0, "abc", 1)
```

property start_states
The start states

property states
Gives the states

Returns **states** – The states

Return type set of *State*

property symbols
The symbols

to_deterministic()
Turns the automaton into a deterministic one

to_dict()
Get the dictionary representation of the transition function. The keys of the dictionary are the source nodes. The items are dictionaries where the keys are the symbols of the transitions and the items are the set of target nodes.

Returns **transition_dict** – The transitions as a dictionary.

Return type dict

Examples

```
>>> 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_dict = enfa.to_dict()
```

to_fst() → *pyformlang.fst.fst.FST*
Turns the finite automaton into a finite state transducer

The transducers accepts only the words in the language of the automaton and output the input word

Returns **fst** – The equivalent FST

Return type *FST*

Examples

```
>>> enfa = EpsilonNFA()
>>> fst = enfa.to_fst()
>>> fst.states
{}
```

to_networkx() → networkx.classes.multidigraph.MultiDiGraph
Transform the current automaton into a networkx graph

Returns **graph** – A networkx MultiDiGraph representing the automaton

Return type networkx.MultiDiGraph

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transitions([(0, "abc", 1), (0, "d", 1),          (0, "epsilon",␣
↪2)])
>>> enfa.add_start_state(0)
>>> enfa.add_final_state(1)
>>> graph = enfa.to_networkx()
```

write_as_dot(*filename*)
Write the automaton in dot format into a file

Parameters **filename** (*str*) – The filename where to write the dot file

Examples

```
>>> 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.write_as_dot("enfa.dot")
```

exception pyformlang.finite_automaton.**InvalidEpsilonTransition**
Exception raised when an epsilon transition is created in deterministic automaton

class pyformlang.finite_automaton.**NondeterministicFiniteAutomaton**(*states: Optional[AbstractSet[*pyformlang.finite_automaton.
= None, *input_symbols: Optional[AbstractSet[*pyformlang.finite_automaton.
= None, *transition_function: Optional[*pyformlang.finite_automaton.nondetermin
= None, *start_state: Optional[AbstractSet[*pyformlang.finite_automaton.
= None, *final_states: Optional[AbstractSet[*pyformlang.finite_automaton.
= None)

Represents a nondeterministic finite automaton

This class represents a nondeterministic finite automaton, where epsilon transition are forbidden.

Parameters

- **states** (set of *State*, optional) – A finite set of states
- **input_symbols** (set of *Symbol*, optional) – A finite set of input symbols
- **transition_function** (*NondeterministicTransitionFunction* , optional) – Takes as arguments a state and an input symbol and returns a state.
- **start_state** (*State*, optional) – A start state, element of states
- **final_states** (set of *State*, optional) – A set of final or accepting states. It is a subset of states.

Examples

```
>>> nfa = NondeterministicFiniteAutomaton()
```

Creates the NFA.

```
>>> nfa.add_transitions([(0, "a", 1), (0, "a", 2)])
```

Adds two transitions.

```
>>> nfa.add_start_state(0)
```

Adds a start state.

```
>>> nfa.add_final_state(1)
```

Adds a final state.

```
>>> nfa.accepts(["a"])
True
```

```
>>> nfa.is_deterministic()
False
```

accepts(*word: Iterable[*pyformlang.finite_automaton.symbol.Symbol*]*) → bool

Checks whether the nfa accepts a given word

Parameters **word** (iterable of *Symbol*) – A sequence of input symbols

Returns **is_accepted** – Whether the word is accepted or not

Return type bool

Examples

```
>>> nfa = NondeterministicFiniteAutomaton()
>>> nfa.add_transitions([(0, "a", 1), (0, "a", 2)])
>>> nfa.add_start_state(0)
>>> nfa.add_final_state(1)
>>> nfa.accepts(["a"])
True
```

add_transition(*s_from:* pyformlang.finite_automaton.state.State, *symb_by:* pyformlang.finite_automaton.symbol.Symbol, *s_to:* pyformlang.finite_automaton.state.State) → int

Adds a transition to the nfa

Parameters

- **s_from** (*State*) – The source state
- **symb_by** (*Symbol*) – The transition symbol
- **s_to** (*State*) – The destination state

Returns **done** – Always 1

Return type int

Raises ***DuplicateTransitionError*** – If the transition already exists

Examples

```
>>> enfa = EpsilonNFA()
>>> enfa.add_transition(0, "abc", 1)
```

is_deterministic() → bool

Checks whether an automaton is deterministic

Returns **is_deterministic** – Whether the automaton is deterministic

Return type bool

Examples

```
>>> nfa = NondeterministicFiniteAutomaton()
>>> nfa.add_transitions([(0, "a", 1), (0, "a", 2)])
>>> nfa.add_start_state(0)
>>> nfa.add_final_state(1)
>>> nfa.is_deterministic()
False
```

to_deterministic() → *DeterministicFiniteAutomaton*

Transforms the nfa into a dfa

Returns **dfa** – A dfa equivalent to the current nfa

Return type DeterministicFiniteAutomaton

Examples

```
>>> nfa = NondeterministicFiniteAutomaton()
>>> nfa.add_transitions([(0, "a", 1), (0, "a", 2)])
>>> nfa.add_start_state(0)
>>> nfa.add_final_state(1)
>>> dfa = nfa.to_deterministic()
>>> nfa.is_equivalent_to(dfa)
True
```

class pyformlang.finite_automaton.**NondeterministicTransitionFunction**

A nondeterministic transition function in a finite automaton.

The difference with a deterministic transition is that the return value is a set of States

Examples

```
>>> transition = NondeterministicTransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
```

Creates a transition function and adds a transition.

add_transition(*s_from:* pyformlang.finite_automaton.state.State, *symb_by:* pyformlang.finite_automaton.symbol.Symbol, *s_to:* pyformlang.finite_automaton.state.State) → int

Adds a new transition to the function

Parameters

- **s_from** (*State*) – The source state
- **symb_by** (*Symbol*) – The transition symbol
- **s_to** (*State*) – The destination state

Returns **done** – Always 1

Return type int

Examples

```
>>> transition = NondeterministicTransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
```

get_edges()

Gets the edges

Returns **edges** – A generator of edges

Return type generator of (*State*, *Symbol*, *State*)

get_number_transitions() → int

Gives the number of transitions describe by the function

Returns **n_transitions** – The number of transitions

Return type int

Examples

```
>>> transition = NondeterministicTransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
>>> transition.get_number_transitions()
1
```

is_deterministic()
Whether the transition function is deterministic

Returns **is_deterministic** – Whether the function is deterministic

Return type bool

Examples

```
>>> transition = NondeterministicTransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
>>> transition.is_deterministic()
True
```

remove_transition(*s_from:* pyformlang.finite_automaton.state.State, *symb_by:* pyformlang.finite_automaton.symbol.Symbol, *s_to:* pyformlang.finite_automaton.state.State) → int
Removes a transition to the function

Parameters

- **s_from** (*State*) – The source state
- **symb_by** (*Symbol*) – The transition symbol
- **s_to** (*State*) – The destination state

Returns **done** – 1 is the transition was found, 0 otherwise

Return type int

Examples

```
>>> transition = NondeterministicTransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
>>> transition.remove_transition(State(0), Symbol("a"), State(1))
```

to_dict()
Get the dictionary representation of the transition function. The keys of the dictionary are the source nodes. The items are dictionaries where the keys are the symbols of the transitions and the items are the set of target nodes.

Returns **transition_dict** – The transitions as a dictionary.

Return type dict

class pyformlang.finite_automaton.**State**(*value*)
A state in a finite automaton

Parameters **value** (*any*) – The value of the state

Examples

```
>>> from pyformlang.finite_automaton import State
>>> State("A")
A
```

class pyformlang.finite_automaton.**Symbol**(*value: Any*)

A symbol in a finite automaton

Parameters **value** (*any*) – The value of the symbol

Examples

```
>>> from pyformlang.finite_automaton import Symbol
>>> Symbol("A")
A
```

class pyformlang.finite_automaton.**TransitionFunction**

A transition function in a finite automaton.

This is a deterministic transition function.

_transitions

A dictionary which contains the transitions of a finite automaton

Type dict

Examples

```
>>> transition = TransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
```

Creates a transition function and adds a transition.

add_transition(*s_from:* pyformlang.finite_automaton.state.State, *symb_by:* pyformlang.finite_automaton.symbol.Symbol, *s_to:* pyformlang.finite_automaton.state.State) → int

Adds a new transition to the function

Parameters

- **s_from** (*State*) – The source state
- **symb_by** (*Symbol*) – The transition symbol
- **s_to** (*State*) – The destination state

Returns **done** – Always 1

Return type int

Raises *DuplicateTransitionError* – If the transition already exists

Examples

```
>>> transition = TransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
```

get_edges()
Gets the edges

Returns **edges** – A generator of edges

Return type generator of (*State*, *Symbol*, *State*)

get_number_transitions() → int
Gives the number of transitions describe by the deterministic function

Returns **n_transitions** – The number of deterministic transitions

Return type int

Examples

```
>>> transition = TransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
>>> transition.get_number_transitions()
1
```

remove_transition(*s_from:* pyformlang.finite_automaton.state.State, *symb_by:* pyformlang.finite_automaton.symbol.Symbol, *s_to:* pyformlang.finite_automaton.state.State) → int
Removes a transition to the function

Parameters

- **s_from** (*State*) – The source state
- **symb_by** (*Symbol*) – The transition symbol
- **s_to** (*State*) – The destination state

Returns **done** – 1 is the transition was found, 0 otherwise

Return type int

Examples

```
>>> transition = TransitionFunction()
>>> transition.add_transition(State(0), Symbol("a"), State(1))
>>> transition.remove_transition(State(0), Symbol("a"), State(1))
```

to_dict()
Get the dictionary representation of the transition function. The keys of the dictionary are the source nodes. The items are dictionaries where the keys are the symbols of the transitions and the items are the set of target nodes.

Returns **transition_dict** – The transitions as a dictionary.

Return type dict

3.3 Finite State Transducer

3.3.1 pyformlang.fst

This module deals with finite state transducers.

Available Classes

FST A Finite State Transducer

class pyformlang.fst.**FST**
Representation of a Finite State Transducer

add_final_state(*final_state: Any*)
Add a final state

Parameters **final_state** (*any*) – The final state to add

add_start_state(*start_state: Any*)
Add a start state

Parameters **start_state** (*any*) – The start state

add_transition(*s_from: Any*, *input_symbol: Any*, *s_to: Any*, *output_symbols: Iterable[Any]*)
Add a transition to the FST

Parameters

- **s_from** (*any*) – The source state
- **input_symbol** (*any*) – The symbol to read
- **s_to** (*any*) – The destination state
- **output_symbols** (*iterable of Any*) – The symbols to output

add_transitions(*transitions_list*)
Adds several transitions to the FST

Parameters **transitions_list** (*list of tuples*) – The tuples have the form (s_from, in_symbol, s_to, out_symbols)

concatenate(*other_fst*)
Makes the concatenation of two fst :param other_fst: The other FST :type other_fst: *FST*

Returns **fst_concatenate** – A new FST which is the concatenation of the two given FST

Return type *FST*

property final_states
Get the final states of the FST

Returns **final_states** – The final states of the FST

Return type set of any

classmethod from_networkx(*graph*)
Import a networkx graph into an finite state transducer. The imported graph requires to have the good format, i.e. to come from the function to_networkx

Parameters **graph** – The graph representation of the FST

Returns A FST read from the graph

Return type enfa

get_number_transitions() → int
Get the number of transitions in the FST

Returns **n_transitions** – The number of transitions

Return type int

property input_symbols
Get the input symbols of the FST

Returns **input_symbols** – The input symbols of the FST

Return type set of any

intersection(*indexed_grammar*)
Compute the intersection with an other object

Equivalent to: >> fst and indexed_grammar

kleene_star()
Computes the kleene star of the FST

Returns **fst_star** – A FST representing the kleene star of the FST

Return type *FST*

property output_symbols
Get the output symbols of the FST

Returns **output_symbols** – The output symbols of the FST

Return type set of any

property start_states
Get the start states of the FST

Returns **start_states** – The start states of the FST

Return type set of any

property states
Get the states of the FST

Returns **states** – The states

Return type set of any

to_networkx() → networkx.classes.multidigraph.MultiDiGraph
Transform the current fst into a networkx graph

Returns **graph** – A networkx MultiDiGraph representing the fst

Return type networkx.MultiDiGraph

property transitions
Gives the transitions as a dictionary

translate(*input_word: Iterable[Any]*, *max_length: int = - 1*) → Iterable[Any]
Translate a string into another using the FST

Parameters

- **input_word** (*iterable of any*) – The word to translate
- **max_length** (*int, optional*) – The maximum size of the output word, to prevent infinite generation due to epsilon transitions

Returns **output_word** – The translation of the input word

Return type iterable of any

union(*other_fst*)
Makes the union of two fst :param other_fst: The other FST :type other_fst: *FST*

Returns **union_fst** – A new FST which is the union of the two given FST

Return type *FST*

write_as_dot(*filename*)
Write the FST in dot format into a file

Parameters **filename** (*str*) – The filename where to write the dot file

3.4 Context Free Grammar

3.4.1 pyformlang.cfg

This submodule implements functions related to context-free grammars.

Available Classes

CFG The main context-free grammar class

Production A class to represent a production in a CFG

Variable A context-free grammar variable

Terminal A context-free grammar terminal

Epsilon The epsilon symbol (special terminal)

class pyformlang.cfg.**CFG**(*variables: Optional[AbstractSet[*pyformlang.cfg.variable.Variable*]] = None*, *terminals: Optional[AbstractSet[*pyformlang.cfg.terminal.Terminal*]] = None*, *start_symbol: Optional[*pyformlang.cfg.variable.Variable*] = None*, *productions: Optional[Iterable[*pyformlang.cfg.production.Production*]] = None*)
A class representing a context free grammar

Parameters

- **variables** (set of *Variable*, optional) – The variables of the CFG
- **terminals** (set of *Terminal*, optional) – The terminals of the CFG
- **start_symbol** (*Variable*, optional) – The start symbol
- **productions** (set of *Production*, optional) – The productions or rules of the CFG

concatenate(*other:* pyformlang.cfg.cfg.CFG) → *pyformlang.cfg.cfg.CFG*
Makes the concatenation of two CFGs

Equivalent to: >> cfg0 + cfg1

Parameters **other** (*CFG*) – The other CFG to concatenate with

Returns **new_cfg** – The CFG resulting of the concatenation of the two CFGs

Return type *CFG*

contains(*word: Iterable[*pyformlang.cfg.terminal.Terminal*]*) → bool
Gives the membership of a word to the grammar

Parameters **word** (iterable of *Terminal*) – The word to check

Returns **contains** – Whether word if in the CFG or not

Return type bool

eliminate_unit_productions() → *pyformlang.cfg.cfg.CFG*
Eliminate all the unit production in the CFG

Returns **new_cfg** – A new CFG equivalent without unit productions

Return type *CFG*

classmethod from_text(*text*, *start_symbol=Variable(S)*)
Read a context free grammar from a text. The text contains one rule per line. The structure of a production is: head -> body1 | body2 | … | bodyn where | separates the bodies. A variable (or non terminal) begins by a capital letter. A terminal begins by a non-capital character Terminals and Variables are separated by spaces. An epsilon symbol can be represented by epsilon, $, , or . If you want to have a variable name starting with a non-capital letter or a terminal starting with a capital letter, you can explicitly give the type of your symbol with "VAR:yourVariableName" or "TER:yourTerminalName" (with the quotation marks). For example: S -> "TER:John" "VAR:d" a b

Parameters

- **text** (*str*) – The text of transform
- **start_symbol** (*str, optional*) – The start symbol, S by default

Returns **cfg** – A context free grammar.

Return type *CFG*

generate_epsilon()
Whether the grammar generates epsilon or not

Returns **generate_epsilon** – Whether epsilon is generated or not by the CFG

Return type bool

get_closure() → *pyformlang.cfg.cfg.CFG*
Gets the closure of the CFG (*)

Returns **new_cfg** – The closure of the current CFG

Return type *CFG*

get_cnf_parse_tree(*word*)
Get a parse tree of the CNF of this grammar

Parameters **word** (iterable of *Terminal*) – The word to look for

Returns **derivation** – The parse tree

Return type ParseTree

get_generating_symbols() → AbstractSet[pyformlang.cfg.cfg_object.CFGObject]
Gives the objects which are generating in the CFG

Returns **generating_symbols** – The generating symbols of the CFG

Return type set of CFGObject

get_nullable_symbols() → AbstractSet[pyformlang.cfg.cfg_object.CFGObject]
Gives the objects which are nullable in the CFG

Returns **nullable_symbols** – The nullable symbols of the CFG

Return type set of `CFGObject`

`get_positive_closure`() → *pyformlang.cfg.cfg.CFG*
Gets the positive closure of the CFG (+)

Returns **new_cfg** – The positive closure of the current CFG

Return type *CFG*

`get_reachable_symbols`() → AbstractSet[pyformlang.cfg.cfg_object.CFGObject]
Gives the objects which are reachable in the CFG

Returns **reachable_symbols** – The reachable symbols of the CFG

Return type set of `CFGObject`

`get_unit_pairs`() → AbstractSet[Tuple[*pyformlang.cfg.variable.Variable*, *pyformlang.cfg.variable.Variable*]]
Finds all the unit pairs

Returns **unit_pairs** – The unit pairs

Return type set of tuple of `Variable`

`get_words`(*max_length: int = - 1*)
Get the words generated by the CFG

Parameters **max_length** (*int*) – The maximum length of the words to return

`intersection`(*other: Any*) → *pyformlang.cfg.cfg.CFG*
Gives the intersection of the current CFG with an other object

Equivalent to: >> cfg and regex

Parameters **other** (*any*) – The other object

Returns **new_cfg** – A CFG representing the intersection with the other object

Return type *CFG*

Raises `NotImplementedError` – When trying to intersect with something else than a regex or a finite automaton

`is_empty`() → bool
Says whether the CFG is empty or not

Returns **is_empty** – Whether the CFG is empty or not

Return type bool

`is_finite`() → bool
Tests if the grammar is finite or not

Returns **is_finite** – Whether the grammar is finite or not

Return type bool

`is_normal_form`()
Tells is the current grammar is in Chomsky Normal Form or not

Returns **is_normal_form** – If the current grammar is in CNF

Return type bool

property productions: AbstractSet[pyformlang.cfg.production.Production]
Gives the productions

Returns **productions** – The productions of the CFG

Return type set of *Production*

remove_epsilon() → *pyformlang.cfg.cfg.CFG*
Removes the epsilon of a cfg

Returns **new_cfg** – The CFG without epsilons

Return type *CFG*

remove_useless_symbols() → *pyformlang.cfg.cfg.CFG*
Removes useless symbols in a CFG

Returns **new_cfg** – The CFG without useless symbols

Return type *CFG*

reverse() → *pyformlang.cfg.cfg.CFG*
Reverse the current CFG

Equivalent to: >> ~cfg

Returns **new_cfg** – Reverse the current CFG

Return type *CFG*

property start_symbol: pyformlang.cfg.variable.Variable
Gives the start symbol

Returns **start_variable** – The start symbol of the CFG

Return type *Variable*

substitute(*substitution: Dict[*pyformlang.cfg.terminal.Terminal, pyformlang.cfg.cfg.CFG*]*) → *pyformlang.cfg.cfg.CFG*
Substitutes CFG to terminals in the current CFG

Parameters **substitution** (dict of *Terminal* to *CFG*) – A substitution

Returns **new_cfg** – A new CFG recognizing the substitution

Return type *CFG*

property terminals: AbstractSet[pyformlang.cfg.terminal.Terminal]
Gives the terminals

Returns **terminals** – The terminals of the CFG

Return type set of *Terminal*

to_normal_form() → *pyformlang.cfg.cfg.CFG*
Gets the Chomsky Normal Form of a CFG

Returns **new_cfg** – A new CFG equivalent in the CNF form

Return type *CFG*

> **Warning:** As described in Hopcroft's textbook, a normal form does not generate the epsilon word. So, the grammar generated by this function is equivalent to the original grammar except if this grammar generates the epsilon word. In that case, the language of the generated grammar contains the same word as before, except the epsilon word.

to_pda() → *pyformlang.pda.pda.PDA*
Converts the CFG to a PDA that generates on empty stack an equivalent language

Returns **new_pda** – The equivalent PDA when accepting on empty stack

Return type *PDA*

to_text()
Turns the grammar into its string representation. This might lose some type information and the start_symbol. :returns: **text** – The grammar as a string. :rtype: str

union(*other:* pyformlang.cfg.cfg.CFG) → *pyformlang.cfg.cfg.CFG*
Makes the union of two CFGs

Equivalent to: >> cfg0 or cfg1

Parameters **other** (*CFG*) – The other CFG to unite with

Returns **new_cfg** – The CFG resulting of the union of the two CFGs

Return type *CFG*

property variables: AbstractSet[pyformlang.cfg.variable.Variable]
Gives the variables

Returns **variables** – The variables of the CFG

Return type set of *Variable*

class pyformlang.cfg.**Epsilon**
An epsilon terminal

class pyformlang.cfg.**LLOneParser**(*cfg*)
A LL(1) parser

Parameters **cfg** (*CFG*) – A context-free Grammar

get_first_set()
Used in LL(1)

get_follow_set()
Get follow set

get_llone_parse_tree(*word*)
Get LL(1) parse Tree

Parameters **word** (*list*) – The word to parse

Returns **parse_tree** – The parse tree

Return type ParseTree

Raises **NotParsableException** – When the word cannot be parsed

get_llone_parsing_table()
Get the LL(1) parsing table From: https://www.slideshare.net/MahbuburRahman273/ll1-parser-in-compilers

is_llone_parsable()
Checks whether the grammar can be parse with the LL(1) parser.

Returns is_parsable

Return type bool

class pyformlang.cfg.**Production**(*head:* pyformlang.cfg.variable.Variable, *body:* *List[pyformlang.cfg.cfg_object.CFGObject]*, *filtering=True*)
A production or rule of a CFG

Parameters

- **head** (*Variable*) – The head of the production
- **body** (iterable of CFGObject) – The body of the production

property body: List[pyformlang.cfg.cfg_object.CFGObject]
Get the body objects

property head: pyformlang.cfg.variable.Variable
Get the head variable

is_normal_form()
Tells is the production is in Chomsky Normal Form

Returns is_normal_form – If the production is in CNF

Return type bool

class pyformlang.cfg.**Terminal**(*value: Any*)
An terminal in a CFG

Parameters value (*any*) – The value of the terminal

class pyformlang.cfg.**Variable**(*value*)
An variable in a CFG

Parameters value (*any*) – The value of the variable

3.5 Push-Down Automata

3.5.1 pyformlang.pda

This module deals with push-down automata.

Available Classes

PDA A Push-Down Automaton

State A push-down automaton state

Symbol A push-down automaton symbol

StackSymbol A push-down automaton stack symbol

Epsilon A push-down automaton epsilon symbol

class pyformlang.pda.**Epsilon**
An epsilon symbol

class pyformlang.pda.**PDA**(*states: Optional[AbstractSet[*pyformlang.pda.state.State*]] = None*, *input_symbols: Optional[AbstractSet[*pyformlang.pda.symbol.Symbol*]] = None*, *stack_alphabet: Optional[AbstractSet[*pyformlang.pda.stack_symbol.StackSymbol*]] = None*, *transition_function: Optional[pyformlang.pda.transition_function.TransitionFunction] = None*, *start_state: Optional[*pyformlang.pda.state.State*] = None*, *start_stack_symbol: Optional[*pyformlang.pda.stack_symbol.StackSymbol*] = None*, *final_states: Optional[AbstractSet[*pyformlang.pda.state.State*]] = None*)

Representation of a pushdown automaton

Parameters

- **states** (set of *State*, optional) – A finite set of states
- **input_symbols** (set of *Symbol*, optional) – A finite set of input symbols
- **stack_alphabet** (set of *StackSymbol*, optional) – A finite stack alphabet
- **transition_function** (TransitionFunction, optional) – Takes as arguments a state, an input symbol and a stack symbol and returns a state and a string of stack symbols push on the stacked to replace X
- **start_state** (*State*, optional) – A start state, element of states
- **start_stack_symbol** (*StackSymbol*, optional) – The stack is initialized with this stack symbol
- **final_states** (set of *State*, optional) – A set of final or accepting states. It is a subset of states.

add_final_state(*state:* pyformlang.pda.state.State)

Adds a final state to the automaton

Parameters **state** (*State*) – The state to add

add_transition(*s_from:* pyformlang.pda.state.State, *input_symbol:* pyformlang.pda.symbol.Symbol, *stack_from:* pyformlang.pda.stack_symbol.StackSymbol, *s_to:* pyformlang.pda.state.State, *stack_to: List[*pyformlang.pda.stack_symbol.StackSymbol*]*)

Add a transition to the PDA

Parameters

- **s_from** (*State*) – The starting symbol
- **input_symbol** (*Symbol*) – The input symbol for the transition
- **stack_from** (*StackSymbol*) – The stack symbol of the transition
- **s_to** (*State*) – The new state
- **stack_to** (list of *StackSymbol*) – The string of stack symbol which replace the stack_from

add_transitions(*transitions*)

Adds several transitions

Parameters **transitions** – Transitions as they would be given to add_transition

property final_states

The final states of the PDA :returns: **final_states** – The final states of the PDA :rtype: iterable of *State*

classmethod from_networkx(*graph*)

Import a networkx graph into a PDA. The imported graph requires to have the good format, i.e. to come from the function to_networkx

Parameters **graph** – The graph representation of the PDA

Returns A PDA automaton read from the graph

Return type pda

get_number_transitions() → int
Gets the number of transitions in the PDA

Returns **n_transitions** – The number of transitions

Return type int

property input_symbols
The input symbols of the PDA

Returns **input_symbols** – The input symbols of the PDA

Return type iterable of *Symbol*

intersection(*other: Any*) → *pyformlang.pda.pda.PDA*
Gets the intersection of the language L generated by the current PDA when accepting by final state with something else

Currently, it only works for regular languages (represented as regular expressions or finite automata) as the intersection between two PDAs is not context-free (it cannot be represented with a PDA).

Equivalent to: >> pda and regex

Parameters **other** (*any*) – The other part of the intersection

Returns **new_pda** – The pda resulting of the intersection

Return type *PDA*

Raises **NotImplementedError** – When intersecting with something else than a regex or a finite automaton

set_start_stack_symbol(*start_stack_symbol:* pyformlang.pda.stack_symbol.StackSymbol)
Sets the start stack symbol to the automaton

Parameters **start_stack_symbol** (*StackSymbol*) – The start stack symbol

set_start_state(*start_state:* pyformlang.pda.state.State)
Sets the start state to the automaton

Parameters **start_state** (*State*) – The start state

property stack_symbols
The stack symbols of the PDA

Returns **stack_symbols** – The stack symbols of the PDA

Return type iterable of *StackSymbol*

property start_state
Get start state

property states
Get the states fo the PDA :returns: **states** – The states of the PDA :rtype: iterable of *State*

to_cfg() → *pyformlang.cfg.cfg.CFG*
Turns the language L generated by this PDA when accepting on empty stack into a CFG that accepts the same language L

Returns **new_cfg** – The equivalent CFG

Return type *CFG*

`to_dict()`
Get the transitions of the PDA as a dictionary :returns: **transitions** – The transitions :rtype: dict

`to_empty_stack()` → *pyformlang.pda.pda.PDA*
Turns the current PDA that accepts a language L by final state to another PDA that accepts the same language L by empty stack

Returns **new_pda** – The new PDA which accepts by empty stack the language that was accepted by final state

Return type *PDA*

`to_final_state()` → *pyformlang.pda.pda.PDA*
Turns the current PDA that accepts a language L by empty stack to another PDA that accepts the same language L by final state

Returns **new_pda** – The new PDA which accepts by final state the language that was accepted by empty stack

Return type *PDA*

`to_networkx()` → networkx.classes.multidigraph.MultiDiGraph
Transform the current pda into a networkx graph

Returns **graph** – A networkx MultiDiGraph representing the pda

Return type networkx.MultiDiGraph

`write_as_dot`(*filename*)
Write the PDA in dot format into a file

Parameters **filename** (*str*) – The filename where to write the dot file

`class pyformlang.pda.StackSymbol`(*value*)
A StackSymbol in a pushdown automaton

Parameters **value** (*any*) – The value of the state

`property value`
Returns the value of the stack symbol

Returns **value** – any

Return type The value

`class pyformlang.pda.State`(*value*)
A State in a pushdown automaton

Parameters **value** (*any*) – The value of the state

`property value`
Returns the value of the state

Returns **value** – any

Return type The value

`class pyformlang.pda.Symbol`(*value*)
A Symbol in a pushdown automaton

Parameters **value** (*any*) – The value of the state

`property value`
Returns the value of the symbol

Returns **value** – any

Return type The value

3.6 Indexed Grammar

3.6.1 pyformlang.indexed_grammar

This module deals with indexed grammars.

Available Classes

IndexedGrammar An indexed grammar

Rules A representation of a set of indexed grammar rules

EndRule An end rule, turning a variable into a terminal

ConsumptionRule A consumption rule, consuming something from the stack

ProductionRule A production rule, pushing something on the stack

DuplicationRule A duplication rule, duplicating the stack

`class` pyformlang.indexed_grammar.**ConsumptionRule**(*f: Any*, *left: Any*, *right: Any*)

Contains a representation of a consumption rule, i.e. a rule of the form: C[r sigma] -> B[sigma]

Parameters

- **f** (*any*) – The consumed symbol
- **left** (*any*) – The non terminal on the left (here C)
- **right** (*any*) – The non terminal on the right (here B)

`property` **f_parameter**: `Any`
Gets the symbol which is consumed

Returns **f** – The symbol being consumed by the rule

Return type any

is_consumption() → bool
Whether the rule is a consumption rule or not

Returns **is_consumption** – Whether the rule is a consumption rule or not

Return type bool

`property` **left_term**: `Any`
Gets the symbol on the left of the rule

left [any] The left symbol of the rule

`property` **non_terminals**: `Iterable[Any]`
Gets the non-terminals used in the rule

non_terminals [iterable of any] The non_terminals used in the rule

property production
The production

Returns **right_terms** – The production

Return type any

property right: Any
Gets the symbole on the right of the rule

right [any] The right symbol

property right_term
The unique right term

Returns **right_term** – The unique right term of the rule

Return type iterable of any

property right_terms
The right terms

Returns **right_terms** – The right terms of the rule

Return type iterable of any

property terminals: AbstractSet[Any]
Gets the terminals used in the rule

terminals [set of any] The terminals used in the rule

class pyformlang.indexed_grammar.**DuplicationRule**(*left_term*, *right_term0*, *right_term1*)

Represents a duplication rule, i.e. a rule of the form: A[sigma] -> B[sigma] C[sigma]

Parameters

- **left_term** (*any*) – The non-terminal on the left of the rule (A here)
- **right_term0** (*any*) – The first non-terminal on the right of the rule (B here)
- **right_term1** (*any*) – The second non-terminal on the right of the rule (C here)

property f_parameter
The f parameter

Returns **f** – The f parameter

Return type any

is_duplication() → bool
Whether the rule is a duplication rule or not

Returns **is_duplication** – Whether the rule is a duplication rule or not

Return type bool

property left_term: Any
Gives the non-terminal on the left of the rule

Returns **left_term** – The left term of the rule

Return type any

property non_terminals: Iterable[Any]
Gives the set of non-terminals used in this rule

Returns **non_terminals** – The non terminals used in this rule

Return type iterable of any

property production
The production

Returns **right_terms** – The production

Return type any

property right_term
The unique right term

Returns **right_term** – The unique right term of the rule

Return type iterable of any

property right_terms: Tuple[Any, Any]
Gives the non-terminals on the right of the rule

Returns **right_terms** – The right terms of the rule

Return type iterable of any

property terminals: AbstractSet[Any]
Gets the terminals used in the rule

Returns **terminals** – The terminals used in this rule

Return type set of any

class pyformlang.indexed_grammar.**EndRule**(*left*, *right*)

Represents an end rule, i.e. a rule of the form: A[sigma] -> a

Parameters

- **left** (*any*) – The non-terminal on the left, "A" here
- **right** (*any*) – The terminal on the right, "a" here

property f_parameter
The f parameter

Returns **f** – The f parameter

Return type any

static is_end_rule() → bool
Whether the rule is an end rule or not

Returns **is_end** – Whether the rule is an end rule or not

Return type bool

property left_term: Any
Gets the non-terminal on the left of the rule

Returns **left_term** – The left non-terminal of the rule

Return type any

property non_terminals: Iterable[Any]
Gets the non-terminals used

Returns **non_terminals** – The non terminals used in this rule

Return type iterable of any

property production
The production

Returns **right_terms** – The production

Return type any

property right_term: Any
Gets the terminal on the right of the rule

Returns **right_term** – The right terminal of the rule

Return type any

property right_terms
The right terms

Returns **right_terms** – The right terms of the rule

Return type iterable of any

property terminals: AbstractSet[Any]
Gets the terminals used

Returns **terminals** – The terminals used in this rule

Return type set of any

class pyformlang.indexed_grammar.**IndexedGrammar**(*rules:* pyformlang.indexed_grammar.rules.Rules, *start_variable: Any = 'S'*)
Describes an indexed grammar.

Parameters

- **rules** (*Rules*) – The rules of the grammar, in reduced form put into a Rule
- **start_variable** (*Any, optional*) – The start symbol of the indexed grammar

get_generating_non_terminals() → AbstractSet[Any]
Get the generating symbols

Returns **generating** – The generating symbols from the start state

Return type set of any

get_reachable_non_terminals() → AbstractSet[Any]
Get the reachable symbols

Returns **reachables** – The reachable symbols from the start state

Return type set of any

intersection(*other: Any*) → *pyformlang.indexed_grammar.indexed_grammar.IndexedGrammar*
Computes the intersection of the current indexed grammar with the other object

>> indexed_grammar and regex

Parameters **other** (*any*) – The object to intersect with

Returns **i_grammar** – The indexed grammar which useless rules

Return type *IndexedGrammar*

Raises NotImplementedError – When trying to intersection with something else than a regular expression or a finite automaton

is_empty() → bool
Checks whether the grammar generates a word or not

Returns is_empty – Whether the grammar is empty or not

Return type bool

remove_useless_rules() → *pyformlang.indexed_grammar.indexed_grammar.IndexedGrammar*
Remove useless rules in the grammar

More precisely, we remove rules which do not contain only generating or reachable non terminals.

Returns i_grammar – The indexed grammar which useless rules

Return type *IndexedGrammar*

property terminals: Iterable[Any]
Get all the terminals in the grammar

Returns terminals – The terminals used in the rules

Return type iterable of any

class pyformlang.indexed_grammar.**ProductionRule**(*left*, *right*, *prod*)

Represents a production rule, i.e. a rule of the form: A[sigma] -> B[r sigma]

Parameters

- **left** (*any*) – The non-terminal on the left side of the rule, A here
- **right** (*any*) – The non-terminal on the right side of the rule, B here
- **prod** (*any*) – The terminal used in the rule, "r" here

property f_parameter
The f parameter

Returns f – The f parameter

Return type any

is_production() → bool
Whether the rule is a production rule or not

Returns is_production – Whether the rule is a production rule or not

Return type bool

property left_term: Any
Gets the non-terminal on the left side of the rule

Returns left_term – The left term of this rule

Return type any

property non_terminals: Iterable[Any]
Gets the non-terminals used in the rule

Returns non_terminals – The non terminals used in this rules

Return type any

property production: Any
Gets the terminal used in the production

Returns **production** – The production used in this rule

Return type any

property right_term: Any
Gets the non-terminal on the right side of the rule

Returns **right_term** – The right term used in this rule

Return type any

property right_terms
The right terms

Returns **right_terms** – The right terms of the rule

Return type iterable of any

property terminals: AbstractSet[Any]
Gets the terminals used in the rule

Returns **terminals** – The terminals used in this rule

Return type any

class pyformlang.indexed_grammar.Rules(*rules: Iterable[pyformlang.indexed_grammar.reduced_rule.ReducedRule]*, *optim: int = 7*)
Store a set of rules and manipulate them

Parameters

- **rules** (iterable of ReducedRule) – A list of all the rules
- **optim** (*int*) – Optimization of the order of the rules 0 -> given order 1 -> reverse order 2 -> order by core number 3 -> reverse order of core number 4 -> reverse order by arborescence 5 -> order by arborescence 6 -> order by number of edges 7 -> reverse order by number of edges 8 -> random order

add_production(*left: Any*, *right: Any*, *prod: Any*)

Add the production rule: left[sigma] -> right[prod sigma]

Parameters

- **left** (*any*) – The left non-terminal in the rule
- **right** (*any*) – The right non-terminal in the rule
- **prod** (*any*) – The production used in the rule

property consumption_rules: Dict[Any, Iterable[pyformlang.indexed_grammar.consumption_rule.ConsumptionRule]]
Gets the consumption rules

Returns **consumption_rules** – A dictionary contains the consumption rules gathered by consumed symbols

Return type dict of any to iterable of *ConsumptionRule*

property length: (<class 'int'>, <class 'int'>)
Get the total number of rules

Returns number_rules – A couple with first the number of non consumption rules and then the number of consumption rules

Return type couple of int

property non_terminals: List[Any]
Gets all the non-terminals used by all the rules

Returns non_terminals – The non terminals used in the rule

Return type iterable of any

property optim
Gets the optimization number

Returns non_consumption_rules – The optimization number

Return type int

remove_production(*left: Any*, *right: Any*, *prod: Any*)

Remove the production rule: left[sigma] -> right[prod sigma]

Parameters

- **left** (*any*) – The left non-terminal in the rule
- **right** (*any*) – The right non-terminal in the rule
- **prod** (*any*) – The production used in the rule

property rules: Iterable[pyformlang.indexed_grammar.reduced_rule.ReducedRule]
Gets the non consumption rules

Returns non_consumption_rules – The non consumption rules

Return type iterable of ReducedRule

property terminals: Iterable[Any]
Gets all the terminals used by all the rules

Returns terminals – The terminals used in the rules

Return type iterable of any

CHAPTER

FOUR

AUTHORS

4.1 Main Author

- Julien Romero <julien.romero@telecom-paris.fr>

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W