
FAdo Documentation

Release 2.0.4

Rogério Reis & Nelma Moreira

Mar 14, 2022

CONTENTS

1	What is FAdo?	3
1.1	Regular Languages	3
1.2	Finite Languages	4
1.3	Transducers	4
1.4	Codes	4
2	FAdo.fa	5
3	FAdo.conversions	43
4	FAdo.common	49
5	FAdo.fio	55
6	FAdo.reex	57
7	FAdo.transducers	105
8	FAdo.fl	113
9	FAdo.cgf	121
10	FAdo.rndfap	125
11	FAdo.rndadfa	127
12	FAdo.comboperations	131
13	FAdo.codes	135
14	FAdo.sst	147
15	FAdo.prax	153
16	FAdo.graphs	155
17	Small Tutorial	159
17.1	A small tutorial for FAdo	159
17.1.1	General conventions	159
17.1.2	Finite Automata	159
17.1.3	DFAs	160
17.1.4	NFAs	162

17.1.5	Converting between NFAs and DFAs	162
17.1.6	Regular Expressions	163
17.1.7	Converting Finite Automata to Regular Expressions	165
17.1.8	General Example	165
17.1.9	More classes and modules	166
18	Indices and tables	167
	Python Module Index	169
	Index	171

FAdo: Tools for Language Models Manipulation

Authors: Rogério Reis & Nelma Moreira

The support of transducers and all its operations, as well of Set Specifications, is a joint work with *Stavros Konstantinidis* (St. Mary's University, Halifax, NS, Canada) (<http://cs.smu.ca/~stavros/>).

Contributions by

- Marco Almeida
- Ivone Amorim
- Rafaela Bastos
- Miguel Ferreira
- Hugo Gouveia
- Rizó Isrof
- Eva Maia
- Casey Meijer
- Davide Nabais
- Meng Yang
- Joshua Young

Page of the project: <http://fado.dcc.fc.up.pt>.

Version: 2.0.4

Copyright: 1999-2022 Rogério Reis & Nelma Moreira {rogerio.reis,nelma.moreira}@fc.up.pt

Faculdade de Ciências da Universidade do Porto

Centro de Matemática da Universidade do Porto

Licence:

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your Option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

WHAT IS FADO?

The **FAdo** system aims to provide an open source extensible high-performance software library for the symbolic manipulation of automata and other models of computation.

To allow high-level programming with complex data structures, easy prototyping of algorithms, and portability (to use in computer grid systems for example), are its main features. Our main motivation is the theoretical and experimental research, but we have also in mind the construction of a pedagogical tool for teaching automata theory and formal languages.

1.1 Regular Languages

It currently includes most standard operations for the manipulation of regular languages. Regular languages can be represented by regular expressions (RegExp) or finite automata, among other formalisms. Finite automata may be deterministic (DFA), non-deterministic (NFA) or generalized (GFA). In **FAdo** these representations are implemented as Python classes.

Elementary regular languages operations as union, intersection, concatenation, complementation and reverse are implemented for each class. Also several combined operations are available for specific models.

Several conversions between these representations are implemented:

- NFA -> DFA: subset construction
- NFA -> RE: recursive method
- GFA -> RE: state elimination, with possible choice of state orderings
- RE -> NFA: Thompson method, Glushkov method, follow, Brzozowski, and partial derivatives.
- For DFAs several minimization algorithms are available: Moore, Hopcroft, and some incremental algorithms. Brzozowski minimization is available for NFAs.
- An algorithm for hyper-minimization of DFAs
- Language equivalence of two DFAs can be determined by reducing their correspondent minimal DFA to a canonical form, or by the Hopcroft and Karp algorithm.
- Enumeration of the first words of a language or all words of a given length (Cross Section)
- Some support for the transition semigroups of DFAs

1.2 Finite Languages

Special methods for finite languages are available:

- Construction of a ADFA (acyclic finite automata) from a set of words
- Minimization of ADFAs
- Several methods for ADFAs random generation
- Methods for deterministic cover finite automata (DCFA)

1.3 Transducers

Several methods for transducers in standard form (SFT) are available:

- Rational operations: union, inverse, reversal, composition, concatenation, Star
- Test if a transducer is functional
- Input intersection and Output intersection operations

1.4 Codes

A *language property* is a set of languages. Given a property specified by a transducer, several language tests are possible.

- Satisfaction i.e. if a language satisfies the property
- Maximality i.e. the language satisfies the property and is maximal
- Properties implemented by transducers include: input preserving, input altering, trajectories, and fixed properties
- Computation of the edit distance of a regular language, using input altering transducers

FADO.FA

Finite automata manipulation.

Deterministic and non-deterministic automata manipulation, conversion and evaluation.

class DFA

Class for Deterministic Finite Automata.

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **Initial** (*int*) – the initial state index.
- **Final** (*set*) – set of final states indexes.
- **delta** (*dict*) – the transition function.
- **delta_inv** (*dict*) – possible inverse transition map
- **i** (*bool*) – is inverse map computed?



Delta(*state, symbol*)

Evaluates the action of a symbol over a state

Parameters

- **state** (*int*) – state index
- **symbol** (*Any*) – symbol

Returns the action of symbol over state

Return type *int*

HKeqP(*other, strict=True*)

Tests the DFA's equivalence using Hopcroft and Karp's state equivalence algorithm

Parameters

- **other** –
- **strict** –

Returns bool

See also:

J. E. Hopcroft and r. M. Karp. A Linear Algorithm for Testing Equivalence of Finite Automata. TR 71–114. U. California. 1971

Attention: The automaton must be complete.

MyhillNerodePartition()

Myhill-Nerode partition, Moore’s way

New in version 1.3.5.

Attention: No state should be named with `DeadName`. This states is removed from the obtained partition.

See also:

F.Bassino, J.David and C.Nicaud, On the Average Complexity of Moores’s State Minimization Algorihm, Symposium on Theoretical Aspects of Computer Science

aEquiv()

Computes almost equivalence, used by hyperMinimal

Returns partition of states

Return type dict

Note: may be optimized to avoid dupped

addTransition(sti1, sym, sti2)

Adds a new transition from sti1 to sti2 consuming symbol sym.

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** (*Any*) – symbol consumed

Raises *DFAnotNFA* (page 49) – if one tries to add a non-deterministic transition

compat(s1, s2, data)

Tests compatibility between two states.

Parameters

- **data** –
- **s1** (*int*) – state index
- **s2** (*int*) – state index

Return type bool

complete(*dead*='DeaD')

Transforms the automata into a complete one. If sigma is empty nothing is done.

Parameters **dead** (*str*) – dead state name

Returns the complete FA

Return type *DFA* (page 5)

Note: Adds a dead state (if necessary) so that any word can be processed with the automata. The new state is named dead, so this name should never be used for other purposes.

<p>Attention: The object is modified in place.</p>

Changed in version 1.0.

completeMinimal()

Completes a DFA assuming it is a minimal and avoiding de destruction of its minimality If the automaton is not complete, all the non-final states are checked to see if they are not already a dead state. Only in the negative case a new (dead) state is added to the automaton.

Return type *DFA* (page 5)

<p>Attention: The object is modified in place. If the alphabet is empty nothing is done</p>
--

completeP()

Checks if it is a complete FA (if delta is total)

Returns bool

completeProduct(*other*)

Product structure

Parameters **other** (*DFA* (page 5)) – the other DFA

Return type *DFA* (page 5)

computeKernel()

The Kernel of a IC DFA is the set of states that accept a non-finite language.

Returns triple (comp, center , mark) where comp are the strongly connected components, center the set of center states and mark the kernel states

Return type tuple

concat(*fa2*, *strict=False*)

Concatenation of two DFAs. If DFAs are not complete, they are completed.

Parameters

- **strict** (*bool*) – should alphabets be checked?
- **fa2** (*DFA* (page 5)) – the second DFA

Returns the result of the concatenation

Return type *DFA* (page 5)

Raises *DFAdifferentSigma* (page 49) – if alphabet are not equal

concatI(*fa2*, *strict=False*)

Concatenation of two DFAs.

Parameters

- **fa2** ([DFA](#) (page 5)) – the second DFA
- **strict** ([bool](#)) – should alphabets be checked?

Returns the result of the concatenation

Return type [DFA](#) (page 5)

Raises [DFAdifferentSigma](#) (page 49) – if alphabet are not equal

New in version 0.9.5.

Note: this is to be used with non-complete DFAs

delTransition(*sti1*, *sym*, *sti2*, *_no_check=False*)

Remove a transition if existing and perform cleanup on the transition function's internal data structure.

Parameters

- **sti1** ([int](#)) – state index of departure
- **sym** (*Any*) – symbol consumed
- **sti2** ([int](#)) – state index of arrival
- **_no_check** ([bool](#)) – use unsecure code?

Note: Unused alphabet symbols will be discarded from sigma.

deleteStates(*del_states*)

Delete given iterable collection of states from the automaton.

Parameters **del_states** – collection of state indexes

Note: in-place action

Note: delta function will always be rebuilt, regardless of whether the states list to remove is a suffix, or a sublist, of the automaton's states list.

static deterministicP()

Yes it is deterministic!

Return type [bool](#)

dist()

Evaluate the distinguishability language for a DFA

Return type [DFA](#) (page 5)

See also:

Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100

New in version 0.9.8.

distMin()

Evaluates the list of minimal words that distinguish each pair of states

Return type set of minimal distinguishing words ([FL](#) (page 118))

New in version 0.9.8.

Attention: If the DFA is not minimal, the method loops forever

distR()

Evaluate the right distinguishability language for a DFA

Return type [DFA](#) (page 5)

..seealso:: Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100

distRMin()

Compute distRMin for DFA

Return type [FL](#) (page 118)

..seealso:: Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100

distTS()

Evaluate the two-sided distinguishability language for a DFA

Return type [DFA](#) (page 5)

..seealso:: Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100

dup()

Duplicate the basic structure into a new DFA. Basically a copy.deep.

Return type [DFA](#) (page 5)

enumDFA(n=None)

returns the set of words of words of length up to n accepted by self :param int n: highest length or all words if finite

Return type list of strings or None

equal(other)

Verify if the two automata are equivalent. Both are verified to be minimum and complete, and then one is matched against the other... Doesn't destroy either dfa...

Parameters **other** ([DFA](#) (page 5)) – the other DFA

Return type [bool](#)

evalSymbol(init, sym)

Returns the state reached from given state through a given symbol.

Parameters

- **init** ([int](#)) – set of current states indexes

- **sym** (*str*) – symbol to be consumed

Returns reached state

Return type `int`

Raises

- **DFASymbolUnknown** (page 49) – if symbol not in alphabet
- **DFAstopped** (page 49) – if transition function is not defined for the given input

evalSymbolI(*init*, *sym*)

Returns the state reached from a given state.

Parameters

- **init** (*init*) – current state
- **sym** (*str*) – symbol to be consumed

Returns reached state or -1

Return type set of `int`

Raises **DFASymbolUnknown** (page 49) – if symbol not in alphabet

New in version 0.9.5.

Note: this is to be used with non-complete DFAs

evalSymbolL(*ls*, *sym*)

Returns the set of states reached from a given set of states through a given symbol

Parameters

- **ls** (*set of int*) – set of states indexes
- **sym** (*str*) – symbol to be read

Returns set of reached states

Return type set of `int`

evalSymbolLI(*ls*, *sym*)

Returns the set of states reached from a given set of states through a given symbol

Parameters

- **ls** (*set of int*) – set of current states
- **sym** (*str*) – symbol to be consumed

Returns set of reached states

Return type set of `int`

New in version 0.9.5.

Note: this is to be used with non-complete DFAs

evalWord(*wrđ*)

Evaluates a word

Parameters **wrđ** (**Word** (page 51)) – word

Returns final state or None

Return type `int` | None

New in version 1.3.3.

evalWordP(*word*, *initial=None*)

Verifies if the DFA recognises a given word

Parameters

- **word** (*list of symbols.*) – word to be recognised
- **initial** (*int*) – starting state index

Return type `bool`

finalCompP(*s*)

Verifies if there is a final state in strongly connected component containing *s*.

Parameters **s** (*int*) – state

Returns 1 if yes, 0 if no

hasTrapStateP()

Tests if the automaton has a dead trap state

Return type `bool`

New in version 1.1.

hyperMinimal(*strict=False*)

Hyperminimization of a minimal DFA

Parameters **strict** (*bool*) – if *strict=True* it first minimizes the DFA

Returns an hyperminimal DFA

Return type *DFA* (page 5)

See also:

M. Holzer and A. Maletti, An $n \log n$ Algorithm for Hyper-Minimizing a (Minimized) Deterministic Automata, TCS 411(38-39): 3404-3413 (2010)

Note: if *strict=False* minimality is assumed

inDegree(*st*)

Returns the in-degree of a given state in an FA

Parameters **st** (*int*) – index of the state

Return type `int`

infix()

Returns a dfa that recognizes $\text{infix}(L(a))$

Return type *DFA* (page 5)

initialComp()

Evaluates the connected component starting at the initial state.

Returns list of state indexes in the component

Return type list of `int`

initialP(*state*)

Tests if a state is initial

Parameters *state* (*int*) – state index**Return type** *bool***initialSet**()

The set of initial states

Returns the set of the initial states**Return type** *set***joinStates**(*lst*)

Merge a list of states.

Parameters *lst* (*iterable of state indexes.*) – set of equivalent states**makeReversible**()

Make a DFA reversible (if possible)

See also:

M.Holzer, s. Jakobi, M. Kutrib ‘Minimal Reversible Deterministic Finite Automata’

Return type *DFA* (page 5)**make_prefix_free**()

Turns a DFA in a prefix-free automaton deleting all outgoing transitions from final states

Return type *DFA* (page 5)

New in version 2.0.3.

markNonEquivalent(*s1, s2, data*)Mark states with indexes *s1* and *s2* in given map as non-equivalent states. If any back-effects exist, apply them.**Parameters**

- *s1* (*int*) – one state’s index
- *s2* (*int*) – the other state’s index
- *data* – the matrix relating *s1* and *s2*

mergeStates(*f, t*)

Merge the first given state into the second. If the first state is an initial state the second becomes the initial state.

Parameters

- *f* (*int*) – index of state to be absorbed
- *t* (*int*) – index of remaining state

Attention: It is up to the caller to remove the disconnected state. This can be achieved with ``trim()`.**minimal**(*method='minimalHopcroft', complete=True*)

Evaluates the equivalent minimal complete DFA

Parameters

- **method** – method to use in the minimization
- **complete** (*bool*) – should the result be completed?

Returns equivalent minimal DFA

Return type *DFA* (page 5)

minimalHopcroft()

Evaluates the equivalent minimal complete DFA using Hopcroft algorithm

Returns equivalent minimal DFA

Return type *DFA* (page 5)

See also:

John Hopcroft, An $n \log \{n\}$ algorithm for minimizing states in a finite automaton. The Theory of Machines and Computations. AP. 1971

minimalHopcroftP()

Tests if a DFA is minimal

Return type *bool*

minimalIncremental(*minimal_test=False*)

Minimizes the DFA with an incremental method using the Union-Find algorithm and memoized non-equivalence intermediate results

Parameters **minimal_test** (*bool*) – starts by verifying that the automaton is not minimal?

Returns equivalent minimal DFA

Return type *DFA* (page 5)

See also:

M. Almeida and N. Moreira and r. Reis. Incremental DFA minimisation. CIAA 2010. LNCS 6482. pp 39-48. 2010

minimalIncrementalP()

Tests if a DFA is minimal

Return type *bool*

minimalMoore()

Evaluates the equivalent minimal automata with Moore's algorithm

See also:

John E. Hopcroft and Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, AW, 1979

Returns minimal complete DFA

Return type *DFA* (page 5)

minimalMooreSq()

Evaluates the equivalent minimal complete DFA using Moore's (quadratic) algorithm

See also:

John E. Hopcroft and Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, AW, 1979

Returns equivalent minimal DFA

Return type [DFA](#) (page 5)

minimalMooreSqP()

Tests if a DFA is minimal using the quadratic version of Moore's algorithm

Return type [bool](#)

minimalNCompleteP()

Tests if a non necessarily complete DFA is minimal, i.e., if the DFA is non-complete, if the minimal complete has only one more state.

Returns True if not minimal

Return type [bool](#)

Attention: obsolete: use minimalP
--

minimalNotEquivP()

Tests if the DFA is minimal by computing the set of distinguishable (not equivalent) pairs of states

Return type [bool](#)

minimalP(*method*='minimalMooreSq')

Tests if the DFA is minimal

Parameters **method** – the minimization algorithm to be used

Return type [bool](#)

..note: if DFA non-complete test if complete minimal has one more state

minimalWatson(*test_only*=False)

Evaluates the equivalent minimal complete DFA using Waton's incremental algorithm

Parameters **test_only** ([bool](#)) – is it only to test minimality

Returns equivalent minimal DFA

Return type [DFA](#) (page 5)

Raises [DFAnotComplete](#) (page 49) – if automaton is not complete

..attention:: automaton must be complete

minimalWatsonP()

Tests if a DFA is minimal using Watson's incremental algorithm

Return type [bool](#)

notequal(*other*)

Test non equivalence of two DFAs

Parameters **other** ([DFA](#) (page 5)) – the other DFA

Return type [bool](#)

orderedStrConnComponents()

Topological ordered list of strong components

New in version 1.3.3.

Return type [list](#)

pairGraph()

Returns pair graph

Return type DiGraphVM

See also:

A graph theoretic approach to automata minimality. Antonio Restivo and Roberto Vaglica. Theoretical Computer Science, 429 (2012) 282-291. doi:10.1016/j.tcs.2011.12.049 Theoretical Computer Science, 2012 vol. 429 (C) pp. 282-291. http://dx.doi.org/10.1016/j.tcs.2011.12.049

possibleToReverse()

Tests if language is reversible

New in version 1.3.3.

pref()

Returns a dfa that recognizes pref(L(self))

Return type [DFA](#) (page 5)

New in version 1.1.

prefix_free_p()

Checks if a DFA is prefix-free :rtype: bool

New in version 2.0.3.

print_data(data)

Prints table of compatibility (in the context of the minimalization algorithm).

Parameters **data** – data to print

product(other)

Returns a DFA resulting of the simultaneous execution of two DFA. No final states set.

Note: this is a fast version of the method. The resulting state names are not meaningful.

Parameters **other** – the other DFA

Return type [DFA](#) (page 5)

productSlow(other, complete=True)

Returns a DFA resulting of the simultaneous execution of two DFA. No final states set.

Note: this is a slow implementation for those that need meaningful state names

New in version 1.3.3.

Parameters

- **other** – the other DFA
- **complete** (*bool*) – evaluate product as a complete DFA

Return type [DFA](#) (page 5)

reorder(dicti)

Reorders states according to given dictionary. Given a dictionary (not necessarily complete)... reorders states accordingly.

Parameters **dicti** (*dict*) – reorder dictionary

reverseTransitions(*rev*)

Evaluate reverse transition function.

Parameters **rev** (*DFA* (page 5)) – DFA in which the reverse function will be stored

reversibleP()

Test if an automaton is reversible

Return type *bool*

sMonoid()

Evaluation of the syntactic monoid of a DFA

Returns the semigroup

Return type *SSemiGroup*

sSemigroup()

Evaluation of the syntactic semigroup of a DFA

Returns the semigroup

Return type *SSemiGroup*

shuffle(*other*, *strict=False*)

CShuffle of two languages: L1 W L2

Parameters

- **other** (*DFA* (page 5)) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA* (page 5)

See also:

C. Câmpeanu, K. Salomaa and s. Yu, *Tight lower bound for the state complexity of CShuffle of regular languages*. J. Autom. Lang. Comb. 7 (2002) 303–310.

simDiff(*other*)

Symetrical difference

Parameters **other** –

Returns

sop(*other*)

Strange operation

Parameters **other** (*DFA* (page 5)) – the other automaton

Return type *DFA* (page 5)

See also:

Nelma Moreira, Giovanni Pighizzini, and Rogério Reis. Universal disjunctive concatenation and star. In Jeffrey Shallit and Alexander Okhotin, editors, Proceedings of the 17th Int. Workshop on Descriptive Complexity of Formal Systems (DCFS15), number 9118 in LNCS, pages 197–208. Springer, 2015.

New in version 1.2b2.

star(*flag=False*)

Star of a DFA. If the DFA is not complete, it is completed.

..versionchanged: 0.9.6

Parameters **flag** (*bool*) – plus instead of star

Returns the result of the star

Return type *DFA* (page 5)

starI()

Star of an incomplete DFA.

Returns the Kleene closure DFA

Return type *DFA* (page 5)

stateChildren(*state*, *strict=False*)

Set of children of a state

Parameters

- **strict** (*bool*) – if not strict a state is never its own child even if a self loop is in place
- **state** (*int*) – state id queried

Returns map children -> multiplicity

Return type dictionary

stronglyConnectedComponents()

Dummy method that uses the NFA conterpart

New in version 1.3.3.

Return type *list*

subword()

A dfa that recognizes subword(L(self))

Return type *DFA* (page 5)

New in version 1.1.

succintTransitions()

Collects the transition information in a compact way suitable for graphical representation.

Returns list of tuples

Return type *list*

New in version 0.9.8.

suff()

Returns a dfa that recognizes suff(L(self))

Return type *DFA* (page 5)

New in version 0.9.8.

syncPower()

Evaluates the Power automata for the action of each symbol

Returns The Power automata being the set of all states the initial state and all singleton states
final

Return type *DFA* (page 5)

toADFA()

Try to convert DFA to ADFA

Returns the same automaton as a ADFA

Return type *ADFA* (page 113)

Raises *notAcyclic* (page 52) – if this is not an acyclic DFA

New in version 1.2.

Changed in version 1.2.1.

toDFA()

Dummy function. It is already a DFA

Returns a self deep copy

Return type *DFA* (page 5)

toNFA()

Migrates a DFA to a NFA as dup()

Returns DFA seen as new NFA

Return type *NFA* (page 27)

transitions()

Iterator over transitions :rtype: symbol, int

transitionsA()

Iterator over transitions :rtype: symbol, int

uniqueRepr()

Normalise unique string for the string icdfa's representation. .. seealso:: TCS 387(2):93-102, 2007 <https://www.dcc.fc.up.pt/~nam/publica/tcsamr06.pdf>

Returns normalised representation

Return type *list*

Raises *DFAnotComplete* (page 49) – if DFA is not complete

universalP(minimal=False)

Checks if the automaton is universal through minimisation

Parameters *minimal* (*bool*) – is the automaton already minimal?

Return type *bool*

unmark()

Unmarked NFA that corresponds to a marked DFA: in which each alphabetic symbol is a tuple (symbol, index)

Returns a NFA

Return type *NFA* (page 27)

usefulStates(initial_states=None)

Set of states reachable from the given initial state(s) that have a path to a final state.

Parameters *initial_states* (*iterable of int*) – starting states

Returns set of state indexes

Return type set of int

witness()

Witness of non emptiness

Returns word

Return type *str*

witnessDiff(*other*)

Returns a witness for the difference of two DFAs and:

0	if the witness belongs to the other language
1	if the witness belongs to the self language

Parameters **other** ([DFA](#) (page 5)) – the other DFA

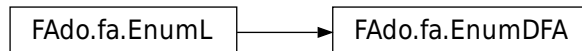
Returns a witness word

Return type list of symbols

Raises [DFAequivalent](#) (page 49) – if automata are equivalent

class EnumDFA(*aut, store=False*)

Class for enumerating languages defined by DFAs

**fillStack**(*w*)

Computes S_1, \dots, S_{n-1} where S_i is the set of (n-i)-complete states reachable from S_{i-1}

Parameters **w** – word

initStack()

Initializes the stack with initial states

minWordT(*n*)

Computes for each state the minimal word of length $i < n$ accepted by the automaton. Stores the values in tmin

Parameters **n** ([int](#)) – length of the word

Note: Makinen algorithm for DFAs

nextWord(*w*)

Given an word, returns next word on the nth cross-section of $L(\text{aut})$ according to the radix order

Parameters **w** ([str](#)) – word

Return type [str](#)

class EnumL(*aut, store=False*)

Class for enumerate FA languages See: Efficient enumeration of words in regular languages, M. Ackerman and J. Shallit, Theor. Comput. Sci. 410, 37, pp 3461-3470. 2009. <http://dx.doi.org/10.1016/j.tcs.2009.03.018>

Variables

- **aut** ([FA](#) (page 21)) – Automaton of the language

- **tmin** (*dict*) – table for minimal words for each *s* in *aut.States*
- **Words** (*list*) – list of words (if stored)
- **sigma** (*list*) – alphabet
- **stack** (*deque*) –

FAdo.fa.EnumL

New in version 0.9.8.

enum(*m*)

Enumerates the first *m* words of *L(A)* according to the lexicographic order if there are at least *m* words. Otherwise, enumerates all words accepted by *A*.

Parameters *m* (*int*) – max number of words

enumCrossSection(*n*)

Enumerates the *n*th cross-section of *L(A)*

Parameters *n* (*int*) – nonnegative integer

abstract fillStack(*w*)

Abstract method :param str *w*: :type *w*: str

iCompleteP(*i*, *q*)

Tests if state *q* is *i*-complete

Parameters

- *i* (*int*) – int
- *q* (*int*) – state index

abstract initStack()

Abstract method

minWord(*m*)

Computes the minimal word of length *m* accepted by the automaton :param *m*: :type *m*: int

abstract minWordT(*n*)

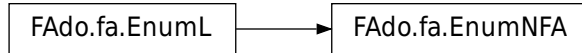
Abstract method :param int *n*: :type *n*: int

abstract nextWord(*w*)

Abstract method :param *w*: :type *w*: str

class EnumNFA(*aut*, *store=False*)

Class for enumerating languages defined by NFAs



fillStack(*w*)

Computes S_1, \dots, S_{n-1} where S_i is the set of $(n-i)$ -complete states reachable from S_{i-1}

Parameters *w* – word

initStack()

Initializes the stack with initial states

minWordT(*n*)

Computes for each state the minimal word of length $i \leq n$ accepted by the automaton. Stores the values in *tmin*.

Parameters *n* (*int*) – length of the word

nextWord(*w*)

Given an word, returns next word in the the *n*th cross-section of $L(\text{aut})$ according to the radix order

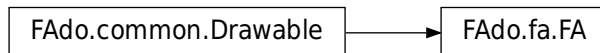
Parameters *w* (*str*) – word

class FA

Base class for Finite Automata. This is just an abstract class. **Not to be used directly!!**

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **Initial** (*int*) – the initial state index.
- **Final** (*set*) – set of final states indexes.
- **delta** (*dict*) – the transition function.



addFinal(*stateindex*)

A new state is added to the already defined set of final states.

Parameters **stateindex** (*int*) – index of the new final state.

addSigma(*sym*)

Adds a new symbol to the alphabet.

Parameters **sym** (*str*) – symbol to be added

Raises *DFAepsilonRedefinition* (page 49) – if sym is Epsilon

Note:

- There is no problem with duplicate symbols because sigma is a Set.
 - No symbol Epsilon can be added.
-

addState(*name=None*) → *int*

Adds a new state to an FA. If no name is given a new name is created.

Parameters **name** (Object, optional) – Name of the state to be added.

Returns Current number of states (the new state index).

Return type *int*

Raises *DuplicateName* (page 50) – if a state with that name already exists

conjunction(*other*)

A simple literate invocation of __and__

Parameters **other** (*FA* (page 21)) – right-hand operand.

Returns Intersection of self and other.

Return type *FA* (page 21)

New in version 0.9.6.

countTransitions()

Evaluates the size of FA transitionwise

Returns the number of transitions

Return type *int*

Changed in version 1.0.

delFinal(*st*)

Deletes a state from the final states list

Parameters **st** (*int*) – state to be marked as not final.

delFinals()

Deletes all the information about final states.

deleteState(*sti: int*)

Remove the given state and the transitions related with that state.

Parameters **sti** (*int*) – index of the state to be removed

Raises *DFAstateUnknown* (page 49) – if state index does not exist

disj(*other*)

Another simple literate invocation of __or__

Parameters **other** (*FA* (page 21)) – the other FA.

Returns Union of self and other.

Return type *FA* (page 21)

New in version 0.9.6.

disjunction(*other*)

A simple literate invocation of `__or__`

Parameters **other** ([FA](#) (page 21)) – the other FA

Returns Union of self and other.

Return type [FA](#) (page 21)

dotDrawState(*sti, sep='\n', _strict=False, _maxlblsz=6*)

Draw a state in dot format

Parameters

- **sti** ([int](#)) – index of the state.
- **sep** ([str](#), optional) – separator.
- **_maxlblsz** ([int](#), optional) – max size of labels before getting removed
- **_strict** ([bool](#), optional) – use limitations of label size

Returns string to be added to the dot file.

Return type [str](#)

static dotDrawTransition(*st1, label, st2, sep='\n'*)

Draw a transition in dot format

Parameters

- **st1** ([str](#)) – departing state
- **label** ([str](#)) – label
- **st2** ([str](#)) – arriving state
- **sep** ([str](#)) – separator

Return type [str](#)

dotFormat(*size='20,20', filename=None, direction='LR', strict=False, maxlblsz=6, sep='\n'*) → [str](#)

A dot representation

Parameters

- **direction** ([str](#)) – direction of drawing - “LR” or “RL”
- **size** ([str](#)) – size of image
- **filename** ([str](#)) – output file name
- **sep** ([str](#)) – line separator
- **maxlblsz** ([int](#)) – max size of labels before getting removed
- **strict** ([bool](#)) – use limitations of label sizes

Returns the dot representation

Return type [str](#)

New in version 0.9.6.

Changed in version 1.2.1.

eliminateDeadName()

Eliminates dead state name (`common.DeadName`) renaming the state

Returns `self`

Return type [DFA](#) (page 5)

Attention: works inplace

New in version 1.2.

equivalentP(*other*)

Test equivalence between automata

Parameters **other** ([FA](#) (page 21)) – the other automata

Return type `bool`

New in version 0.9.6.

abstract evalSymbol(*stil*, *sym*)

Evaluation of a single symbol

finalP(*state*: `int`) → `bool`

Tests if a state is final

Parameters **state** (`int`) – state index.

Returns is the state final?

Return type `bool`

finalsP(*states*: `set`) → `bool`

Tests if all the states in a set are final

Parameters **states** (`set`) – set of state indexes.

Returns are all the states final?

Return type `bool`

New in version 1.0.

hasStateIndexP(*st*: `int`) → `bool`

Checks if a state index pertains to an FA

Parameters **st** (`int`) – index of the state.

Return type `bool`

images(*sti*, *c*)

The set of images of a state by a symbol

Parameters

- **sti** (`int`) – state
- **c** (`object`) – symbol

Return type iterable

indexList(*lstn*)

Converts a list of stateNames into a set of stateIndexes.

Parameters **lstn** (`list`) – list of names

Returns the list of state indexes

Return type `set`

Raises [DFAstateUnknown](#) (page 49) – if a state name is unknown

initialP(*state: int*) → bool

Tests if a state is initial

Parameters **state** – state index

Returns is the state initial?

Return type bool

initialSet()

The set of initial states

Returns set of States.

Return type set

inputS(*i*)

Input labels coming out of state *i*

Parameters **i** (*int*) – state

Returns set of input labels

Return type set of str

New in version 1.0.

noBlankNames()

Eliminates blank names

Returns self

Return type *FA* (page 21)

Attention: in place transformation

plus()

Plus of a FA (star without the adding of epsilon)

New in version 0.9.6.

renameState(*st, name*)

Rename a given state.

Parameters

- **st** (*int*) – state index.
- **name** (*object*) – name.

Returns self.

Return type *FA* (page 21)

Note: Deals gracefully both with int and str names in the case of name collision.

Attention: the object is modified in place

renameStates(*name_list=None*)

Renames all states using a new list of names.

Parameters `name_list` (*list*) – list of new names.

Returns `self`.

Return type *FA* (page 21)

Raises *DFAerror* (page 49) – if provided list is too short.

Note: If no list of names is given, state indexes are used.

Attention: the object is modified in place

reversal()

Returns a NFA that recognizes the reversal of the language

Returns NFA recognizing reversal language

Return type *NFA* (page 27)

same_nullability(*s1: int, s2: int*) → *bool*

Tests if this two states have the same nullability

Parameters

- *s1* (*int*) – state index.
- *s2* (*int*) – state index.

Returns have the states the same nullability?

Return type *bool*

setFinal(*statelist*)

Sets the final states of the FA

Parameters `statelist` (*int / list / set*) – a list (or set) of final states indexes.

Caution: Erases any previous definition of the final state set.

setInitial(*stateindex*)

Sets the initial state of a FA

Parameters `stateindex` (*int*) – index of the initial state.

setSigma(*symbol_set*)

Defines the alphabet for the FA.

Parameters `symbol_set` (*list / set*) – alphabet symbols

stateAlphabet(*sti: int*) → *list*

Active alphabet for this state

Parameters `sti` (*int*) – state

Return type *list*

stateIndex(*name, auto_create=False*)

Index of given state name.

Parameters

- **name** (*object*) – name of the state.
- **auto_create** (*bool*, optional) – flag to create state if not already done.

Returns state index

Return type *int*

Raises *DFastateUnknown* (page 49) – if the state name is unknown and autoCreate==False

Note: Replaces stateName

Note: If the state name is not known and flag is set creates it on the fly

New in version 1.0.

stateName(*name*, *auto_create=False*)

Index of given state name.

Parameters

- **name** (*object*) – name of the state
- **auto_create** (*bool*, optional) – flag to create state if not already done

Returns state index

Return type *int*

Raises *DFastateUnknown* (page 49) – if the state name is unknown and autoCreate==False

Deprecated since version 1.0: Use: *stateIndex()* (page 26) instead

Deprecated since version 1.0: Use the stateIndex() function instead

abstract succinctTransitions()

Collapsed transitions

union(*other*)

A simple literate invocation of __or__

Parameters **other** (*FA* (page 21)) – right-hand operand.

Returns Union of self and other.

Return type *FA* (page 21)

words(*stringo=True*)

Lexicographical word generator

Parameters **stringo** (*bool*, optional) – are words strings? Default is True.

Yields *Word* – the next word generated.

Attention: Does not generate the empty word

New in version 0.9.8.

class NFA

Class for Non-deterministic Finite Automata (epsilon-transitions allowed).

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **Initial** (*set*) – initial state indexes.
- **Final** (*set*) – set of final states indexes.
- **delta** (*dict*) – the transition function.



HKeqP(*other*, *strict=True*)

Test NFA equivalence with extended Hopcroft-Karp method

Parameters

- **other** (*NFA* (page 27)) –
- **strict** (*bool*) – if True checks for same alphabets

Return type *bool*

See also:

J. E. Hopcroft and r. M. Karp. A Linear Algorithm for Testing Equivalence of Finite Automata. TR 71–114. U. California. 1971

addEpsilonLoops()

Add epsilon loops to every state

Attention: in-place modification

New in version 1.0.

addInitial(*stateindex*)

Add a new state to the set of initial states.

Parameters **stateindex** (*int*) – index of new initial state

addTransition(*sti1*, *sym*, *sti2*)

Adds a new transition. Transition is from *sti1* to *sti2* consuming symbol *sym*. *sti2* is a unique state, not a set of them.

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** (*str*) – symbol consumed

addTransitionQ(*srci*, *dest*, *symb*, *qfuture*, *qpast*)

Add transition to the new transducer instance.

Parameters

- **qpast** (*set*) – past queue
- **qfuture** (*set*) – future queue
- **symb** – symbol
- **dest** (*int*) – destination state
- **srci** (*int*) – source state

New in version 1.0.

autobisimulation()

Largest right invariant equivalence between states of the NFA

Returns Incomplete equivalence relation (transitivity, and reflexivity not calculated) as a set of unordered pairs of states

Return type *set*

See also:

L. Ilie and S. Yu, Follow automata Inf. Comput. 186 - 1, pp 140-162, 2003

autobisimulation2()

Alternative space-efficient definition of NFA.autobisimulation.

Returns Incomplete equivalence relation (reflexivity, symmetry, and transitivity not calculated) as a set of pairs of states

Return type *list*

closeEpsilon(st)

Add all non epsilon transitions from the states in the epsilon closure of given state to given state.

Parameters **st** (*int*) – state index

Attention: in-place modification

computeFollowNames()

Computes the follow set to use in names

Return type *list*

concat(other, middle='middle')

Concatenation of NFA

Parameters

- **middle** (*str*) – glue state name
- **other** (*FA* (page 21)) – the other NFA

Returns the result of the concatenation

Return type *NFA* (page 27)

countTransitions()

Count the number of transitions of a NFA

Return type *int*

delTransition(sti1, sym, sti2, _no_check=False)

Remove a transition if existing and perform cleanup on the transition function's internal data structure.

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** – symbol consumed
- **_no_check** (*bool*) – dismiss secure code

Note: unused alphabet symbols will be discarded from sigma.

deleteStates(*del_states*)

Delete given iterable collection of states from the automaton.

Parameters **del_states** (*set / list*) – collection of int representing states

Note: delta function will always be rebuilt, regardless of whether the states list to remove is a suffix, or a sublist, of the automaton’s states list.

detSet(*generic=False*)

Computes the determination uppon a followFromPosition result

Return type *NFA* (page 27)

deterministicP()

Verify whether this NFA is actually deterministic

Return type *bool*

dotFormat(*size='20,20', filename=None, direction='LR', strict=False, maxlblsz=6, sep='\n') → str*

A dot representation

Parameters

- **direction** (*str*) – direction of drawing - “LR” or “RL”
- **size** (*str*) – size of image
- **filename** (*str*) – output file name
- **sep** (*str*) – line separator
- **maxlblsz** (*int*) – max size of labels before getting removed
- **strict** (*bool*) – use limitations of label sizes

Returns the dot representation

Return type *str*

New in version 0.9.6.

Changed in version 1.2.1.

dup()

Duplicate the basic structure into a new NFA. Basically a copy.deep.

Return type *NFA* (page 27)

elimEpsilon()

Eliminate epsilon-transitions from this automaton.

:rtype : *NFA*

Attention: performs in place modification of automaton

Changed in version 1.1.1.

eliminateEpsilonTransitions()

Eliminates all epsilon-transitions with no state addition

Attention: in-place modification

eliminateTSymbol(symbol)

Delete all transitions through a given symbol

Parameters **symbol** (*str*) – the symbol to be excluded from delta

Attention: in-place modification

New in version 0.9.6.

enumNFA(n=None)

The set of words of length up to n accepted by self

Parameters **n** (*int*) – highest length or all words if finite

Returns list of strings or None

Return type *list*

epsilonClosure(st)

Returns the set of states epsilon-connected to from given state or set of states.

Parameters **st** (*int / set*) – state index or set of state indexes

Returns the list of state indexes epsilon connected to st

Return type *set*

Attention: st must exist beforehand.

epsilonP()

Whether this NFA has epsilon-transitions

Return type *bool*

epsilonPaths(start, end)

All states in all paths (DFS) through empty words from a given starting state to a given ending state.

Parameters

- **start** (*int*) – start state
- **end** (*int*) – end state

Returns states in epsilon paths from start to end

Return type *set*

equivReduced(equiv_classes)

Equivalent NFA reduced according to given equivalence classes.

Parameters `equiv_classes` (*UnionFind*) – Equivalence classes

Returns Equivalent NFA

Return type *NFA* (page 27)

evalSymbol(*stil*, *sym*)

Set of states reachable from given states through given symbol and epsilon closure.

Parameters

- **stil** (*set/list*) – set of current states
- **sym** (*str*) – symbol to be consumed

Returns set of reached state indexes

Return type *set*

Raises *DFASymbolUnknown* (page 49) – if symbol is not in alphabet

evalWordP(*word*)

Verify if the NFA recognises given word.

Parameters **word** (*str*) – word to be recognised

Return type *bool*

finalCompP(*s*)

Verify whether there is a final state in strongly connected component containing given state.

Parameters **s** (*int*) – state index

Return type *bool*

followFromPosition()

computes follow automaton from a Position automaton

Return type *NFA* (page 27)

half()

Half operation

Return type *NFA* (page 27)

New in version 0.9.6.

hasTransitionP(*state*, *symbol=None*, *target=None*)

Whether there's a transition from given state, optionally through given symbol, and optionally to a specific target.

Parameters

- **state** (*int*) – source state
- **symbol** (*str*) – (optional) transition symbol
- **target** (*int*) – (optional) target state

Returns if there is a transition

Return type *bool*

homogeneousFinalityP()

Tests if states have incoming transitions from states with different finalities

Return type *bool*

homogenousP(*x*)

Whether this NFA is homogenous; that is, for all states, whether all incoming transitions to that state are through the same symbol.

Parameters *x* – dummy parameter to agree with the method in DFAr

Return type `bool`

initialComp()

Evaluate the connected component starting at the initial state.

Returns list of state indexes in the component

Return type list of int

lEquivNFA()

Equivalent NFA obtained from merging equivalent states from autobisimulation of this NFA's reversal.

Return type *NFA* (page 27)

Note: returns copy of self if autobisimulation renders no equivalent states.

lrEquivNFA()

Equivalent NFA obtained from merging equivalent states from autobisimulation of this NFA, and from autobisimulation of its reversal; i.e., merges all states that are equivalent w.r.t. the largest right invariant and largest left invariant equivalence relations.

Return type *NFA* (page 27)

Note: returns copy of self if autobisimulations render no equivalent states.

minimal()

Evaluates the equivalent minimal DFA

Returns equivalent minimal DFA

Return type *DFA* (page 5)

minimalDFA()

Evaluates the equivalent minimal complete DFA

Returns equivalent minimal DFA

Return type *DFA* (page 5)

product(*other*)

Returns a NFA (skeleton) resulting of the simultaneous execution of two DFA.

Parameters *other* (*NFA* (page 27)) – the other automata

Return type *NFA* (page 27)

Note: No final states are set.

Attention:

- the name `EmptySet` is used in a unique special state name

- the method uses 3 internal functions for simplicity of code (really!)

rEquivNFA()

Equivalent NFA obtained from merging equivalent states from autobisimulation of this NFA.

Return type *NFA* (page 27)

Note: returns copy of self if autobisimulation renders no equivalent states.

renameStatesFromPosition()

Rename states of a Glushkov automaton using the positions of the marked RE

Return type *NFA* (page 27)

reorder(dicti)

Reorder states indexes according to given dictionary.

Parameters **dicti** (*dict*) – state name reorder

Attention: in-place modification

Note: dictionary does not have to be complete

reversal()

Returns a NFA that recognizes the reversal of the language

Returns NFA recognizing reversal language

Return type *NFA* (page 27)

reverseTransitions(rev)

Evaluate reverse transition function.

Parameters **rev** (*NFA* (page 27)) – NFA in which the reverse function will be stored

setInitial(statelist)

Sets the initial states of an NFA

Parameters **statelist** (*set* / *list* / *int*) – an iterable of initial state indexes

shuffle(other)

Shuffle of a NFA

Parameters **other** (*FA* (page 21)) – an FA

Returns the resulting NFA

Return type *NFA* (page 27)

star(flag=False)

Kleene star of a NFA

Parameters **flag** (*bool*) – plus instead of star?

Returns the resulting NFA

Return type *NFA* (page 27)

stateChildren(*state*, *strict=False*)

Set of children of a state

Parameters

- **state** (*int*) – state id queried
- **strict** (*bool*) – if not strict a state is never its own child even if a self loop is in place

Returns children states

Return type *set*

stronglyConnectedComponents()

Strong components

Return type *list*

New in version 1.0.

subword()

NFA that recognizes subword(L(self))

Return type *NFA* (page 27)

succintTransitions()

Collects the transition information in a compact way suitable for graphical representation. :rtype: list

toDFA()

Construct a DFA equivalent to this NFA, by the subset construction method.

Return type *DFA* (page 5)

Note: valid to epsilon-NFA

toNFA()

Dummy identity function

Return type *NFA* (page 27)

toNFAR()

NFA with the reverse mapping of the delta function.

Returns shallow copy with reverse delta function added

Return type *NFAR* (page 36)

uniqueRepr()

Dummy representation. Used DFA.uniqueRepr()

Return type *tuple*

usefulStates(*initial_states=None*)

Set of states reachable from the given initial state(s) that have a path to a final state.

Parameters **initial_states** (*set*) – set of initial states

Returns set of state indexes

Return type *set*

witness()

Witness of non emptiness

Returns word

Return type `str`

wordImage(*word*, *ist=None*)

Evaluates the set of states reached consuming given word

Parameters

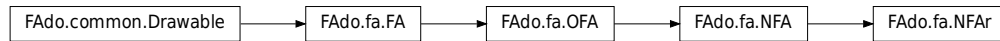
- **word** – the word
- **ist** (`int`) – starting state index (or set of)

Returns the set of ending states

Return type set of int

class NFAr

Class for Non-deterministic Finite Automata with reverse delta function added by construction. Includes efficient methods for merging states.



addTransition(*sti1*, *sym*, *sti2*)

Adds a new transition. Transition is from *sti1* to *sti2* consuming symbol *sym*. *sti2* is a unique state, not a set of them. Reversed transition function is also computed

Parameters

- **sti1** (`int`) – state index of departure
- **sti2** (`int`) – state index of arrival
- **sym** (`str`) – symbol consumed

delTransition(*sti1*, *sym*, *sti2*, *_no_check=False*)

Remove a transition if existing and perform cleanup on the transition function's internal data structure and in the reversal transition function

Parameters

- **sti1** (`int`) – state index of departure
- **sti2** (`int`) – state index of arrival
- **sym** (`str`) – symbol consumed
- **_no_check** (`bool`) – (optional) dismiss secure code

deleteStates(*del_states*)

Delete given iterable collection of states from the automaton. Performe deletion in the transition function and its reversal.

Parameters **del_states** (`set / list`) – collection of states indexes

elimEpsilon0()

Eliminate epsilon-transitions from this automaton, with reduction of states through elimination of Epsilon-cycles, and single epsilon-transition cases.

Return type *NFA* (page 36)

Attention: performs inplace modification of automaton

homogenousP(*inplace=False*)

Checks if the automaton is homogenous, i.e. the transitions that reach a state have all the same label.

Parameters **inplace** (*bool*) – if True performs Epsilon transitions elimination

Returns True if homogenous

Return type *bool*

mergeStates(*f, t*)

Merge the first given state into the second. If the first state is an initial or final state, the second becomes respectively an initial or final state.

Parameters

- **f** (*int*) – index of state to be absorbed
- **t** (*int*) – index of remaining state

Attention: It is up to the caller to remove the disconnected state. This can be achieved with `trim()`.

mergeStatesSet(*to_merge, target=None*)

Merge a set of states with a target merge state. If the states in the set have transitions among them, those transitions will be directly merged into the target state.

Parameters

- **to_merge** (*set*) – set of states to merge with target
- **target** (*int*) – optional target state

Note: if target state is not given, the minimal index will be considered.

Attention: The states of the list will become unreachable, but won't be removed. It is up to the caller to remove them. That can be achieved with `trim()`.

toNFA()

Turn into an instance of NFA, and remove the reverse mapping of the delta function.

Returns shallow copy without reverse delta function

Return type *NFA* (page 27)

unlinkSoleIncoming(*state*)

If given state has only one incoming transition (indegree is one), and it's through epsilon, then remove such transition and return the source state.

Parameters **state** (*int*) – state to check

Returns source state

Return type *int* | None

Note: if conditions aren't met, returned source state is None, and automaton remains unmodified.

unlinkSoleOutgoing(*state*)

If given state has only one outgoing transition (outdegree is one), and it's through epsilon, then remove such transition and return the target state.

Parameters *state* (*int*) – state to check

Returns target state

Return type *int* | None

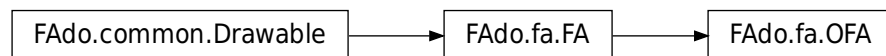
Note: if conditions aren't met, returned target state is None, and automaton remains unmodified.

class **OFA**

Base class for one-way automata

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **Initial** (*int*) – the initial state index.
- **Final** (*set*) – set of final states indexes.
- **delta** (*dict*) – the transition function.



acyclicP(*strict=True*)

Checks if the FA is acyclic

Parameters *strict* (*bool*) – if not True loops are allowed

Returns: True if the FA is acyclic bool: True if the FA is acyclic

abstract **addTransition**(*st1*, *sym*, *st2*)

Add transition

Parameters

- *st1* (*int*) – departing state
- *sym* (*str*) – label
- *st2* (*int*) – arriving state

dotDrawTransition(*st1*, *label*, *st2*, *sep='\n'*)

Draw a transition in dot format

Parameters

- **st1** (*str*) – departing state
- **label** (*str*) – symbol
- **st2** (*str*) – arriving state
- **sep** (*str*) – separator

Return type *str*

dump()

Returns a python representation of the object

Returns the python representation (Tags,States,sigma,delta,Initial,Final)

Return type *tuple*

dup()

Duplicate OFA

Returns duplicate object

Return type *OFA* (page 38)

eliminateStout(st)

Eliminate all transitions outgoing from a given state

Parameters **st** (*int*) – the state index to lose all outgoing transitions

Attention: performs in place alteration of the automata

New in version 0.9.6.

emptyP()

Tests if the automaton accepts an empty language

Return type *bool*

New in version 1.0.

abstract evalSymbol(stil, sym)

Eval symbol

minimalBrzozowski()

Constructs the equivalent minimal DFA using Brzozowski's algorithm

Returns equivalent minimal DFA

Return type *DFA* (page 5)

minimalBrzozowskiP()

Tests if the FA is minimal using Brzozowski's algorithm

Return type *bool*

abstract stateChildren(_state, _strict=None)

To be implemented below

Parameters

- **_state** (*state*) –
- **_strict** (*int*) – state id queried

Return type *list*

abstract succinctTransitions()

Collapsed transitions

topoSort()

Topological order for the FA

Returns List of state indexes

Return type `list`

Note: self loops are taken in consideration

trim()

Removes the states that do not lead to a final state, or, inclusively, that can't be reached from the initial state. Only useful states remain.

Return type `FA` (page 21)

Attention: in place transformation

trimP()

Tests if the FA is trim: initially connected and co-accessible

Return type `bool`

class SemiDFA

Class of automata without initial or final states

Variables

- **States** (`list`) – set of states.
- **sigma** (`set`) – alphabet set.
- **delta** (`dict`) – the transition function.

dotDrawState(`sti: int, sep='\n'`) → `str`

Dot representation of a state

Parameters

- **sti** (`int`) – state index.
- **sep** (`str`, optional) – separator.

Returns line to add to the dot file.

Return type `str`

static dotDrawTransition(`st1: str, lbl1: str, st2, sep='\n'`) → `str`

Draw a transition in dot format

Parameters

- **st1** (`str`) – departing state.
- **lbl1** (`str`) – label.
- **st2** (`str`) – arriving state.
- **sep** (`str`, optional) – separator.

Returns line to add to the dot file.

Return type `str`

dotFormat(*size*='20,20',*filename*=None,*direction*='LR',*strict*=False,*maxlblsz*=6,*sep*='\n') → `str`

A dot representation

Parameters

- **direction** (`str`) – direction of drawing - “LR” or “RL”
- **size** (`str`) – size of image
- **filename** (`str`) – Name of the output file
- **sep** (`str`) – line separator
- **maxlblsz** (`int`) – max size of labels before getting removed
- **strict** (`bool`) – use limitations of label sizes

Returns the dot representation

Return type `str`

New in version 0.9.6.

Changed in version 1.2.1.

emptyDFA(*sigma*=None)

Given an alphabet returns the minimal DFA for the empty language

Parameters **sigma** (`set`) – set of symbols

Return type `DFA` (page 5)

New in version 1.3.4.2.

saveToString(*aut*, *sep*='&')

Finite automata definition as a string using the input format.

New in version 0.9.5.

Changed in version 0.9.6: Names are now used instead of indexes.

Changed in version 0.9.7: New format with quotes and alphabet

Parameters

- **aut** (`FA` (page 21)) – the FA
- **sep** (`str`) – separation between *lines*

Returns the representation

Return type `str`

sigmaStarDFA(*sigma*=None)

Given a alphabet *s* returns the minimal DFA for *s**

Parameters **sigma** (`set`) – set of symbols

Return type `DFA` (page 5)

New in version 1.2.

statePP(*state*)

Pretty print state

Parameters **state** –

Returns

stringToDFA(*s*, *f*, *n*, *k*)

Converts a string icdfa's representation to dfa.

Parameters

- **s** (*list*) – canonical string representation
- **f** (*list*) – bit map of final states
- **n** (*int*) – number of states
- **k** (*int*) – number of symbols

Returns a complete dfa with sigma [**k**], States [**n**]

Return type *DFA* (page 5)

Changed in version 0.9.8: symbols are converted to str

FADO.CONVERSIONS

Conversions between objects.

Deterministic and non-deterministic automata manipulation, conversion and evaluation. .. *Authors:* Rogério Reis & Nelma Moreira .. *This is part of FAdo project* <https://fado.dcc.fc.up.pt>.

DFA2regexpDijkstra(*aut*) → *FAdo.reex.RegExp* (page 84)

Returns a regexp for the current DFA considering the recursive method. Very inefficient.

Parameters *aut* (*DFA* (page 5)) – the automaton

Returns a regexp equivalent to the current DFA

Return type *reex.RegExp* (page 84)

DFA2syncWords(*aut*)

Evaluates the regular expression corresponding to the synchronizing pwords of the automata.

Parameters *aut* (*DFA* (page 5)) – the automata

Returns a regular expression of the sync words of the automata

Return type *reex.RegExp* (page 84)

FA2GFA(*aut*)

Creates a GFA equivalent to NFA

Parameters *aut* (*OFA* (page 38)) – the automaton

Returns a GFA deep copy

Return type *GFA* (page 45)

FA2regexpCG(*aut*)

Regular expression from state elimination whose language is recognised by the FA. Uses a heuristic to choose the order of elimination.

Parameters *aut* (*OFA* (page 38)) – the automaton

Returns the equivalent regular expression

Return type *reex.RegExp* (page 84)

FA2regexpCG_nn(*aut*: *FAdo.fa.OFA* (page 38))

Regular expression from state elimination whose language is recognised by the FA. Uses a heuristic to choose the order of elimination. The FA is not normalized before the state elimination.

Parameters *aut* (*OFA* (page 38)) – the automaton

Returns the equivalent regular expression

Return type *reex.RegExp* (page 84)

FA2regexpDynamicCycleHeuristic(*aut*)

State elimination Heuristic based on the number of cycles that passes through each state. Here those numbers are evaluated dynamically after each elimination step

Parameters *aut* ([OFA](#) (page 38)) – the automaton

Returns an equivalent regular expression

Return type [reex.RegExp](#) (page 84)

See also:

Nelma Moreira, Davide Nabais, and Rogério Reis. State elimination ordering strategies: Some experimental results. Proc. of 11th Workshop on Descriptive Complexity of Formal Systems (DCFS10), pages 169-180.2010. DOI: 10.4204/EPTCS.31.16

FA2regexpSE(*aut*)

A regular expression obtained by state elimination algorithm whose language is recognised by the FA *aut*.

Parameters *aut* ([OFA](#) (page 38)) – the automaton

Returns the equivalent regular expression

Return type [reex.RegExp](#) (page 84)

FA2regexpSEO(*aut*, *order=None*)

Regular expression from state elimination whose language is recognised by the FA. The FA is normalized before the state elimination.

Parameters

- *aut* ([OFA](#) (page 38)) – the automaton
- *order* ([list](#)) – state elimination sequence

Returns the equivalent regular expression

Return type [reex.RegExp](#) (page 84)

FA2regexpSE_nn(*aut*, *order=None*)

Regular expression from state elimination whose language is recognised by the FA. The FA is not normalized before the state elimination.

Parameters

- *aut* ([OFA](#) (page 38)) – the automaton
- *order* ([list](#)) – state elimination sequence

Returns the equivalent regular expression

Return type [reex.RegExp](#) (page 84)

FA2regexpStaticCycleHeuristic(*aut*)

State elimination Heuristic based on the number of cycles that passes through each state. Here those numbers are evaluated statically in the beginning of the process

Parameters *aut* ([OFA](#) (page 38)) – the automaton

Returns a equivalent regular expression

Return type [reex.RegExp](#) (page 84)

See also:

Nelma Moreira, Davide Nabais, and Rogério Reis. State elimination ordering strategies: Some experimental results. Proc. of 11th Workshop on Descriptive Complexity of Formal Systems (DCFS10), pages 169-180.2010. DOI: 10.4204/EPTCS.31.16

FAallRegExps(*aut*)

Evaluates the alphabetic length of the equivalent regular expression using every possible order of state elimination.

Parameters *aut* (OFA (page 38)) – the automaton

Return type list of tuples (*int*, list of states)

FAeliminateSingles(*aut*)

Eliminates every state that only have one successor and one predecessor.

Parameters *aut* (OFA (page 38)) – the automaton

Returns GFA after eliminating states

Return type GFA (page 45)

class GFA

Class for Generalized Finite Automata: NFA with a unique initial state and transitions are labeled with RegExp.

**DFS**(*io*)

Depth first search

Parameters *io* –

addTransition(*sti1*, *sym*, *sti2*)

Adds a new transition from *sti1* to *sti2* consuming symbol *sym*. Label of the transition function is a RegExp.

Parameters

- *sti1* (*int*) – state index of departure
- *sti2* (*int*) – state index of arrival
- *sym* (*str*) – symbol consumed

Raises **DFAepsilonRedefinition** – if *sym* is Epsilon

assignLow(*st*)

Parameters *st* –

assignNum(*st*)

Parameters *st* –

completeDelta()

Adds empty set transitions between the automaton's final and initial states in order to make it complete. It's only meant to be used in the final stage of SEA...

deleteState(*sti*)

Deletes a state from the GFA :param sti:

dfs_visit(*s, visited, io*)

Parameters

- **s** – state
- **visited** – list of states visited
- **io** –

dup()

Returns a copy of a GFA

Return type *GFA* (page 45)

eliminate(*st*)

Eliminate a state.

Parameters **st** (*int*) – state to be eliminated

eliminateAll(*lr*)

Eliminate a list of states.

Parameters **lr** (*list*) – list of states indexes

eliminateState(*st*)

Deletes a state and updates the automaton

Parameters **st** (*int*) – the state to be deleted

evalNumberOfStateCycles()

Evaluates the number of cycles each state participates

Returns state->list of cycle lengths

Return type *dict*

evalSymbol(*stil, sym*)

Eval symbol

normalize()

Create a single initial and final state with Epsilon transitions.

Attention: works in place

reorder(*dictio*)

Reorder states indexes according to given dictionary.

Parameters **dictio** (*dict*) – order

Note: dictionary does not have to be complete

stateChildren(*state, strict=False*)

Set of children of a state

Parameters

- **strict** (*bool*) – a state is never its own children even if a self loop is in place
- **state** (*int*) – state id queried

Returns map: children -> alphabetic length

Return type dictionary

succintTransitions()

Collapsed transitions

weight(*state*)

Calculates the weight of a state based on a heuristic

Parameters **state** (*int*) – state

Returns the weight of the state

Return type *int*

weightWithCycles(*state*, *cycles*)**Parameters**

- **state** –
- **cycles** –

Returns

SP2regexp(*aut*)

Checks if FA is SP (Serial-PArallel), and if so returns the regular expression whose language is recognised by the FA

Parameters **aut** (*OFA* (page 38)) – the automaton

Returns equivalent regular expression

Return type *reex.RegExp* (page 84)

Raises *NotSP* (page 51) – if the automaton is not Serial-Parallel

See also:

Moreira & Reis, Fundamenta Informatica, Series-Parallel automata and short regular expressions, n.91 3-4, pag 611-629. <https://www.dcc.fc.up.pt/~nam/publica/spa07.pdf>

Note: Automata must be Serial-Parallel

cutPoints(*aut*)

Set of FA's cut points

Parameters **aut** (*OFA* (page 38)) – the automaton

Returns set of states

Return type set of *int*

FADO.COMMON

Common definitions for FAdo files

class AllWords(*alphabet*)
Iterator that generates all words of an alphabet in militar order

exception CFGerror

exception CFGgrammarError(*rule*)

exception CFGterminalError(*size*)

exception CodesError(*msg*)

exception CodingTheoryError(*msg*)

exception DFAEmptyDFA

exception DFAEmptySigma

exception DFAFileError

exception DFAFound(*word*)

exception DFASyntaticError(*line*)

exception DFAdifferentSigma

exception DFAepsilonRedefinition

exception DFAequivalent

exception DFAerror

exception DFAinputError(*word*)

exception DFAmarkedError(*sym*)

exception DFAnoInitial

exception DFAnotComplete

exception DFAnotMinimal

exception DFAnotNFA(*msg*)

exception DFAstateUnknown(*stidx*)

exception DFAstopped

exception DFAsymbolUnknown(*sym*)

class Drawable

Any FAdo object that is drawable

display(*filename=None, size='30,20', strict=False, maxlblsz=6*)

Display automata using dot

Parameters

- **size** – size of representation
- **fileName** – filename to use for the graphic representation (default a os tmpfile)
- **maxlblsz** (*int*) – max size of labels before getting removed
- **strict** (*bool*) – use limitations of label sizes

Changed in version 1.2.1.

abstract dotFormat(*size='20,20', filename=None, direction='LR', strict=False, maxlblsz=6, sep='\n'*)

Some dot representation

Parameters

- **size** (*str*) – size parameter for dotviz
- **filename** (*str*) – filename
- **direction** (*str*) –
- **strict** (*bool*) –
- **maxlblsz** (*int*) –
- **sep** (*str*) –

Returns: str:

dotLabel(*lbl0*)

Label string

makePNG(*filename=None, size='30,20'*)

Produce png file to display

Parameters

- **filename** (*str*) – file name, if None will be a tmpfile
- **size** – size for graphviz

Returns name of the file created

New in version 1.0.4.

exception DuplicateName(*number*)

exception FAException

exception FAdoError

exception FAdoGeneralError(*msg*)

exception FAdoNotImplemented

exception FAdoSyntacticError

exception GraphError(*message*)

exception IllegalBias(*msg*)

exception NFADoEmpty

exception `NFAerror`

exception `NImplemented`

exception `NonPlanar`

exception `NotSP`

exception `PDAerror`

exception `PDAsymbolUnknown`(*symb*)

exception `PEGError`(*msg*)

exception `PropertyNotSatisfied`(*msg*)

class `SPLabel`(*val=None*)

Label class for Serial-Parallel test algorithm

See also:

Moreira & Reis, Fundamenta Informatica, ‘Series-Parallel automata and short regular expressions’, n.91 3-4, pag 611-629

exception `SSBadTransition`

exception `SSError`

exception `SSMissAlphabet`

exception `TFAAccept`

exception `TFAResject`

exception `TFAResjectBlocked`

exception `TFAResjectLoop`

exception `TFAResjectNonFinal`

exception `TFASignal`

exception `TRError`

exception `TstError`(*message*)

exception `TypeError`

exception `VersoError`(*msg*)

exception `VertexNotInGraph`

class `Word`(*data=None*)

Class to implement generic words as iterables with pretty-print

Basically a unified way to deal with words with characters of sizes different of one with no much fuss

binomial(*n, k*)

Exactly what it seems

Parameters

- **n** (*int*) – n
- **k** (*int*) – k

Return type *int*

delFromList(*l, ll*)

Delete every element of ll from l

Parameters

- **l** (*list*) –
- **ll** (*list*) –

dememoize(*cls, method_name*)

Restore method of given class from memoized state. Stored attributes will be removed.

exception fnhException

forceIterable(*x*)

Forces a non iterable object into a list, otherwise returns itself

Parameters **x** (*list*) – the object

Returns object as a list

Return type *list*

graphvizTranslate(*s, strict=False, maxlblsz=6*)

Translate epsilons for graphviz

Parameters

- **s** (*str*) – symbol
- **maxlblsz** – max size of labels before getting removed
- **strict** (*bool*) – use limitations of label sizes

Return type *str*

homogeneousP(*l*)

Is the list homogeneous?

Parameters **l** (*list*) – list to be inspected

Return type *bool*

lSet(*s*)

returns the last element of a set

Parameters **s** (*set*) – the set

Returns the last element of the set

New in version 1.3.3.

memoize(*cls, method_name*)

Memoizes a given method result on instances of given class.

Given method should have no side effects. Results are stored as instance attributes — given parameters are disregarded.

Parameters

- **cls** –
- **method_name** –

class memoized(*func*)

Decorator that caches a function's return value each time it is called.

If called later with the same arguments, the cached value is returned, and not re-evaluated.

exception notAcyclic

overlapFreeP(*word*)

Returns True if word is overlap free, i.e, no proper and nonempty prefix is a suffix

Parameters **word** – the word

Return type Boolean

exception `regexInvalid(word)`

exception `regexInvalidMethod`

exception `regexInvalidSymbols`

sConcat(*x*, *y*)

CConcat words

Parameters

- **x** – first word
- **y** – second word

Returns concatenation word

suffixes(*word*)

Returns the list of proper suffixes of a word

Parameters **word** (*str*) – the word

Return type *list*

class `twDict(fw=None)`

A class for dictionaries ‘both ways’

uSet(*s*)

returns the first element of a set

Parameters **s** (*set*) – the set

Returns the first element of s

unique(*l*)

Eliminate duplicates

Parameters **l** (*list*) – source list

Returns list without repetitions

Return type *lst*

FADO.FIO

In/Out.

FAdo I/O methods. The parsing grammars for most of the objects reside here.

class BuildFadoObject(*visit_tokens=True*)
Semantics of the FAdo grammars' objects

readFromFile(*FileName*)
Reads list of finite automata definition from a file.

Parameters **FileName** (*str*) – file name

Return type *list*

The format of these files must be the as simple as possible:

- # begins a comment
- @DFA or @NFA begin a new automata (and determines its type) and must be followed by the list of the final states separated by blanks
- fields are separated by a blank and transitions by a CR: `state symbol new state`
- in case of a NFA declaration, the “symbol” @epsilon is interpreted as an epsilon-transition
- the source state of the first transition is the initial state
- in the case of a NFA, its declaration @NFA can, after the declaration of the final states, have a * followed by the list of initial states
- both, NFA and DFA, may have a declaration of alphabet starting with a \$ followed by the symbols of the alphabet
- a line with a sigle name, decreares a state

```

FAdo      ::=  FA | FA CR FAdo
FA         ::=  DFA | NFA | Transducer
DFA        ::=  "@DFA" LsStates Alphabet CR dTrans
NFA        ::=  "@NFA" LsStates Initials Alphabet CR nTrans
Transducer ::=  "@Transducer" LsStates Initials Alphabet Output CR tTrans
Initials   ::=  "*" LsStates | /Epsilon
Alphabet   ::=  "$" LsSymbols | /Epsilon
Output     ::=  "$" LsSymbols | /Epsilon
nSymbol    ::=  symbol | "@epsilon"
LsStates   ::=  stateid | stateid , LsStates
LsSymbols  ::=  symbol | symbol , LsSymbols
dTrans     ::=  stateid symbol stateid |
                | stateid symbol stateid CR dTrans
nTrans     ::=  stateid nSymbol stateid |
                | stateid nSymbol stateid CR nTrans
tTrans     ::=  stateid nSymbol nSymbol stateid |

```

| stateid nSymbol nSymbol stateid CR nTrans

Note: If an error occur, either syntactic or because of a violation of the declared automata type, an exception is raised

Changed in version 0.9.6.

Changed in version 1.0.

readOneFromFile(*fileName*)

Read the first of the FAdo objects from File

Parameters **fileName** (*str*) – name of the file

Return type *DFA* (page 5)|*FA* (page 21)|*STF*|*SST* (page 149)

readOneFromString(*s*)

Reads one finite automata definition from a file.

See also:

readFromFile for description of format

Parameters **s** (*str*) – the string

Return type *DFA* (page 5)|*NFA* (page 27)|*SFT* (page 106)

saveToFile(*FileName*, *fa*, *mode*='a')

Saves a list finite automata definition to a file using the input format

Changed in version 0.9.5.

Changed in version 0.9.6.

Changed in version 0.9.7: New format with quotes and alphabet

Parameters

- **FileName** (*str*) – file name
- **fa** (*list of FA*) – the FA
- **mode** (*str*) – writing mode

saveToJson(*FileName*, *aut*, *mode*='w')

Saves a finite automata definition to a file using the JSON format

saveToString(*fa*)

Saves a finite automaton definition to a string :param fa: automaton :return: the string containing the automaton definition :rtype: str

..versionadded:: 1.2.1

show(*obj*)

General, context sensitive, display method :param obj: the object to show

New in version 1.2.1.

toJson(*aut*)

Json for a FA

Parameters **aut** (*FA* (page 21)) – the automaton

Return type *str*

FADO.REEX

Regular expressions manipulation

Regular expression classes and manipulation

class **BuildRPNRegexp**(*context=None*)

class **BuildRPNSRE**(*context=None*)

class **BuildRegexp**(*context=None*)

Semantics of the FAdo grammars' regexps Priorities of operators: disj > conj > shuffle > concat > not > star >= option

class **BuildSRE**(*context=None*)

Parser for sre

class **CAtom**(*val, sigma=None*)

Simple Atom (symbol)

Variables

- **Sigma** – alphabet set of strings
- **val** – the actual symbol

FAdo.reex.RegularExpression

FAdo.reex.RegExp

Constructor of a regular expression symbol.

Parameters **val** – the actual symbol

PD()

Closure of partial derivatives of the regular expression in relation to all words.

Returns set of regular expressions

Return type *set*

See also:

Antimirov, 95

static alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression.

Return type `int`

Attention: Doesn't include the empty word.

derivative(*sigma*)

Derivative of the regular expression in relation to the given symbol.

Parameters **sigma** (*str*) – an arbitrary symbol.

Returns regular expression

Return type *RegExp* (page 84)

Note: whether the symbols belong to the expression's alphabet goes unchecked. The given symbol will be matched against the string representation of the regular expression's symbol.

See also:

J. A. Brzozowski, Derivatives of Regular Expressions. J. ACM 11(4): 481-494 (1964)

static epsilonLength()

Number of occurrences of the empty word in the regular expression.

Return type `int`

first(*parent_first=None*)

List of possible symbols matching the first symbol of a string in the language of the regular expression.

Parameters **parent_first** (*list*) –

Returns list of symbols

Return type *list*

first_l()

First set for locations

followLists(*lists=None*)

Map of each symbol's follow list in the regular expression.

Parameters **lists** (*dict*) –

Returns map of symbols' follow lists {symbol: list of symbols}

Return type *dict*

Attention: For first() and last() return lists, the follow list for certain symbols might have repetitions in the case of follow maps calculated from Star operators. The union of last(), first() and follow() sets are always disjoint when the regular expression is in Star normal form (Bruggemann-Klein, 92), therefore FAdo implements them as lists. You should order exclusively, or take a set from a list in order to resolve repetitions.

followListsD(*lists=None*)

Map of each symbol's follow list in the regular expression.

Parameters `lists` (*dict*) –

Returns map of symbols' follow list {symbol: list of symbols}

Return type *dict*

Attention: For `first()` and `last()` return lists, the follow list for certain symbols might have repetitions in the case of follow maps calculated from star operators. The union of `last()`, `first()` and `follow()` sets are always disjoint

See also:

Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. On the average size of glushkov and partial derivative automata. International Journal of Foundations of Computer Science, 23(5):969-984, 2012.

followListsStar(*lists=None*)

Map of each symbol's follow list in the regular expression under a star.

Parameters `lists` (*dict*) –

Returns map of symbols' follow lists {symbol: list of symbols}

Return type

dict

See also:

Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. On the average size of glushkov and partial derivative automata. International Journal of Foundations of Computer Science, 23(5):969-984, 2012.

follow_l()

Follow set for locations

last(*parent_last=None*)

List of possible symbols matching the last symbol of a string in the language of the regular expression.

Parameters `parent_last` (*list*) –

Returns list of symbols

Return type *list*

last_l()

Last set for locations

linearForm()

Linear form of the regular expression , as a mapping from heads to sets of tails, so that each pair (head, tail) is a monomial in the set of linear forms.

Returns: *dict*: dictionary mapping heads to sets of tails

See also:

Antimirov, 95

linearFormC()

Returns linear form

Return type *dict*

linearP()

Whether the regular expression is linear; i.e., the occurrence of a symbol in the expression is unique.

Return type `bool`

mark()

Return type `MAtom` (page 82)

static measure(*from_parent=None*)

A list with four measures for regular expressions.

Parameters **from_parent** –

Returns the measures

Return type `[int,int,int,int]`

[alphabeticLength, treeLength, epsilonLength, starHeight]

1. alphabeticLength: number of occurrences of symbols of the alphabet;
2. treeLength: number of functors in the regular expression, including constants.
3. epsilonLength: number of occurrences of the empty word.
4. starHeight: highest level of nested Kleene stars, starting at one for one star occurrence.
5. disjLength: number of occurrences of the disj operator
6. concatLength: number of occurrences of the concat operator
7. starLength: number of occurrences of the star operator
8. conjLength: number of occurrences of the conj operator
9. starLength: number of occurrences of the shuffle operator

Attention: Methods for each of the measures are implemented independently. This is the most effective for obtaining more than one measure.

nfaThompson()

Epsilon-NFA constructed with Thompson's method that accepts the regular expression's language.

Returns NFA Thompson

Return type `NFA` (page 27)

See also:

K. Thompson. Regular Expression Search Algorithm. CACM 11(6), 419-422 (1968)

partialDerivatives(*sigma*)

Set of partial derivatives of the regular expression in relation to given symbol.

Parameters **sigma** (*str*) – symbol in relation to which the derivative will be calculated.

Returns set of regular expressions

Return type `set`

See also:

Antimirov, 95

partialDerivativesC(*sigma*)

Parameters **sigma** (*str*) – symbol

Returns set of partial derivatives

Return type *set*

reduced(*has_epsilon=False*)

Equivalent regular expression with the following cases simplified:

1. $\text{Epsilon.RE} = \text{RE.Epsilon} = \text{RE}$
2. $\text{EmptySet.RE} = \text{RE.EmptySet} = \text{EmptySet}$
3. $\text{EmptySet} + \text{RE} = \text{RE} + \text{EmptySet} = \text{RE}$
4. $\text{Epsilon} + \text{RE} = \text{RE} + \text{Epsilon} = \text{RE}$, where Epsilon is in $L(\text{RE})$
5. $\text{RE}^{**} = \text{RE}^*$
6. $\text{EmptySet}^* = \text{Epsilon}^* = \text{Epsilon}$
7. $\text{Epsilon:RE} = \text{RE:Epsilon} = \text{RE}$

Parameters

- **has_epsilon** (*bool*) – used internally to indicate that the language of which this term is a subterm has the empty
- **word.** –

Returns regular expression

Return type *RegExp* (page 84)

Attention: Returned structure isn't strictly a duplicate. Use `__copy__()` for that purpose.

reversal()

Reversal of *RegExp*

Return type *Regexp*

rpn()

RPN representation

Returns printable RPN representation

Return type *str*

setOfSymbols()

Set of symbols that occur in a regular expression..

Returns set of symbols

Return type *set*

snf(*hollowdot=False*)

Star Normal Form (SNF) of the regular expression.

Parameters **hollowdot** (*bool*) – if True computes hollow dot function else black dot

Returns regular expression in star normal form

Return type *RegExp* (page 84)

static starHeight()

Maximum level of nested regular expressions with a star operation applied. For instance, `starHeight(((a*b)*+b*)*)` is 3.

Return type `int`

stringLength()

Length of the string representation of the regular expression.

Returns string length

Return type `int`

support(*side=True*)

Support of a regular expression.

Parameters **side** (*bool*) – if True concatenation of a set on the left if False on the right (prefix support)

Returns set of regular expressions

Return type `set`

See also:

Champarnaud, J.M., Ziadi, D.: From Mirkin's prebases to Antimirov's word partial derivative. Fundam. Inform. 45(3), 195-205 (2001)

See also:

Maia et al, Prefix and Right-partial derivative automata, 11th CIE 2015, 258-267 LNCS 9136, 2015

supportlast(*side=True*)

Subset of support such that elements have ewp

Parameters **side** (*bool*) – if True left-derivatives else right-derivatives

Returns set of partial derivatives

Return type `set`

static syntacticLength()

Number of nodes of the regular expression's syntactical tree (sets).

Return type `int`

tailForm()

Returns tail form

Return type `dict`

static treeLength()

Number of nodes of the regular expression's syntactical tree.

Return type `int`

unmarked()

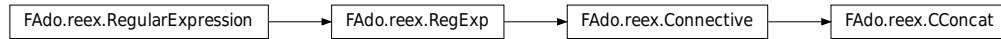
The unmarked form of the regular expression. Each leaf in its syntactical tree becomes a `RegExp()`, the `CEpsilon()` or the `CEmptySet()`.

Returns (general) regular expression

Return type *RegExp* (page 84)

class CConcat(*arg1, arg2, sigma=None*)

Class for catenation operation on regular expressions.



ewp()

Whether the empty word property holds for this regular expression's language.

Return type *bool*

first(*parent_first=None*)

First set

Returns first position set

Return type *set*

first_l()

First sets for locations

followLists(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type *dict*

followListsD(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type *dict*

follow_l()

Follow sets for locations

last(*parent_last=None*)

Last set

Returns last position set

Return type *set*

last_l()

Last sets for locations

linearForm()

Returns linear form

Return type *dict*

mark()

Make all atoms maked (tag False)

Return type *RegExp* (page 84)

reversal()

Reversal of RegExp

Return type *reex.RegExp* (page 84)

rpn()

Return type *str*

snf(*_hollowdot=False*)

Star Normal Form

support(*side=True*)

Set of partial derivatives

tailForm()

Returns tail form

Return type *dict*

unmark()

Conversion back to unmarked atoms :rtype: CConcat

class CConj(*arg1, arg2, sigma=None*)

Intersection operation of regexps

ewp()

Whether the empty word property holds for this regular expression's language.

Return type *bool*

first_l()

First sets for locations

follow_l()

Follow sets for locations

last_l()

Last sets for locations

linearForm()

Returns linear form

Return type *dict*

mark()

Make all atoms maked (tag False)

Return type *RegExp* (page 84)

rpn()

RPN representation

Returns printable RPN representation

Return type *str*

snf()

Star Normal Form

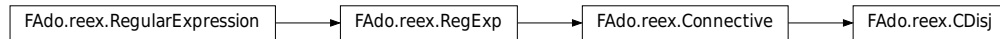
tailForm()

Returns tail form

Return type `dict`

class CDisj(*arg1, arg2, sigma=None*)

Class for disjunction/union operation on regular expressions.



ewp()

Whether the empty word property holds for this regular expression's language.

Return type `bool`

first(*parent_first=None*)

First set

Returns first position set

Return type `set`

first_l()

First sets for locations

followLists(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsD(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

follow_l()

Follow sets for locations

last(*parent_last=None*)

Last set

Returns last position set

Return type `set`

last_l()

Last sets for locations

linearForm()

Returns linear form

Return type `dict`

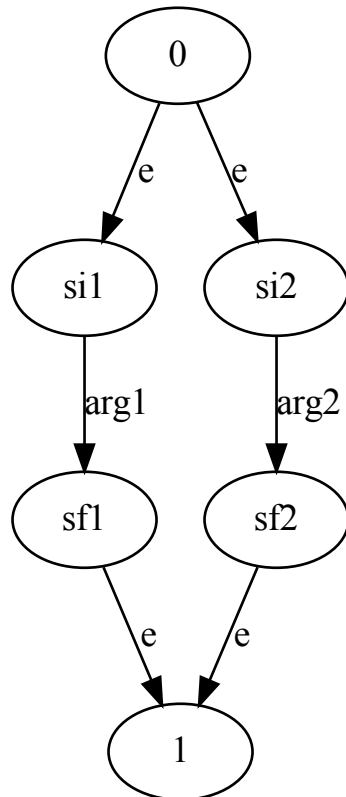
mark()

Conversion to marked atoms :rtype: CDisj

nfaThompson()

Returns an NFA (Thompson) that accepts the RE.

Return type *NFA* (page 27)



reversal()

Reversal of RegExp

Return type *reex.RegExp* (page 84)

rpn()

RPN representation

Returns printable RPN representation

Return type `str`

snf(*hollowdot=False*)

Star Normal Form

support(*side=True*)
Set of partial derivatives

tailForm()

Returns tail form

Return type dict

unmark()

Conversion back to unmarked atoms :rtype: CDisj

class CEmptySet(*sigma=None*)
Class that represents the empty set.



Constructor of a regular expression symbol.

Parameters **val** – the actual symbol

static emptysetP()

Returns

static epsilonLength()

Returns

static epsilonP()

Returns

ewp()

Returns

static measure(*from_parent=None*)

Parameters **from_parent** –

Returns

nfaPD(*pdmethode='nfaPDNaive'*)

Computes the partial derivative automaton

partialDerivatives(*_*)

Partial derivatives

partialDerivativesC(*_*)

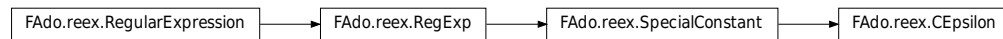
Returns

rpn()

Returns

snf(*_hollowdot=False*)
Star Normal Form

class CEpsilon(*sigma=None*)
Class that represents the empty word.



Constructor of a regular expression symbol.

Parameters **val** – the actual symbol

static epsilonLength()
Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type `int`

static epsilonP()

Return type `bool`

ewp()

Return type `bool`

static measure(*from_parent=None*)

Parameters **from_parent** –

Returns measures

nfaThompson()

Return type *NFA* (page 27)

partialDerivatives(*_*)

Returns

partialDerivativesC(*_*)

Returns

rpn()

Returns `str`

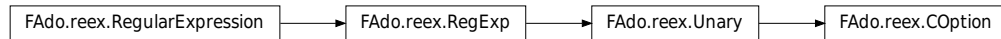
snf(*_hollowdot=False*)

Parameters *_hollowdot* –

Returns

class COption(*arg, sigma=None*)

Class for option operation (reg + @epsilon) on regular expressions.



epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type `int`

ewp()

Whether the empty word property holds for this regular expression's language.

Return type `bool`

first(*parent_first=None*)

First set

Returns first position set

Return type `set`

first_l()

First sets for locations

followLists(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsD(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsStar(*lists=None*)

to be fixed

follow_l()

Follow sets for locations

last(*parent_first=None*)

Last set

Returns last position set

Return type `set`

last_l()

Last sets for locations

linearForm()

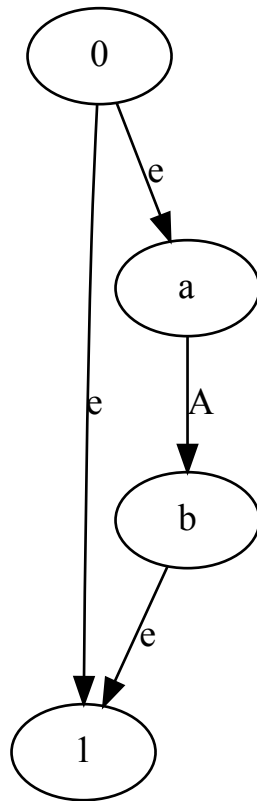
Returns linear form

Return type `dict`

nfaThompson()

Returns a NFA that accepts the RE.

Return type *NFA* (page 27)



rpn()

RPN representation

Returns printable RPN representation

Return type `str`

setOfSymbols()

Returns set of symbols

Return type `set`

snf(*_hollowdot=False*)
Star Normal Form

starHeight()
Maximum level of nested regular expressions with a star operation applied.
For instance, `starHeight(((a*b)*+b*)*)` is 3.

Returns number of nested star

Return type `int`

support(*side=True*)
Set of partial derivatives

tailForm()

Returns tail form

Return type `dict`

class CShuffle(*arg1, arg2, sigma=None*)
Shuffle operation of regexps

ewp()
Whether the empty word property holds for this regular expression's language.

Return type `bool`

first(*parent_list=None*)

Parameters *parent_list* –

Returns

first_l()
First sets for locations

followListsD(*lists=None*)
in progress

follow_l()
Follow sets for locations

last_l()
Last sets for locations

linearForm()

Returns linear form

Return type `dict`

mark()
Make all atoms maked (tag False)

Return type `RegExp` (page 84)

rpn()
RPN representation

Returns printable RPN representation

Return type `str`

snf()

Star Normal Form

tailForm()

Returns tail form

Return type `dict`

class CShuffleU(*arg, sigma=None*)

Unary Shuffle operation of regexprs

epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type `int`

ewp()

Whether the empty word property holds for this regular expression's language.

Return type `bool`

first(*parent_list=None*)

Parameters *parent_list* –

Returns

followListsD(*lists=None*)

in progress

last(*parent_last=None*)

Last set

Returns last position set

Return type `set`

linearForm()

Returns linear form

Return type `dict`

mark()

Make all atoms maked (tag False)

Return type [*RegExp*](#) (page 84)

rpn()

RPN representation

Returns printable RPN representation

Return type `str`

snf()

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, `starHeight(((a*b)*+b*)*)` is 3.

Returns number of nested star

Return type `int`

tailForm()

Returns tail form

Return type `dict`

unmark()

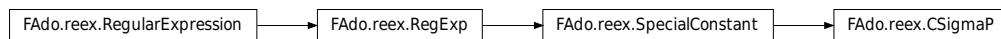
Conversion back to `RegExp`

Return type `reex.__class__`

class `CSigmaP(sigma=None)`

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;
 associativity of concatenation; identities σ^* and σ^+ .

`CSigmaP`: Class that represents the complement of the `EmptySet` word (σ^+)



Constructor of a regular expression symbol.

Parameters **val** – the actual symbol

derivative(sigma)

Parameters **sigma** –

Returns

ewp()

Returns

linearForm()

Returns

linearFormC()

Returns

nfaPD(pDMETHOD='nfaPDNaive')

Computes the partial derivative automaton

partialDerivatives(*sigma*)

Parameters *sigma* –

Returns

partialDerivativesC(*_*)

Parameters *_* –

Returns

rpn()

RPN representation

Returns printable RPN representation

Return type `str`

support(*side=True*)

Returns

class CSigmaS(*sigma=None*)

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;
 associativity of concatenation; identities σ^* and σ^+ .

CSigmaS: Class that represents the complement of the EmptySet set (σ^*)



Constructor of a regular expression symbol.

Parameters *val* – the actual symbol

derivative(*sigma*)

Parameters *sigma* –

Returns

ewp()

Returns

linearForm()

Returns

linearFormC()

Returns

nfaPD(*pmethod='nfaPDNaive'*)
Computes the partial derivative automaton

partialDerivatives(*sigma*)

Parameters *sigma* –

Returns

partialDerivativesC(*sigma*)

Parameters *sigma* –

Returns

rpn()

RPN representation

Returns printable RPN representation

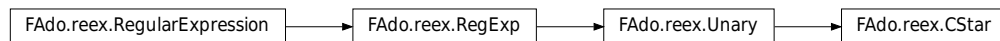
Return type `str`

support(*side=True*)

Returns

class CStar(*arg, sigma=None*)

Class for iteration operation (aka Kleene star, or Kleene closure) on regular expressions.



epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type `int`

ewp()

Whether the empty word property holds for this regular expression's language.

Return type `bool`

first(*parent_first=None*)

First set

Returns first position set

Return type `set`

first_l()

First sets for locations

followLists(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsD(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

follow_l()

Follow sets for locations

last(*parent_first=None*)

Last set

Returns last position set

Return type `set`

last_l()

Last sets for locations

linearForm()

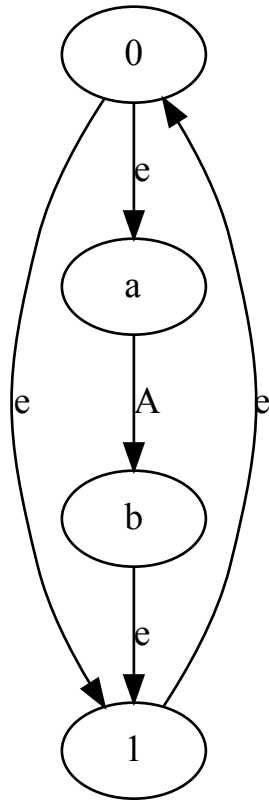
Returns linear form

Return type `dict`

nfaThompson()

Returns a NFA that accepts the RE.

Return type `NFA` (page 27)

**rpn()**

RPN representation

Returns printable RPN representation**Return type** `str`**snf**(*_hollowdot=False*)

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

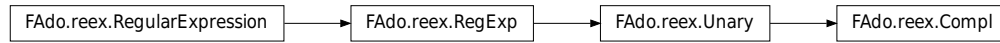
For instance, `starHeight(((a*b)*+b*)*)` is 3.**Returns** number of nested star**Return type** `int`**support**(*side=True*)

Set of partial derivatives

tailForm()**Returns** tail form**Return type** `dict`

class Compl(*arg*, *sigma=None*)

Class for not operation on regular expressions.



epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type `int`

ewp()

Whether the empty word property holds for this regular expression's language.

Return type `bool`

first()

First set

Returns first position set

Return type `set`

followLists()

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsD()

Follow set

Returns for each key position a set of follow positions

Return type `dict`

last()

Last set

Returns last position set

Return type `set`

linearForm()

Returns linear form

Return type `dict`

mark()

Make all atoms maked (tag False)

Return type *RegExp* (page 84)

rpn()

RPN representation

Returns printable RPN representation

Return type `str`

snf()

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, `starHeight(((a*b)*+b*)*)` is 3.

Returns number of nested star

Return type `int`

support(*side=True*)

Set of partial derivatives

tailForm()

Returns tail form

Return type `dict`

treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns tree lenght

Return type `int`

unmark()

Conversion back to `RegExp`

Return type `reex.__class__`

class Connective(*arg1, arg2, sigma=None*)

Base class for (binary) operations: concatenation, disjunction, etc



alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression. :returns: alphapetic length :rtype: int

Attention: Doesn't include the empty word.

epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type `int`

first(*parent_first=None*)

First set

Returns first position set

Return type `set`

followLists(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsD(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

last(*parent_last=None*)

Last set

Returns last position set

Return type `set`

abstract linearForm()

Returns linear form

Return type `dict`

abstract mark()

Make all atoms maked (tag False)

Return type `RegExp` (page 84)

abstract rpn()

RPN representation

Returns printable RPN representation

Return type `str`

setOfSymbols()

Returns set of symbols

Return type `set`

abstract snf()

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, `starHeight(((a*b)*+b*)*)` is 3.

Returns number of nested star

Return type `int`

treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns tree lenght

Return type `int`

class `DAG(reg)`

Class to support dags representing regexps

...seealso: P. Flajolet, P. Sipala, J.-M. Steyaert, **Analytic variations on the common subexpression problem**, in: Automata, Languages and Programmin, LNCS, vol. 443, Springer, New York, 1990, pp. 220–234.

Args: reg (RegExp): regular expression

NFA()

Returns the partial derivative automaton

Return type `NFA` (page 27)

catLF(idl, idr, delay=False)

Linear form for concatenation :param idl: node :type idl: int :param idr: node :type idr: int :param delay: if true partial derivatives are delayed :type delay: bool

Returns partial derivatives

Return type `dict`

..note:: both arguments are assumed to be already present in the DAG

getAtomIdx(val)

Node atom :param val: letter :type val: str

Returns node id

Return type `int`

getIdx(reg)

Builds dag nodes :param reg: regular expression :type reg: regexp

Returns node id

Return type `int`

interLF(diff1, diff2)

Intersection of partial derivatives

Parameters

- **diff1** (`dict`) – partial diff of the first argument
- **diff2** (`dict`) – partial diff of the second argument

Return type `dict`

static plusLF(diff1, diff2)

Union of partial derivatives

Parameters

- **diff1** (`dict`) – partial diff of the first argument
- **diff2** (`dict`) – partial diff of the second argument

Return type `dict`

shuffleLF(id1, id2)

Shuffle of partial derivatives :param id1: node :type id1: int :param id2: node :type id2: int

Returns partial derivatives

Return type `dict`

class `MAtom(val, mark, sigma=None)`

Base class for pointed (marked) regular expressions

Used directly to represent atoms (characters). This class is used to obtain Yamada or Asperti automata. There is no evident use for it, outside this module.

Parameters

- **val** – symbol
- **sigma** – alphabet

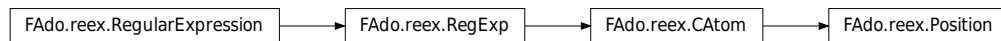
unmark()

Conversion back to `RegExp`

Return type `reex.RegExp` (page 84)

class `Position(val, sigma=None)`

Class for marked regular expression symbols.



Constructor of a regular expression symbol.

Parameters **val** – the actual symbol

setOfSymbols()

Set of symbols that occur in a regular expression..

Returns set of symbols

Return type `set`

unmarked()

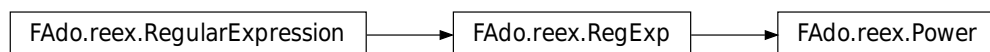
The unmarked form of the regular expression. Each leaf in its syntactical tree becomes a `RegExp()`, the `CEpsilon()` or the `CEmptySet()`.

Returns (general) regular expression

Return type `RegExp` (page 84)

class `Power(arg, n=1, sigma=None)`

Class for Power operation on regular expressions.



alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression. :returns: alphapetic length :rtype: int

Attention: Doesn't include the empty word.

epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type int

first()

First set

Returns first position set

Return type set

followLists()

Follow set

Returns for each key position a set of follow positions

Return type dict

followListsD()

Follow set

Returns for each key position a set of follow positions

Return type dict

last()

Last set

Returns last position set

Return type set

linearForm()

Returns linear form

Return type dict

mark()

Make all atoms maked (tag False)

Return type *RegExp* (page 84)

reversal()

Reversal of RegExp

Return type *reex.RegExp* (page 84)

rpn()

RPN representation

Returns printable RPN representation

Return type str

setOfSymbols()

Returns set of symbols

Return type `set`

snf()

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, `starHeight(((a*b)*+b*)*)` is 3.

Returns number of nested star

Return type `int`

support(*side=True*)

Set of partial derivatives

tailForm()

Returns tail form

Return type `dict`

treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns tree lenght

Return type `int`

class RegExp(*sigma=None*)

Base class for regular expressions.

Variables **Sigma** – alphabet set of strings

FAdo.reex.RegularExpression

FAdo.reex.RegExp



abstract static alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression. :returns: alphapetic length :rtype: int

Attention: Doesn't include the empty word.

compare(*r, cmp_method='compareMinimalDFA', nfa_method='nfaPD'*)

Compare with another regular expression for equivalence.

Parameters

- **r** ([RegExp](#) (page 84)) –

- `cmp_method (str)` –
- `nfa_method (str)` – NFA construction

Returns True if the expressions are equivalent

Return type `bool`

compareMinimalDFA(*r*, *nfa_method*='nfaPosition')

Compare with another regular expression for equivalence through minimal DFAs.

Parameters

- *r* ([RegExp](#) (page 84)) –
- `nfa_method (str)` – NTFA construction

Returns True if equivalent

Return type `bool`

dfaAuPoint()

DFA “au-point” according to Nipkow

Returns “au-point” DFA

Return type [DFA](#) (page 5)

See also:

Andrea Asperti, Claudio Sacerdoti Coen and Enrico Tassi, Regular Expressions, au point. arXiv 2010

See also:

Tobias Nipkow and Dmitriy Traytel, Unified Decision Procedures for Regular Expression Equivalence

dfaBrzozowski(*memo*=None)

Word derivatives automaton of the regular expression

Parameters *memo* – if True memorizes the states already computed

Returns word derivatives automaton

Return type [DFA](#) (page 5)

See also:

J. A. Brzozowski, Derivatives of Regular Expressions. J. ACM 11(4): 481-494 (1964)

dfaNaiveFollow()

DFA that accepts the regular expression’s language, and is obtained from the follow automaton.

Returns DFA follow

Return type [NFA](#) (page 27)

Note: Included for testing purposes.

See also:

Ilie & Yu (Follow Automata, 2003)

dfaYMG()

DFA Yamada-McNaughton-Gluskov according to Nipkow

Returns Y-M-G DFA

Return type [DFA](#) (page 5)

See also:

Tobias Nipkow and Dmitry Traytel, Unified Decision Procedures for Regular Expression Equivalence

static emptysetP()

Whether the regular expression is the empty set.

Return type [bool](#)

abstract static epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type [int](#)

static epsilonP()

Whether the regular expression is the empty word.

Return type [bool](#)

equivP(*other*, *strict=True*)

Test RE equivalence with extended Hopcroft-Karp method

Parameters

- **other** ([RegExp](#) (page 84)) – RE
- **strict** ([bool](#)) – if True checks for same alphabets

Return type [bool](#)

equivalentP(*other*)

Tests equivalence

Parameters **other** ([RegExp](#) (page 84)) – other regexp

Returns True if regexps are equivalent

Return type [bool](#)

evalWordP(*word*)

Verifies if a word is a member of the language represented by the regular expression.

Parameters **word** ([str](#)) – the word

Returns True if word belongs to the language

Return type [bool](#)

static ewp()

Whether the empty word property holds for this regular expression's language.

Return type [bool](#)

abstract first()

First set

Returns first position set

Return type [set](#)

abstract followLists()

Follow set

Returns for each key position a set of follow positions

Return type `dict`

abstract followListsD()

Follow set

Returns for each key position a set of follow positions

Return type `dict`

abstract last()

Last set

Returns last position set

Return type `set`

abstract linearForm()

Returns linear form

Return type `dict`

abstract mark()

Make all atoms marked (tag False)

Return type *RegExp* (page 84)

marked()

Regular expression in which every alphabetic symbol is marked with its Position.

The kind of regular expression returned is known, depending on the literary source, as marked, linear or restricted regular expression.

Returns linear regular expression

Return type *RegExp* (page 84)

See also:

r. McNaughton and H. Yamada, Regular Expressions and State Graphs for Automata, IEEE Transactions on Electronic Computers, V.9 pp:39-47, 1960

..attention: mark and unmark do not preserve the alphabet, neither set the new alphabet

nfaFollow()

NFA that accepts the regular expression's language, whose structure, equiand construction.

Returns NFA follow

Return type *NFA* (page 27)

See also:

Ilie & Yu (Follow Automata, 03)

nfaFollowEpsilon(trim=True)

Epsilon-NFA constructed with Ilie and Yu's method () that accepts the regular expression's language.

Parameters **trim** (*bool*) – if True automaton is trimmed at the end

Returns possibly with epsilon transitions

Return type *NFAe*

Note: The regular expression must be reduced

See also:

Ilie & Yu, Follow automata, Inf. Comp. ,v. 186 (1),140-162,2003

nfaGlushkov()

Position or Glushkov automaton of the regular expression. Recursive method.

Returns NFA position

Return type *NFA* (page 27)

nfaLoc()

Location automaton of the regular expression.

Returns location nfa

Return type *NFA* (page 27)

nfaNaiveFollow()

NFA that accepts the regular expression's language, and is equal in structure to the follow automaton.

Returns NFA follow

Return type *NFA* (page 27)

Note: Included for testing purposes.

See also:

Ilie & Yu (Follow Automata, 2003)

nfaPD(*pdmethod*='nfaPDDAG')

Computes the partial derivative automaton

Parameters **pdmethod** (*str*) – an implementation of the PD automaton. Default value : nfaPDDAG

Returns a PD nfa

Return type *NFA* (page 27)

Attention: for sre classes, CConj and CShuffle use nfaPDNaive directly

nfaPDDAG()

Partial derivative automaton using a DAG for the re and partial derivatives

Returns a PD nfa build using a DAG

Return type *NFA* (page 27)

..seealso:: s.Konstantinidis, A. Machiavelo, N. Moreira, and r. Reis. Partial derivative automaton by compressing regular expressions. DCFS 2021, volume 13037 of LNCS, pages 100–112. Springer, 2022

nfaPDNaive()

NFA that accepts the regular expression's language, and which is constructed from the expression's partial derivatives.

Returns partial derivatives [or equation] automaton

Return type [NFA](#) (page 27)

See also:

V. M. Antimirov, Partial Derivatives of Regular Expressions and Finite Automaton Constructions .Theor. Comput. Sci.155(2): 291-319 (1996)

nfaPDO()

NFA that accepts the regular expression's language, and which is constructed from the expression's partial derivatives.

Returns partial derivatives [or equation] automaton

Return type [NFA](#) (page 27)

Note: optimized version

nfaPSNF()

Position or Glushkov automaton of the regular expression constructed from the expression's star normal form.

Returns Position automaton

Return type [NFA](#) (page 27)

nfaPosition(*lstar=True*)

Position automaton of the regular expression.

Parameters **lstar** (*bool*) – if not None followlists are computed as disjunct

Returns Position NFA

Return type [NFA](#) (page 27)

nfaPre()

Prefix NFA of a regular expression

Returns prefix automaton

Return type [NFA](#) (page 27)

See also:

Maia et al, Prefix and Right-partial derivative automata, 11th CIE 2015, 258-267 LNCS 9136, 2015

notEmptyW()

Witness of non emptiness

Return type word or None

abstract rpn()

RPN representation

Returns printable RPN representation

Return type [str](#)

abstract static setOfSymbols()

Returns set of symbols

Return type [set](#)

setSigma(*symbolset=None, strict=False*)

Set the alphabet for a regular expression and all its nodes

Parameters

- **symbolset** (*set* or *list*) – accepted symbols. If None, alphabet is unset.
- **strict** (*bool*) – if True checks if setOfSymbols is included in symbolSet

..attention: Normally this attribute is not defined in a RegExp()

abstract snf()

Star Normal Form

abstract static starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns number of nested star

Return type *int*

abstract support(*side=True*)

Set of partial derivatives

abstract tailForm()

Returns tail form

Return type *dict*

toDFA()

DFA that accepts the regular expression's language

toNFA(*nfa_method='nfaPDNaive'*)

NFA that accepts the regular expression's language. :param nfa_method:

abstract static treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns tree lenght

Return type *int*

unionSigma(*other*)

Returns the union of two alphabets

Return type *set*

wordDerivative(*word*)

Derivative of the regular expression in relation to the given word, which is represented by a list of symbols.

Parameters **word** (*list*) – list of arbitrary symbols.

Returns regular expression

Return type *RegExp* (page 84)

See also:

J. A. Brzozowski, Derivatives of Regular Expressions. J. ACM 11(4): 481-494 (1964)

class RegularExpression

Abstract base class for all regular expression objects

class SConcat(*arg*, *sigma=None*)

Class that represents the concatenation operation.

**derivative**(*sigma*)

Parameters **sigma** –

Returns

ewp()

Returns

head()

Returns

head_rev()

Returns

linearForm()

Returns

linearFormC()

Returns

partialDerivatives(*sigma*)

Parameters **sigma** –

Returns

partialDerivativesC(*sigma*)

Parameters **sigma** –

Returns

support(*side=True*)

Returns

tail()

Returns

tailForm()

Returns tail form

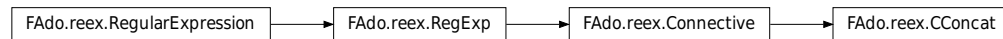
Return type dict

tail_rev()

Returns

class SConj(arg, sigma=None)

Class that represents the conjunction operation.



derivative(sigma)

Parameters sigma –

Returns

ewp()

Returns

linearForm()

Returns

partialDerivatives(sigma)

Parameters sigma –

Returns

partialDerivativesC(sigma)

Parameters sigma –

Returns

support(side=True)

Returns

tailForm()

Returns tail form

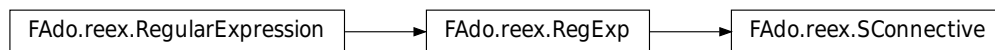
Return type `dict`

class `SConnective`(*arg*, *sigma=None*)

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;

associativity of concatenation; identities σ^* and σ^+ . Connectives are: SDisj: disjunction
SConj: intersection SConcat: concatenation

For parsing use `str2sre`



alphabeticLength()

Returns

epsilonLength()

Returns

first()

First set

Returns first position set

Return type `set`

followLists()

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsD()

Follow set

Returns for each key position a set of follow positions

Return type `dict`

last()

Last set

Returns last position set

Return type `set`

linearForm()

Returns linear form

Return type `dict`

mark()

Make all atoms marked (tag False)

Return type *RegExp* (page 84)

nfaPD(*pmethod='nfaPDNaive'*)

Computes the partial derivative automaton

rpn()

RPN representation

Returns printable RPN representation

Return type `str`

setOfSymbols()

Returns

snf()

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, `starHeight(((a*b)*+b*)*)` is 3.

Returns number of nested star

Return type `int`

abstract support(*side=True*)

Set of partial derivatives

syntacticLength()

Returns

abstract tailForm()

Returns tail form

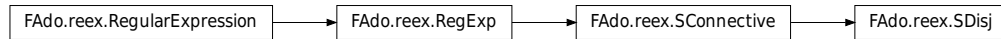
Return type `dict`

treeLength()

Returns

class SDisj(*arg, sigma=None*)

Class that represents the disjunction operation for special regular expressions.



static cross(*ri, s, lists*)

Return type *list*

derivative(*sigma*)

Parameters *sigma* –

Returns

ewp()

Returns

first()

Returns

followLists(*lists=None*)

Parameters *lists* –

Returns

followListsStar(*lists=None*)

Parameters *lists* –

Returns

last()

Returns

linearForm()

Returns

linearFormC()

Returns

partialDerivatives(*sigma*)

Parameters *sigma* –

Returns

partialDerivativesC(*sigma*)

Parameters *sigma* –

Returns

support(*side=True*)

Returns

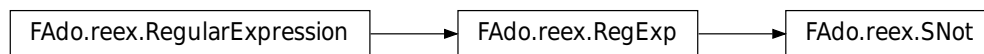
tailForm()

Returns tail form

Return type `dict`

class **SNot**(*arg, sigma=None*)

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection; associativity of concatenation; identities σ^* and σ^+ . SNot: negation



alphabeticLength()

Returns

derivative(*sigma*)

:param *sigma* :return:

epsilonLength()

Returns

ewp()

Returns

first()

First set

Returns first position set

Return type `set`

followLists()

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsD()

Follow set

Returns for each key position a set of follow positions**Return type** `dict`**last()**

Last set

Returns last position set**Return type** `set`**linearForm()****Returns****linearFormC()****Returns****mark()**

Make all atoms marked (tag False)

Return type *RegExp* (page 84)**nfaPD**(*pdmethod*='nfaPDNaive')

Computes the partial derivative automaton

partialDerivatives(*sigma*)**Parameters** *sigma* –**Returns****partialDerivativesC**(*sigma*)**Parameters** *sigma* –**Returns****rpn()**

RPN representation

Returns printable RPN representation**Return type** `str`**setOfSymbols()****Returns****snf()**

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, `starHeight(((a*b)*+b*)*)` is 3.**Returns** number of nested star

Return type `int`

support(*side=True*)

Returns

syntacticLength()

Returns

tailForm()

Returns tail form

Return type `dict`

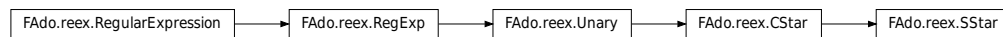
treeLength()

Returns

class SStar(*arg, sigma=None*)

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection; associativity of concatenation; identities σ^* and σ^+ .

SStar: Class that represents Kleene star



derivative(*sigma*)

Parameters *sigma* –

Returns

linearForm()

Returns

nfaPD(*pDMETHOD='nfaPDNaive'*)

Computes the partial derivative automaton

partialDerivatives(*sigma*)

Parameters *sigma* –

Returns

partialDerivativesC(*sigma*)

Parameters **sigma** –

Returns

support(*side=True*)

Returns

class SpecialConstant(*sigma=None*)

Base class for Epsilon and EmptySet



Parameters **sigma** – alphabet

static alphabeticLength()

Returns

derivative(*sigma*)

Parameters **sigma** –

Returns

distDerivative(*sigma*)

Parameters **sigma** – an arbitrary symbol.

Return type regular expression

static epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type `int`

static first(*parent_first=None*)

Parameters **parent_first** –

Returns

followLists(*lists=None*)

Parameters **lists** –

Returns

followListsD(*lists=None*)

Parameters **lists** –

Returns

static followListsStar(*lists=None*)

Parameters *lists* –

Returns

last(*parent_last=None*)

Parameters *parent_last* –

Returns

linearForm()

Returns

mark()

Make all atoms marked (tag False)

Return type *RegExp* (page 84)

partialDerivativesC(*sigma*)

Parameters *sigma* –

Returns

reversal()

Reversal of RegExp

Return type *reex.RegExp* (page 84)

abstract rpn()

RPN representation

Returns printable RPN representation

Return type *str*

static setOfSymbols()

Returns

snf()

Star Normal Form

static starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, `starHeight(((a*b)*+b*)*)` is 3.

Returns number of nested star

Return type *int*

support(*side=True*)

Returns

supportlast(*side=True*)

Returns

tailForm()

Returns

static treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns tree lenght

Return type `int`

unmark()

Conversion back to unmarked atoms :rtype: SpecialConstant

unmarked()

The unmarked form of the regular expression. Each leaf in its syntactical tree becomes a `RegExp()`, the `CEpsilon()` or the `CEmptySet()`.

Return type (general) regular expression

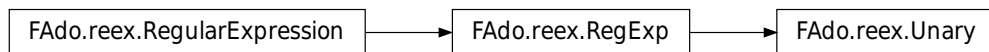
wordDerivative(*word*)

Parameters *word* –

Returns

class Unary(*arg, sigma=None*)

Base class for unary operations: star, option, not, unary shuffle, etc



alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression. :returns: alphapetic length :rtype: int

Attention: Doesn't include the empty word.

abstract epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns number of epsilons

Return type `int`

abstract ewp()

Whether the empty word property holds for this regular expression's language.

Return type `bool`

abstract first(*parent_first=None*)

First set

Returns first position set

Return type `set`

followLists(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

followListsD(*lists=None*)

Follow set

Returns for each key position a set of follow positions

Return type `dict`

abstract last(*parent_last=None*)

Last set

Returns last position set

Return type `set`

abstract linearForm()

Returns linear form

Return type `dict`

mark()

Make all atoms maked (tag False)

Return type *RegExp* (page 84)

reversal()

Reversal of *RegExp*

Return type *reex.RegExp* (page 84)

abstract rpn()

RPN representation

Returns printable RPN representation

Return type `str`

setOfSymbols()

Returns set of symbols

Return type `set`

snf()

Star Normal Form

abstract starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, `starHeight(((a*b)*+b*)*)` is 3.

Returns number of nested star

Return type `int`

treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns tree lenght

Return type `int`

unmark()

Conversion back to RegExp

Return type `reex.__class__`

equivalentP(*first, second*)

Verifies if the two languages given by some representative (DFA, NFA or re) are equivalent

Parameters

- **first** – language
- **second** – language

Return type `bool`

New in version 0.9.6.

powerset(*iterable*)

Powerset of a set. :param iterable: the set :type iterable: list

Returns the powerset

Return type `itertools.chain`

rpn2regexp(*s, sigma=None, strict=False*)

Reads a (simple) RegExp from a RPN representation

```
r ::= .RR | +RR | *r | L | @
L ::= [a-z] | [A-Z]
```

Parameters

- **s** (*str*) – RPN representation
- **strict** (*bool*) – Boolean
- **sigma** (*set*) – alphabet

Return type `reex.RegExp` (page 84)

Note: This method uses python stack... thus depth limitations apply

**str2regexp(*s, parser=Lark(open('/Users/rvr/Work/FAdo/FAdo/regexp_grammar.lark'), parser='lalr',
lexer='contextual', ...), sigma=None, strict=False*)**

Reads a RegExp from string.

Parameters

- **s** (*string*) – the string representation of the regular expression

- **parser** – a parser generator for regexps
- **sigma** (*list or set of symbols*) – alphabet of the regular expression
- **strict** (*boolean*) – if True tests if the symbols of the regular expression are included in sigma

Return type *reex.RegExp* (page 84)

str2sre(*s*, *parser*=*Lark(open('/Users/rvr/Work/FAdo/FAdo/regexp_grammar.lark'), parser='lalr', lexer='contextual', ...)*, *sigma*=*None*, *strict*=*False*)

Reads a sre from string. Arguments as str2regexp.

Return type *reex.sre*

to_s(*r*)

Returns a sre from FAdo regexp.

Parameters *r* (*RegExp* (page 84)) – the FAdo representation regexp for a regular expression.

Return type *RegExp* (page 84)

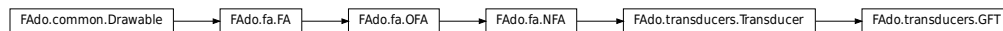
FADO.TRANSUCERS

Finite Tranducer Support

Transducer manipulation.

New in version 1.0.

class GFT
General Form Transducer



addOutput(*sym*)

Add a new symbol to the output alphabet

There is no problem with duplicate symbols because Output is a Set. No symbol Epsilon can be added

Parameters *sym* (*str*) – symbol or regular expression to be added

addTransition(*stsrc*, *wi*, *wo*, *sti2*)

Adds a new transition

Parameters

- **stsrc** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **wi** (*str*) – word consumed
- **wo** (*str*) – word outputed

codeOfTransducer()

Appends into one string the codes of the alphabets and initial and final state sets and the set of transitions

Return type *tuple*

listOfTransitions()

Collects into a sorted list the transitions of the transducer.

Return type set of tuples

toSFT()

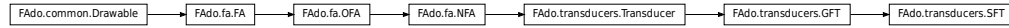
Conversion to an equivalent SFT

rtype: SFT

class NFT

Normal Form Transducer.

Transitions here have labels of the form (s,Epsilon) or (Epsilon,s)



class SFT

Standard Form Transducer

Variables **Output** (*set*) – output alphabet



addEpsilonLoops()

Add a loop transition with epsilon input and output to every state in the transducer.

addTransition(*stsrc*, *symi*, *symo*, *sti2*)

Adds a new transition

Parameters

- **stsrc** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **symi** (*str*) – symbol consumed
- **symo** (*str*) – symbol output

addTransitionProductQ(*src*, *dest*, *ddest*, *sym*, *out*, *futQ*, *pastQ*)

Add transition to the new transducer instance.

Version for the optimized product

Parameters

- **src** – source state
- **dest** – destination state
- **ddest** – destination as tuple
- **sym** – symbol
- **out** – output
- **futQ** (*set*) – queue for later
- **pastQ** (*set*) – past queue

addTransitionQ(*src, dest, sym, out, futQ, pastQ*)

Add transition to the new transducer instance.

Parameters

- **src** – source state
- **dest** – destination state
- **sym** – symbol
- **out** – output
- **futQ** (*set*) – queue for later
- **pastQ** (*set*) – past queue

composition(*other*)

Composition operation of a transducer with a transducer.

Parameters **other** (*SFT* (page 106)) – the second transducer

Return type *SFT* (page 106)

concat(*other*)

Concatenation of transducers

Parameters **other** (*SFT* (page 106)) – the other operand

Return type *SFT* (page 106)

delTransition(*sti1, sym, symo, sti2, _no_check=False*)

Remove a transition if existing and perform cleanup on the transition function's internal data structure.

Parameters

- **symo** – symbol output
- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** – symbol consumed
- **_no_check** (*bool*) – dismiss secure code

deleteState(*sti*)

Remove given state and transitions related with that state.

Parameters **sti** (*int*) – index of the state to be removed

Raises *DFastateUnknown* (page 49) – if state index does not exist

deleteStates(*lstates*)

Delete given iterable collection of states from the automaton.

Parameters **lstates** (*set/list*) – collection of int representing states

dup()

Duplicate of itself :rtype: SFT

Attention: only duplicates the initially connected component

emptyP()

Tests if the relation realized the empty transducer

Return type `bool`

epsilonOutP()

Tests if epsilon occurs in transition outputs

Return type `bool`

epsilonP()

Test whether this transducer has input epsilon-transitions

Return type `bool`

evalWordP(*wp*)

Tests whether the transducer returns the second word using the first one as input

Parameters `wp` (*tuple*) – pair of words

Return type `bool`

evalWordSlowP(*wp*)

Tests whether the transducer returns the second word using the first one as input

Note: original :param tuple wp: pair of words :rtype: bool

functionalP()

Tests if a transducer is functional using Allauzer & Mohri and Béal&Carton&Prieur&Sakarovitch algorithms.

Return type `bool`

See also:

Cyril Allauzer and Mehryar Mohri, Journal of Automata Languages and Combinatorics, Efficient Algorithms for Testing the Twins Property, 8(2): 117-144, 2003.

See also:

M.P. Béal, O. Carton, C. Prieur and J. Sakarovitch. Squaring transducers: An efficient procedure for deciding functionality and sequentiality. Theoret. Computer Science 292:1 (2003), 45-63.

Note: This is implemented using `nonFunctionalW()`

inIntersection(*other*)

Conjunction of transducer and automata: X & Y.

Note: This is a fast version of the method that does not produce meaningful state names.

Note: The resulting transducer is not trim.

Parameters `other` ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automata needs to be operated.

Return type [SFT](#) (page 106)

inIntersectionSlow(*other*)

Conjunction of transducer and automata: X & Y.

Note: This is the slow version of the method that keeps meaningfull names of states.

Parameters **other** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automata needs to be operated.

Return type [SFT](#) (page 106)

inverse()

Switch the input label with the output label.

No initial or final state changed.

Returns Transducer with transitions switched.

Return type [SFT](#) (page 106)

nonEmptyW()

Witness of non emptyness

Returns pair (in-word, out-word)

Return type [tuple](#)

nonFunctionalW()

Returns a witness of non functionality (if is that the case) or a None filled triple

Returns witness

Return type [tuple](#)

outIntersection(other)

Conjunction of transducer and automaton: X & Y using output intersect operation.

Parameters **other** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton used as a filter of the output

Return type [SFT](#) (page 106)

outIntersectionDerived(other)

Naive version of outIntersection

Parameters **other** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton used as a filter of the output

Return type [SFT](#) (page 106)

outputS(s)

Output label coming out of the state i

Parameters **s** ([int](#)) – index state

Return type [set](#)

productInput(other)

Returns a transducer (skeleton) resulting from the execution of the transducer with the automaton as filter on the input.

Note: This version does not use stateIndex() with the price of generating some unreachable sates

Parameters **other** ([NFA](#) (page 27)) – the automaton used as filter

Return type [SFT](#) (page 106)

Changed in version 1.3.3.

productInputSlow(*other*)

Returns a transducer (skeleton) resulting from the execution of the transducer with the automaton as filter on the input.

Note: This is the slow version of the method that keeps meaningful names of states.

Parameters **other** ([NFA](#) (page 27)) – the automaton used as filter

Return type [SFT](#) (page 106)

reversal()

Returns a transducer that recognizes the reversal of the relation.

Returns Transducer recognizing reversal language

Return type [SFT](#) (page 106)

runOnNFA(*nfa*)

Result of applying a transducer to an automaton

Parameters **nfa** ([DFA](#) (page 5) / [NFA](#) (page 27)) – input language to transducer

Returns resulting language

Return type [NFA](#) (page 27)

runOnWord(*word*)

Returns the automaton accepting the output of the transducer on the input word

Parameters **word** – the word

Return type [NFA](#) (page 27)

setInitial(*sts*)

Sets the initial state of a Transducer

Parameters **sts** ([list](#)) – list of states

square()

Conjunction of transducer with itself

Return type [NFA](#) (page 27)

square_fv()

Conjunction of transducer with itself (Fast Version)

Return type [NFA](#) (page 27)

star(*flag=False*)

Kleene star

Parameters **flag** ([bool](#)) – plus instead of star

Returns the resulting Transducer

Return type [SFT](#) (page 106)

toInNFA()

Delete the output labels in the transducer. Translate it into an NFA

Return type [NFA](#) (page 27)

toNFT()

Transformation into Normal Form Transducer

Return type *NFT* (page 106)

toOutNFA()

Returns the result of considering the output symbols of the transducer as input symbols of a NFA (ignoring the input symbol, thus)

Returns the NFA

Return type *NFA* (page 27)

toSFT()

Pacifying rule

Return type *SFT* (page 106)

trim()

Remove states that do not lead to a final state, or, inclusively, that can't be reached from the initial state. Only useful states remain.

Attention: in place transformation

union(*other*)

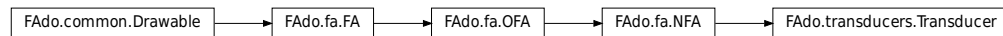
Union of the two transducers

Parameters *other* (*SFT* (page 106)) – the other operand

Return type *SFT* (page 106)

class Transducer

Base class for Transducers



setOutput(*listOfSymbols*)

Set Output

Parameters *listOfSymbols* (*set/list*) – output symbols

succintTransitions()

Collects the transition information in a concat way suitable for graphical representation. :rtype: list of tuples

exception ZERO

Simple exception for functionality testing algorithm

concatN(*x*, *y*)

Concatenation of tuples of words :param *x*: iterable :param *y*: iterable :return: iterable

hypercodeTransducer(*alphabet*, *preserving=False*)

Creates an hypercode property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering

- **alphabet** (*list / set*) – alphabet

Return type *SFT* (page 106)

infixTransducer(*alphabet, preserving=False*)

Creates an infix property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list / set*) – alphabet

Return type *SFT* (page 106)

isLimitExceed(*NFA0Delta, NFA1Delta*)

Decide if the size of NFA0 and NFA1 exceed the limit.

Size of NFA0 is denoted as N, and size of NFA1 is denoted as M. If $N*N*M$ exceeds 1000000, return False, else return True. If both NFA is False, then NFA0 should be NFA, and NFA1 should be Transducer. If both NFA is True, then NFA0 and NFA1 are both NFAs.

Parameters

- **NFA0Delta** (*dict*) – NFA0's transition Delta
- **NFA1Delta** (*dict*) – NFA1's transition Delta

Return type *bool*

outfixTransducer(*alphabet, preserving=False*)

Creates an outfix property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list / set*) – alphabet

Return type *SFT* (page 106)

prefixTransducer(*alphabet, preserving=False*)

Creates an prefix property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list / set*) – alphabet

Return type *SFT* (page 106)

suffixTransducer(*alphabet, preserving=False*)

Creates an suffix property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list / set*) – alphabet

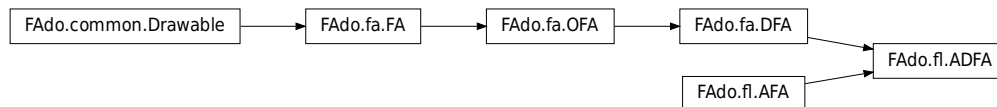
Return type *SFT* (page 106)

Finite languages and related automata manipulation

Finite languages manipulation

class ADFA

Acyclic Deterministic Finite Automata class



Changed in version 1.3.3.

addSuffix(*st*, *w*)

Adds a suffix starting in *st*

Parameters

- **st** (*int*) – state
- **w** (*Word* (page 51)) – suffix

New in version 1.3.3.

Attention: in place transformation

complete(*dead=None*)

Make the ADFA complete

Parameters **dead** (*int*) – a state to be identified as dead state if one was not identified yet

Return type *ADFA* (page 113)

Attention: The object is modified in place

Changed in version 1.3.3.

diss()

Evaluates the dissimilarity language

Return type *FL* (page 118)

New in version 1.2.1.

dissMin(witnesses=None)

Evaluates the minimal dissimilarity language :param dict witnesses: optional witness dictionary :rtype: FL

New in version 1.2.1.

dup()

Duplicate the basic structure into a new ADFA. Basically a copy.deepcopy.

Return type *ADFA* (page 113)

forceToDFA()

Conversion to DFA

Return type *DFA* (page 5)

forceToDFCA()

Conversion to DFCA

Return type *DFA* (page 5)

level()

Computes the level for each state

Returns levels of states

Return type *dict*

New in version 0.9.8.

minDFCA()

Generates a minimal deterministic cover automata from a DFA

Return type *DFCA* (page 118)

New in version 0.9.8.

See also:

Cezar Campeanu, Andrei Păun, and Sheng Yu, An efficient algorithm for constructing minimal cover automata for finite languages, IJFCS

minReversible()

Returns the minimal reversible equivalent automaton

Return type *ADFA* (page 113)

minimal()

Finds the minimal equivalent ADFA

See also:

[TCS 92 pp 181-189] Minimisation of acyclic deterministic automata in linear time, Dominique Revuz

Changed in version 1.3.3.

Returns the minimal equivalent ADFA

Return type *ADFA* (page 113)

minimalP(method=None)

Tests if the DFA is minimal

Parameters **method** – minimization algorithm (here void)

Return type `bool`

Changed in version 1.3.3.

possibleToReverse()

Tests if language is reversible

New in version 1.3.3.

statePairEquiv(*s1*, *s2*)

Tests if two states of a ADFA are equivalent

Parameters

- **s1** (`int`) – state1
- **s2** (`int`) – state2

Return type `bool`

New in version 1.3.3.

toANFA()

Converts the ADFA in a equivalent ANFA

Return type `ANFA` (page 116)

toNFA()

Converts the ADFA in a equivalent NFA

Return type `ANFA` (page 116)

New in version 1.2.

trim()

Remove states that do not lead to a final state, or, inclusively, that can't be reached from the initial state. Only useful states remain.

Attention: in place transformation

wordGenerator()

Creates a random word generator

Returns the random word generator

Return type `RndWGen` (page 119)

New in version 1.2.

class AFA

Base class for Acyclic Finite Automata

FAdo.fl.AFA

note: This is just a container for some common methods. **Not to be used directly!!**

abstract `addState(_)`

Return type `int`

directRank()

Compute rank function

Returns `ranf` map

Return type `dict`

ensureDead()

Ensures that a state is defined as dead

evalRank()

Evaluates the rank map of a automaton

Returns pair of sets of states by rank map, reverse delta accessibility map

Return type `tuple`

getLeaves()

The set of leaves, i.e. final states for last symbols of language words

Returns set of leaves

Return type `set`

ordered()

Orders states names in its topological order

Returns ordered list of state indexes

Return type list of `int`

Note: one could use the `FA.toposort()` method, but special care must be taken with the dead state for the algorithms related with cover automata.

setDeadState(*sti*)

Identifies the dead state

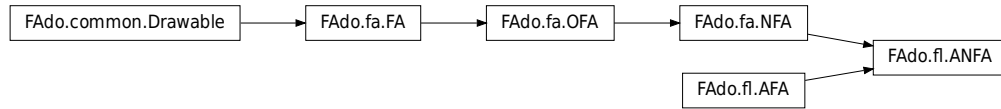
Parameters `sti` (`int`) – index of the dead state

Attention: nothing is done to ensure that the state given is legitimate

Note: without dead state identified, most of the methods for acyclic automata can not be applied

class `ANFA`

Acyclic Nondeterministic Finite Automata class



mergeInitial()

Merge initial states

Attention: object is modified in place

mergeLeaves()

Merge leaves

Attention: object is modified in place

mergeStates(*s1*, *s2*)

Merge state *s2* into state *s1*

Parameters

- **s1** (*int*) – state
- **s2** (*int*) – state

Note: no attempt is made to check if the merging preserves the language of the automaton

Attention: the object is modified in place

moveFinal(*st*, *stf*)

Unsets a set as final transferring transition to another final :param int *st*: the state to be ‘moved’ :param int *stf*: the destination final state

Note: *stf* must be a ‘last’ final state, i.e., must have no out transitions to anywhere but to a possible dead state

Attention: the object is modified in place

DFAtoADFA(*aut*)

Transforms an acyclic DFA into a ADFA

Parameters **aut** (DFA (page 5)) – the automaton to be transformed

Raises *notAcyclic* (page 52) – if the DFA is not acyclic

Returns the converted automaton

Return type *ADFA* (page 113)

class *DFCA*

Deterministic Cover Automata class



property *length*

size of the longest word :rtype: int

Type return

class *FL*(*wordsList=None*, *Sigma=None*)

Finite Language Class

Variables

- **Words** – the elements of the language
- **Sigma** – the alphabet

MADFA()

Generates the minimal acyclical DFA using specialized algorithm

New in version 1.3.3.

See also:

Incremental Construction of Minimal Acyclic Finite-State Automata, J.Daciuk, s.Mihov, B.Watson and r.E.Watson

Return type *ADFA* (page 113)

addWord(*word*)

Adds a word to a FL :type word: Word :rtype: FL

addWords(*wList*)

Adds a list of words to a FL

Parameters *wList* (*list*) – words to add

diff(*other*)

Difference of FL: a - b

Parameters *other* (*FL* (page 118)) – right hand operand

Return type *FL* (page 118)

Raises *FAdoGeneralError* (page 50) – if both arguments are not FL

filter(*automata*)

Separates a language in two other using a DFA of NFA as a filter

Parameters *automata* (*DFA* (page 5) / *NFA* (page 27)) – the automata to be used as a filter

Returns the accepted/unaccepted pair of languages

Return type tuple of FL

intersection(*other*)

Intersection of FL: a & b

Parameters **other** (FL (page 118)) – right hand operand

Raises *FAdoGeneralError* (page 50) – if both arguments are not FL

multiLineAutomaton()

Generates the trivial linear ANFA equivalent to this language

Return type *ANFA* (page 116)

setSigma(*Sigma*, *Strict=False*)

Sets the alphabet of a FL

Parameters

- **Sigma** (*set*) – alphabet
- **Strict** (*bool*) – behaviour

Attention: Unless Strict flag is set to True, alphabet can only be enlarged. The resulting alphabet is in fact the union of the former alphabet with the new one. If flag is set to True, the alphabet is simply replaced.

suffixClosedP()

Tests if a language is suffix closed

Return type *bool*

toDFA()

Generates a DFA recognizing the language

Return type *ADFA* (page 113)

New in version 1.2.

toNFA()

Generates a NFA recognizing the language

Return type *ANFA* (page 116)

New in version 1.2.

trieFA()

Generates the trie automaton that recognises this language

Returns the trie automaton

Return type *ADFA* (page 113)

union(*other*)

union of FL: a | b

Parameters **other** (FL (page 118)) – right hand operand

Return type *FL* (page 118)

Raises *FAdoGeneralError* (page 50) – if both arguments are not FL

class `RndWGen(aut)`

Word random generator class

New in version 1.2.

Parameters `aut` ([ADFA](#) (page 113)) – automata recognizing the language

genRandomTrie(*maxL*, *Sigma*, *safe=True*)

Generates a random trie automaton for a finite language with a given length for max word :param int maxL: length of the max word :param set Sigma: alphabet to be used :param bool safe: should a word of size maxl be present in every language? :return: the generated trie automaton :rtype: ADFA

genRndTrieBalanced(*maxL*, *Sigma*, *safe=True*)

Generates a random trie automaton for a binary language of balanced words of a given length for max word :param int maxL: length of the max word :param set Sigma: alphabet to be used :param bool safe: should a word of size maxl be present in every language? :return: the generated trie automaton :rtype: ADFA

genRndTriePrefix(*maxL*, *Sigma*, *ClosedP=False*, *safe=True*)

Generates a random trie automaton for a finite (either prefix free or prefix closed) language with a given length for max word :param int maxL: length of the max word :param set Sigma: alphabet to be used :param bool ClosedP: should it be a prefix closed language? :param bool safe: should a word of size maxl be present in every language? :return: the generated trie automaton :rtype: ADFA

genRndTrieUnbalanced(*maxL*, *Sigma*, *ratio*, *safe=True*)

Generates a random trie automaton for a binary language of balanced words of a given length for max word

Parameters

- **maxL** (*int*) – length of the max word
- **Sigma** (*set*) – alphabet to be used
- **ratio** (*int*) – the ratio of the unbalance
- **safe** (*bool*) – should a word of size maxl be present in every language?

Returns the generated trie automaton

Return type [ADFA](#) (page 113)

sigmaInitialSegment(*Sigma*, *l*, *exact=False*)

Generates the ADFA recognizing σ^i for $i \leq l$:param set Sigma: the alphabet :param int l: length :param bool exact: only the words with exactly that length? :returns: the automaton :rtype: ADFA

stringToADFA(*s*)

Convert a canonical string representation of a ADFA to a ADFA :param list s: the string in its canonical order :returns: the ADFA :rtype: ADFA

See also:

Marco Almeida, Nelma Moreira, and Rogério Reis. Exact generation of minimal acyclic deterministic finite automata. International Journal of Foundations of Computer Science, 19(4):751-765, August 2008.

FADO.CGF

Context Free Grammars Manipulation.

Basic context-free grammars manipulation for building uniform random generetors

class CFGGenerator(*cfgr, size*)

CFG uniform genetaror

..seealso: **Generating words in a context-free language uniformly at random.** Harry Mairson Information Processing Letters, 49-2, 95-92. 1994

Object initialization :param cfgr: grammar for the random objects :type cfgr: CNF :param size: size of objects :type size: integer

generate()

Generates a new random object generated from the start symbol

Returns object

Return type string

class CFGGrammar(*gram*)

Class for context-free grammars

Variables

- **Rules** – grammar rules
- **Terminals** – terminals symbols
- **Nonterminals** – nonterminals symbols
- **Start** (*str*) – start symbol
- **ntr** – dictionary of rules for each nonterminal

Initialization

Parameters **gram** – is a list for productions; each production is a tuple (LeftHandside, RightHandside) with LeftHandside nonterminal, RightHandside list of symbols, First production is for start symbol

NULLABLE()

Determines which nonterminals $X \rightarrow^* []$

makenonterminals()

Extracts C{nonterminals} from grammar rules.

maketerminals()

Extracts C{terminals} from the rules. Nonterminals must already exist

class **CNF**(*gram*, *mark*='A@')

Chomsky Normal Form. No useless nonterminals or eepsipson rules are ALLOWED... Given a CFG grammar description generates one in CNF Then its possible to random generate words of a given size. Before some pre-calculations are needed.

Initialization

Parameters **gram** – is a list for productions; each production is a tuple (LeftHandside, RightHandside) with LeftHandside nonterminal, RightHandside list of symbols, First production is for start symbol

chomsky()

Transform to CNF

elim_unitary()

Elimination of unitary rules

CYKParserTable(*gramm*, *word*)

Evaluates CYK parser table

Parameters

- **gramm** ([CNF](#) (page 121)) – grammar
- **word** ([str](#)) – word to be parsed

Returns the CYK table

Return type list of lists of symbols

class **REStringRGenerator**(*Sigma*=None, *size*=10, *cfgr*=None, *epsilon*=None, *empty*=None, *ident*='Ti')

Uniform random Generator for reStrings

Uniform random generator for regular expressions. Used without arguments generates an uncollapsible re over {a,b} with size 10. For generate an arbitrary re over an alphabet of 10 symbols of size 100: reStringRGenerator (smallAlphabet(10),100,reGrammar[“g_regular_base”])

Parameters

- **Sigma** ([list/set](#)) – re alphabet (that will be the set of grammar terminals)
- **size** ([int](#)) – word size
- **cfgr** – base grammar
- **epsilon** – if not None is added to a grammar terminals
- **empty** – if not None is added to a grammar terminals

Note: the grammar can have already this symbols

gRules(*rules_list*, *rulesym*='->', *rhssep*=None, *rulesep*='|')

Transforms a list of rules into a grammar description.

Parameters

- **rules_list** ([list](#)) – is a list of rule where rule is a string of the form: Word rulesym Word1 ... Word2 or Word rulesym []
- **rulesym** ([str](#)) – LHS and RHS rule separator
- **rhssep** ([str](#)) – RHS values separator (None for white chars)

- **rulesep** (*str*) – rule separator

Returns a grammar description

Return type *list*

smallAlphabet (*k*, *sigma_base*='a')

Easy way to have small alphabets

Parameters

- **k** – alphabet size (must be less than 52)
- **sigma_base** – initial symbol

Returns alphabet

Return type *list*

FADO.RNDFAP

Random DFA generation (alternative version in python)

ICDFA Random generation binding

New in version 1.0.

class **ICDFArgen**(*n*, *k*, *nd=False*, *pn=1*, *seed=0*)
Generic ICDFA random generator class

Variables

- **n** (*int*) – number of states
- **k** (*int*) – size of the alphabet
- **pn** (*int*) – how more probable shall a non defined transition be?
- **seed** (*int*) – seed for the random generator. Default is to generate a time & system dependent.

See also:

Marco Almeida, Nelma Moreira, and Rogério Reis. Enumeration and generation with a string automata representation. Theoretical Computer Science, 387(2):93-102, 2007

Changed in version 1.3.4: seed added to the random generator

genFinalities()
Generate bit map of final states

Return type *list*

class **ICDFArnd**(*n*, *k*, *seed=0*)
Complete ICDFA random generator class

This is the class for the uniform random generator for Initially Connected DFAs

Variables

- **n** (*int*) – number of states
- **k** (*int*) – size of alphabet
- **seed** (*int*) – seed for the random generator (if 0 uses time as seed)

Note: This is an abstract class, not to be used directly

Changed in version 1.3.4: seed added to the random generator

class `ICDFArndIncomplete`(*n*, *k*, *bias*=None, *seed*=0)

Incomplete ICDFa random generator class

Variables

- **n** (*int*) – number of states
- **k** (*int*) – size of alphabet
- **bias** (*float*) – how often must the gost sink state appear (default None)
- **seed** (*int*) – seed for the random generator (if 0 uses time as seed)

Raises `IllegalBias` (page 50) – if a bias ≥ 1 or ≤ 0 is provided

Changed in version 1.3.4: seed added to the random generator

FADO.RNDADFA

Random ADFA generation

ADFA Random generation binding

New in version 1.2.1.

class `ADFArnd`(*n*, *k*=2, *s*=1)

Sets a random generator for Adfas by sources. By default, *s*=1 to be initially connected

Variables

- **n** (*int*) – number of states
- **k** (*int*) – size of the alphabet
- **s** (*int*) – number of sources

Note: For IC DFA *s*=1

alpha(*n*, *s*, *k*=2)

Number of labeled acyclic initially connected DFA by states and by sources

Parameters

- **k** (*int*) – alphabet size
- **n** (*int*) – number of states
- **s** (*int*) – number of sources

Return type *int*

Note: uses `countAdfabySource`

alpha0(*n*, *s*, *k*=2)

Number of labeled acyclic initially connected DFA by states and by sources

Parameters

- **k** (*int*) – alphabet size
- **n** (*int*) – number of states
- **s** (*int*) – number of sources

Return type *int*

Note: uses `gamma` instead of `beta` or `rndAdfa`

beta($n, s, u, k=2$)

Number of valid configurations of transitions

Parameters

- **k** (*int*) – alphabet size
- **n** (*int*) – number of states
- **s** (*int*) – number of sources
- **u** (*int*) – number of sources of n-s

Return type *int*

Note: not used by alpha or rndAdfa

beta0($n, s, u, k=2$)

Function beta computed using sets

countAdfaBySources($n, s, k=2$)

Number of labelled (initially connected) acyclic automata with n states, alphabet size k, and s sources

Parameters

- **k** (*int*) – alphabet size
- **n** (*int*) – number of states
- **s** (*int*) – number of sources

Raises **IndexError** – if number of states less than number of sources**gamma**(t, u, r)**Parameters**

- **t** (*int*) – size of T
- **u** (*int*) – size of U
- **r** (*int*) – size of r

Return type *int***rndAdfa**(n, s)

Recursively generates a initially connected adfa

Parameters

- **n** (*int*) – number of states
- **s** (*int*) – number of sources

See also:

Felice & Nicaud, CSR 2013 Lncs 7913, pp 88-99, Random Generation of Deterministic Acyclic Automata Using the Recursive Method, DOI:10.1007/978-3-642-38536-0_8

rndNumberSecondSources(n, s)

Uniformly random generates the number of secondary sources

Parameters

- **n** (*int*) – number of states

- **s** (*int*) – number of sources

Return type *int*

rndTransitionsFromSources(*n, s, u*)

Generates the transitions from the sources, ensuring that all secondary sources are connected

Parameters

- **n** (*int*) – number of states
- **s** (*int*) – number of sources
- **u** (*int*) – number of secondary sources

binomial(*n, k*)

Binomial coefficient

Parameters

- **n** – n
- **k** – k

countadfa(*n, k*)

Acyclic (complete) deterministic finite automata structure (unlabeled)

Parameters

- **n** (*int*) – number of states
- **k** (*int*) – alphabetic size

countadfaL(*n, k*)

Acyclic (complete) deterministic finite automata structure (labeled) initially connected (one dead state)

Parameters

- **n** (*int*) – number of states
- **k** (*int*) – alphabetic size

countafa(*n, k*)

(Quasi) Acyclic deterministic finite automata structure (unlabeled)

countafaL(*n, k, r=1*)

(Quasi) Acyclic deterministic finite automata structure (labeled)

Parameters

- **n** (*int*) – number of states
- **k** (*int*) – alphabetic size
- **r** (*int*) – number of dead states

See also:

- V. A. Liskovets. Exact enumeration of acyclic deterministic automata. Discrete Applied Mathematics, 154(3):537-551, March 2006.

surj(*n, m*)

Counting surjections from [n] to [m]

Parameters

- **n** (*int*) – cardinality of domain

- $m(int)$ – cardinality of image

Note: not used by rndAdfa

FADO.COMBOPERATIONS

Several combined operations for DFAs

Combined operations

concatWStar(*fa1*, *fa2*, *strict=False*)

Concatenation combined with star: $(L1.L2^*)$

Parameters

- **fa1** (DFA (page 5)) – first automaton
- **fa2** (DFA (page 5)) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type DFA (page 5)

See also:

Bo Cui, Yuan Gao, Lila Kari, and Sheng Yu. ‘State complexity of two combined operations: Reversal-catenation and star-catenation’. CoRR, abs/1006.4646, 2010.

disjWStar(*f1*, *f2*, *strict=True*)

Union with Star: $(L1 + L2^*)$

Parameters

- **f1** (DFA (page 5)) – first automaton
- **f2** (DFA (page 5)) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type DFA (page 5)

See also:

Yuan Gao and Sheng Yu. ‘State complexity of union and intersection combined with Star and reversal’. CoRR, abs/1006.3755, 2010.

interWStar(*f1*, *f2*, *strict=True*)

Intersection with Star: $(L1 \& L2^*)$

Parameters

- **f1** (DFA (page 5)) – first automaton
- **f2** (DFA (page 5)) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type DFA (page 5)

See also:

Yuan Gao and Sheng Yu. ‘State complexity of union and intersection combined with Star and reversal’. CoRR, abs/1006.3755, 2010.

starConcat(*fa1*, *fa2*, *strict=False*)

Star of concatenation of two languages: $(L1.L2)^*$

Parameters

- **fa1** ([DFA](#) (page 5)) – first automaton
- **fa2** ([DFA](#) (page 5)) – second automaton
- **strict** ([bool](#)) – should the alphabets be necessary equal?

Return type [DFA](#) (page 5)

See also:

Yuan Gao, Kai Salomaa, and Sheng Yu. ‘The state complexity of two combined operations: star of catenation and star of reversal’. Fundamenta Informaticae, 83:75–89, Jan 2008.

starDisj(*fa1*, *fa2*, *strict=False*)

Star of Union of two DFAs: $(L1 + L2)^*$

Parameters

- **fa1** ([DFA](#) (page 5)) – first automaton
- **fa2** ([DFA](#) (page 5)) – second automaton
- **strict** ([bool](#)) – should the alphabets be necessary equal?

Return type [DFA](#) (page 5)

See also:

Arto Salomaa, Kai Salomaa, and Sheng Yu. ‘State complexity of combined operations’. Theor. Comput. Sci., 383(2-3):140–152, 2007.

starInter(*fa1*, *fa2*, *strict=False*)

Star of Intersection of two DFAs: $(L1 \& L2)^*$

Parameters

- **fa1** ([DFA](#) (page 5)) – first automaton
- **fa2** ([DFA](#) (page 5)) – second automaton
- **strict** ([bool](#)) – should the alphabets be necessary equal?

Return type [DFA](#) (page 5)

starInter0(*fa1*, *fa2*, *strict=False*)

Star of Intersection of two DFAs: $(L1 \& L2)^*$

Parameters

- **fa1** ([DFA](#) (page 5)) – first automaton
- **fa2** ([DFA](#) (page 5)) – second automaton
- **strict** ([bool](#)) – should the alphabets be necessary equal?

Return type [DFA](#) (page 5)

See also:

Arto Salomaa, Kai Salomaa, and Sheng Yu. ‘State complexity of combined operations’. Theor. Comput. Sci., 383(2-3):140–152, 2007.

starWConcat(*fa1*, *fa2*, *strict=False*)

Star combined with concatenation: $(L1^*.L2)$

Parameters

- **fa1** ([DFA](#) (page 5)) – first automaton
- **fa2** ([DFA](#) (page 5)) – second automaton
- **strict** ([bool](#)) – should the alphabets be necessary equal?

Return type [DFA](#) (page 5)

See also:

Bo Cui, Yuan Gao, Lila Kari, and Sheng Yu. ‘State complexity of catenation combined with Star and reversal’. CoRR, abs/1008.1648, 2010

FADO.CODES

Code theory module

New in version 1.0.

class `CodeProperty`(*name*, *alph*)

See: K. Dudzinski and s. Konstantinidis: **Formal descriptions of code properties: decidability, complexity, implementation.** International Journal of Foundations of Computer Science 23:1 (2012), 67–85.

Variables `Sigma` – the alphabet

abstract `maximalP`(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- `U` ([DFA](#) (page 5) / [NFA](#) (page 27)) – Universe of permitted words (Σ^* as default)
- `aut` ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton

Return type `bool`

abstract `notMaximalW`(*aut*, *U=None*)

Witness of non maximality

Parameters

- `aut` ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton
- `U` ([DFA](#) (page 5) / [NFA](#) (page 27)) – Universe of permitted words (Σ^* as default)

Returns a witness

Return type `str`

abstract `notSatisfiesW`(*aut*)

Return a witness of non-satisfaction of the property by the automaton language

Parameters `aut` ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton

Returns word witness tuple

Return type `tuple`

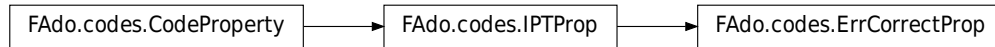
abstract `satisfiesP`(*aut*)

Satisfaction of the property by the automaton language

Parameters `aut` ([NFA](#) (page 27) / [DFA](#) (page 5)) – the automaton

Return type `bool`

class **ErrCorrectProp**(*t*)
Error Correcting Property



Constructor :param SFT aut: Input preserving transducer

notMaximalW(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **aut** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton
- **U** ([DFA](#) (page 5) / [NFA](#) (page 27)) – Universe of permitted words (Σ^* as default)

Return type `bool`

notSatisfiesW(*aut*)

Satisfaction of the code property by the automaton language

Parameters **aut** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton

Return type `tuple`

satisfiesP(*aut*)

Satisfaction of the property by the automaton language

See also:

s. Konstantinidis: Transducers and the Properties of Error-Detection, Error-Correction and Finite-Delay Decodability. Journal Of Universal Computer Science 8 (2002), 278-291.

Parameters **aut** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton

Return type `bool`

ErrDetectProp

alias of [FAdo.codes.IPTProp](#) (page 138)

class **FixedProp**(*name*, *alph*)

Abstract class for fixed properties

abstract **maximalP**(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **U** ([DFA](#) (page 5) / [NFA](#) (page 27)) – Universe of permitted words (Σ^* as default)
- **aut** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton

Return type `bool`

abstract **notMaximalW**(*aut*, *U=None*)

Witness of non maximality

Parameters

- **aut** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton
- **U** ([DFA](#) (page 5) / [NFA](#) (page 27)) – Universe of permitted words (Σ^* as default)

Returns a witness

Return type `str`

abstract notSatisfiesW(*aut*)

Test whether the language is a code.

Parameters **aut** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton

Returns two different factorizations of the same word

Return type tuple of list

abstract satisfiesP(*aut*)

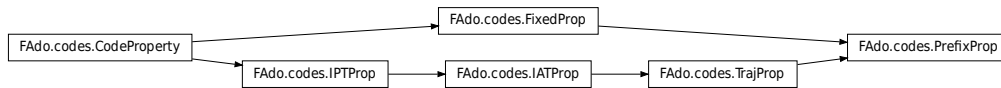
Satisfaction of the property by the automaton language

Parameters **aut** ([NFA](#) (page 27) / [DFA](#) (page 5)) – the automaton

Return type `bool`

class HypercodeProp(*t*)

Hypercode Property



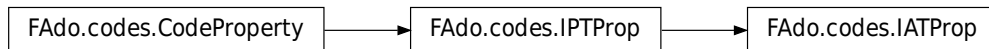
Constructor

Parameters

- **aut** ([DFA](#) (page 5) / [NFA](#) (page 27)) – regular expression over $\{0,1\}$
- **Sigma** (*set*) – the alphabet

class IATProp(*aut, name=None*)

Input Altering Transducer Property



Constructor :param SFT aut: Input preserving transducer

notSatisfiesW(*aut*)

Return a witness of non-satisfaction of the property by the automaton language

Parameters **aut** ([DFA](#) (page 5) / [NFA](#) (page 27)) – the automaton

Returns word witness pair

Return type `tuple`

class `IPTProp`(*aut*, *name=None*)
Input Preserving Transducer Property



Variables

- **Aut** ([SFT](#) (page 106)) – the transducer defining the property
- **sigma** ([set](#)) – alphabet

Constructor :param SFT *aut*: Input preserving transducer

addToCode(*aut*, *N*, *n=2000*)

Returns an NFA and a list *W* of up to *N* words of length *ell*, such that the NFA accepts *L*(*aut*) union *W*, which is an error-detecting language. *ell* is computed from *aut*

Parameters

- **aut** ([NFA](#) (page 27)) – the automaton
- **N** ([int](#)) – the number of words to construct
- **n** ([int](#)) – number of tries when needing a new word

Returns an automaton and a list of strings

Return type `tuple`

makeCode(*N*, *ell*, *s*, *n=2000*, *ov_free=False*)

Returns an NFA and a list *W* of up to *N* words of length *ell*, such that the NFA accepts *W*, which is an error-detecting language. The alphabet to use is $\{0, 1, \dots, s-1\}$. where $s \leq 10$.

Parameters

- **N** ([int](#)) – the number of words to construct
- **ell** ([int](#)) – the codeword length
- **s** ([int](#)) – the alphabet size (must be ≤ 10)
- **n** ([int](#)) – number of tries when needing a new word

Returns an automaton and a list of strings

Return type `tuple`

makeCode0(*N*, *ell*, *s*, *n=2000*, *end=None*, *ov_free=False*)

Returns an NFA and a list *W* of up to *N* words of length *ell*, such that the NFA accepts *W*, which is an error-detecting language. The alphabet to use is $\{0, 1, \dots, s-1\}$. where $s \leq 10$.

Parameters

- **N** ([int](#)) – the number of words to construct
- **ell** ([int](#)) – the codeword length

- **s** (*int*) – the alphabet size (must be ≤ 10)
- **n** (*int*) – number of tries when needing a new word
- **end** (*Word* (page 51)) – a Word or None that should much the end of code words
- **ov_free** (*Boolean*) – if True code words much be overlap free

Returns an automaton and a list of strings

Return type *tuple*

Note: not ov_free and end defined simultaneously Note: end should be a Word

maximalP(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **aut** (*NFA* (page 27)) – the automaton
- **U** (*NFA* (page 27)) – Universe of permitted words (Σ^* as default)

Return type *bool*

notMaxStatW(*aut*, *ell*, *n=2000*, *ov_free=False*)

Returns a word of length ell to add into aut or None; simpler version of function nonMaxStatFEpsW

Parameters

- **aut** (*NFA* (page 27)) – the automaton
- **ell** (*int*) – the length of the words in aut
- **n** (*int*) – number of words to try

Returns a string or None

Return type *str*

notMaximalW(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **aut** (*DFA* (page 5) / *NFA* (page 27)) – the automaton
- **U** (*DFA* (page 5) / *NFA* (page 27)) – Universe of permitted words (Σ^* as default)

Return type *bool*

Raises *PropertyNotSatisfied* (page 51) – if not satisfied

notSatisfiesW(*aut*)

Return a witness of non-satisfaction of the property by the automaton language

Parameters **aut** (*DFA* (page 5) / *NFA* (page 27)) – the automaton

Returns word witness pair

Return type *tuple*

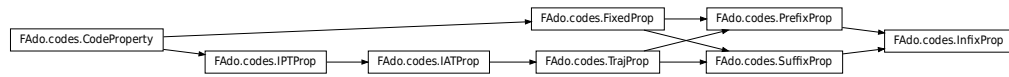
satisfiesP(*aut*)

Satisfaction of the property by the automaton language

Parameters **aut** (*DFA* (page 5) / *NFA* (page 27)) – the automaton

Return type *bool*

class InfixProp(*t*)
Infix Property

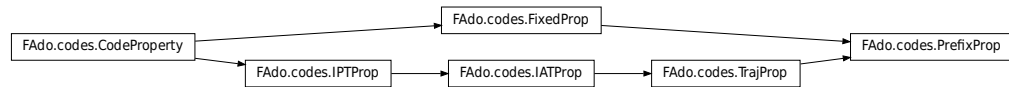


Constructor

Parameters

- **aut** (DFA (page 5)/NFA (page 27)) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

class OutfixProp(*t*)
Outfix Property

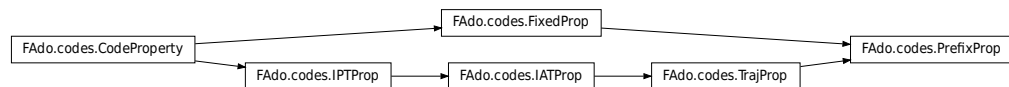


Constructor

Parameters

- **aut** (DFA (page 5)/NFA (page 27)) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

class PrefixProp(*t*)
Prefix Property



Constructor

Parameters

- **aut** (DFA (page 5)/NFA (page 27)) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

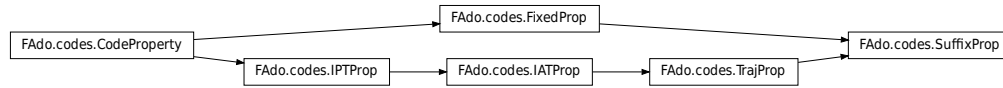
satisfiesPrefixP(*aut*)

Satisfaction of property by the automaton language: faster than satisfiesP

Parameters `aut` ([DFA](#) (page 5)/[NFA](#) (page 27)) – the automaton

Return type `bool`

class `SuffixProp(t)`
Suffix Property

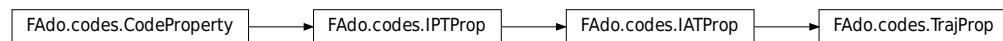


Constructor

Parameters

- `aut` ([DFA](#) (page 5)/[NFA](#) (page 27)) – regular expression over {0,1}
- `Sigma` (`set`) – the alphabet

class `TrajProp(aut, Sigma)`
Class of trajectory properties



Constructor

Parameters

- `aut` ([DFA](#) (page 5)/[NFA](#) (page 27)) – regular expression over {0,1}
- `Sigma` (`set`) – the alphabet

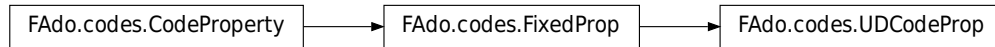
static `trajToTransducer(traj, Sigma)`
Input Altering Transducer corresponding to a Trajectory

Parameters

- `traj` ([NFA](#) (page 27)) – trajectory language
- `Sigma` (`set`) – alphabet

Return type [SFT](#) (page 106)

class `UDCodeProp(alphabet)`
Uniquely decodable Code Property



maximalP(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property :param DFA|NFA *aut*: the automaton :param DFA|NFA *U*: Universe of permitted words (Σ^* as default) :rtype: bool

notSatisfiesW(*aut*)

Test whether the language is a code.

Parameters *aut* (DFA (page 5)/NFA (page 27)) – the automaton

Returns two different factorizations of the same word

Return type tuple of list

satisfiesP(*aut*)

Satisfaction of the code property by the automaton language

Parameters *aut* (DFA (page 5)/NFA (page 27)) – the automaton

Return type bool

buildErrorCorrectPropF(*fname*)

Builds an Error Correcting Property

Parameters *fname* (*str*) – file name

Return type *ErrCorrectProp* (page 135)

buildErrorCorrectPropS(*s*)

Builds an Error Correcting Property from string

Parameters *s* (*str*) – transducer string

Return type *ErrCorrectProp* (page 135)

buildErrorDetectPropF(*fname*)

Builds an Error Detecting Property

Parameters *fname* (*str*) – file name

Return type *ErrDetectProp*

buildErrorDetectPropS(*s*)

Builds an Error Detecting Property from string

Parameters *s* (*str*) – transducer string

Return type *ErrDetectProp*

buildHypercodeProperty(*alphabet*)

Builds a Hypercode Property

Parameters *alphabet* (*set*) – alphabet

Return type *PrefixProp* (page 140)

buildIATPropF(*fname*)

Builds a IATProp from a FAdo SFT file

Parameters *fname* (*str*) – file name

Return type *IATProp* (page 137)

buildIATPropS(*s*)

Builds a IATProp from a FAdo SFT string

Parameters *s* (*str*) – string containing SFT

Return type *IATProp* (page 137)

buildIPTPropF(*fname*)

Builds a IPTProp from a FAdo SFT file

Parameters *fname* (*str*) – file name

Return type *IPTProp* (page 138)

buildIPTPropS(*s*)

Builds a IPTProp from a FAdo SFT string

Parameters *s* (*str*) – file name

Return type *IPTProp* (page 138)

buildInfixProperty(*alphabet*)

Builds a Suffix Code Property

Parameters *alphabet* (*set*) – alphabet

Return type *PrefixProp* (page 140)

buildOutfixProperty(*alphabet*)

Builds a Outfix Code Property

Parameters *alphabet* (*set*) – alphabet

Return type *PrefixProp* (page 140)

buildPrefixProperty(*alphabet*)

Builds a Prefix Code Property

Parameters *alphabet* (*set*) – alphabet

Return type *PrefixProp* (page 140)

buildSuffixProperty(*alphabet*)

Builds a Suffix Code Property

Parameters *alphabet* (*set*) – alphabet

Return type *PrefixProp* (page 140)

buildTrajPropS(*regex*, *sigma*)

Builds a TrajProp from a string RegExp

Parameters

- *regex* (*str*) – the regular expression
- *sigma* (*set*) – alphabet

Return type *TrajProp* (page 141)

buildUDCodeProperty(*alphabet*)

Builds a UDCodeProp (from thin air ;-)

Parameters **alphabet** (*set*) – alphabet**Return type** *UDCodeProp* (page 141)**constructCode**(*n, l, p, ipt=False, seed=None*)

Returns up to n words of length l satisfying the property p, the first one being seed. If *ipt* is True, the property is assumed to be input-preserving transducer type

Return type *list***createInputAlteringSIDTrans**(*n, sigmaSet*)

Create an input-altering SID transducer based

Parameters

- **n** (*int*) – max number of errors
- **sigmaSet** (*set*) – alphabet

Returns a transducer representing the SID channel**Return type** *SFT* (page 106)**editDistanceW**(*auto*)

Compute the edit distance of a given regular language accepted by the NFA via Input-altering transducer.

Parameters **auto** (*NFA* (page 27)) – language recogniser**Returns** The edit distance of the given regular language plus a witness pair**Return type** *tuple***Attention:** language should have at least two words**See also:**

Lila Kari, Stavros Konstantinidis, Steffen Kopecki, Meng Yang. An efficient algorithm for computing the edit distance of a regular language via input-altering transducers. arXiv:1406.1041 [cs.FL]

exponentialDensityP(*aut*)**Checks if language density is exponential** Using breadth first search (BFS)**Parameters** **aut** (*NFA* (page 27)) – the representation of the language**Return type** *bool***Attention:** aut should not have Epsilon transitions**fixedHierSubset**(*x, y*)

Returns whether *x==y*, or the fixed property with name *x* is a subset of *y* Currently (Jan 2015) the fixed properties names are 'UD_codes', 'Prefix_codes', 'Suffix_codes', 'Infix_codes', 'Outfix_codes', 'Hypercodes'

Parameters

- **x** (*tuple*) – first argument
- **y** (*tuple*) – second argument

Return type *bool*

isSubclass(*p1*, *p2*)

Which property (language class) is a subclass of the other (if any). It returns 1 if *p1* is a subclass of *p2*; 2 if *p2* is a subclass of *p1*; 3 if they are equal; 0 otherwise

Parameters

- **p1** (*IPTRProp* (page 138)) – an input preserving transducer property
- **p2** (*IPTRProp* (page 138)) – an input preserving transducer property

Return type *int*

list2string(*lt*, *dy*)

Parameters

- **lt** (*list*) – list of nonnegative integers from some set $\{0, 1, \dots, q-1\}$
- **dy** (*dict*) – mapping from $\{0, 1, \dots, q-1\}$ to some alphabet symbols

Returns string of symbols corresponding to the integers in *lt*

Return type *str*

long2base(*n*, *q*)

Maps *n* to a list of digits corresponding to the base *q* representation of *n* in reverse order

Parameters

- **n** (*int*) – a positive integer
- **q** (*int*) – base to represent *n*

Returns list of *q*-ary ‘digits’, that is, elements of $\{0, 1, \dots, q-1\}$

Return type *list*

notUniversalStatW(*a*, *l*, *maxIter*=20000)

Tests statistically whether the NFA *a* is *l*-non-universal, by evaluating *a* on \leq *maxIter* randomly chosen words of length *l*

Parameters **l** (*int*) – nonnegative integer

Returns (*w*, *i*) where *w* is the word found at *i*-th try; or (*None*, *i*) after *i* tries

Return type *tuple*

pickFrom(*s*, *ell*)

Returns a random string of length *ell* over $\{0, 1, \dots, s-1\}$

Parameters

- **s** (*int*) – the size of the alphabet (should be ≤ 10)
- **ell** (*int*) – the length of the desired string

Returns a string

Return type *str*

symmAndRefl(*t*, *ipt=False*)

Return the transducer $t \mid t.inverse$, if *ipt* is **True;** return the transducer $t \mid t.inverse \mid id$, otherwise

Return type *SFT* (page 106)

unionOfIDs(*first*, *second*)

Returns the “union” of the two sets: property ID X and propoerty ID Y. The result is the set union minus any element in one set that names a property containing a property named in the other set

Parameters

- **first** (*set*) – first argument
- **second** (*set*) – second argument

Return type *set*

Set Specification Transducer support

New in version 1.4.

NFA2SSFA(*aut*)

Transforms a NFA to and SSFA

Parameters *aut* ([fa.NFA](#) (page 27)) – NFA

Return type [SSFA](#) (page 148)

class PSP

Relation pair of set specifications

class PSPDiff(*arg1*, *arg2*)

Relation pair of two set specifications (constrained by non equality)

inIntersection(*other*, *alph*)

Evaluates the intersect on input wit anothe Set Specification

Parameters

- **other** ([SetSpec](#) (page 151)) – the other
- **alph** ([set](#)) – alphabet

Return type [PSP](#) (page 147)

class PSPEqual(*arg1*)

Relation pair of two set specifications (constrained by equality)

inIntersection(*other*, *alph*)

Evaluates the intersect on input wit anothe Set Specification

Parameters

- **other** ([SetSpec](#) (page 151)) – the other
- **alph** ([set](#)) – alphabet

Return type [PSP](#) (page 147)

class PSPVanila(*arg1*, *arg2*)

Relation pair of two set specifications

alphabet()

The covering alphabet of a PSP

Return type `set`

behaviour(*sigma*)

Expansion of a PSP

Return type (`set`, `set`)

inIntersection(*other*, *alph*)

Evaluates the intersect on input with another Set Specification

Parameters

- **other** (`SetSpec` (page 151)) – the other
- **alph** (`set`) – alphabet

Return type `PSP` (page 147)

inverse()

Inverse of a PSP

Return type `PSPVanila` (page 147)

isAInvariant()

Is this an alphabet invariant PSP?

Return type `bool`

class SSAnyOf

Set specification for ‘any’

SSConditionalNoneOf(*oset*)

Auxiliary function that coalesces an SSNoneOf into an SSAnyOf if oset is empty

class SSEmpty

class SSEpsilon

class SSFA(*alph*)

NFAs with Set Specifications as transition labels

Parameters **alph** – alphabet

addEpsilonLoops()

Add epsilon loops to every state

Attention: in-place modification

New in version 1.0.

addTransition(*sti1*, *spec*, *sti2*)

Add af Set Specification transition

Parameters

- **sti1** (`int`) – start state index
- **sti2** (`int`) – end state index
- **spec** (`SetSpec` (page 151)) – symbolic spec

emptyP()

Tests if the automaton accepts an empty language

Return type `bool`

New in version 1.0.

epsilonP()

Whether this NFA has epsilon-transitions

Return type `bool`**toNFA()**

Dummy identity function

Return type *NFA* (page 27)**witness()**

Witness of non emptiness

Returns `word`**Return type** `str`**class** `SSNoneOf(oset)`

Set specification for ‘none of...’

class `SSOneOf(oset)`

Set specification for ‘one of...’

class `SST(sigma=None)`

SFT with set specification labels

**addEpsilonLoops()**

Add a loop transition with epsilon input and output to every state in the transducer.

addToSigma(*sym*)

Adds a new symbol to the alphabet (it it is not already there)

Parameters **sym** (*unicode*) – symbol to add**Return type** `int`**Returns** the index of the new symbol**addTransition(*stsrc*, *pair*, *sti2*)****Parameters** **sti2** – int**addTransitionProductQ(*src*, *dest*, *ddest*, *sym*, *futQ*, *pastQ*)**

Add transition to the new transducer instance.

Version for the optimized product

Parameters

- **src** – source state
- **dest** – destination state
- **ddest** – destination as tuple
- **sym** – symbol
- **futQ** (*set*) – queue for later
- **pastQ** (*set*) – past queue

epsilonOutP()

Tests if epsilon occurs in transition outputs

Return type *bool*

epsilonP()

Test whether this transducer has input epsilon-transitions

Return type *bool*

inIntersection(*other*)

Conjunction of transducer and automata: X & Y.

Note: This is a fast version of the method that does not produce meaningfull state names.

Note: The resulting transducer is not trim.

Parameters **other** (*DFA* (page 5) / *NFA* (page 27)) – the automata needs to be operated.

Return type *SFT* (page 106)

inverse()

Switch the input label with the output label.

No initial or final state changed.

Returns Transducer with transitions switched.

Return type *SFT* (page 106)

nonEmptyW()

Witness of non emptyness

Returns pair (in-word, out-word)

Return type *tuple*

outIntersection(*other*)

Conjunction of transducer and automaton: X & Y using output intersect operation.

Parameters **other** (*DFA* (page 5) / *NFA* (page 27)) – the automaton used as a filter of the output

Return type *SFT* (page 106)

productInput(*other*)

Returns a transducer (skeleton) resulting from the execution of the transducer with the automaton as filter on the input.

Note: This version does not use stateIndex() with the price of generating some unreachable sates

Parameters *other* ([SSFA](#) (page 148)) – the automaton used as filter

Return type [SST](#) (page 149)

Changed in version 1.3.3.

reversal()

Returns a transducer that recognizes the reversal of the relation.

Returns Transducer recognizing reversal language

Return type [SFT](#) (page 106)

toInNFA()

Delete the output labels in the transducer. Translate it into an NFA

Return type [NFA](#) (page 27)

toInSSFA()

Delete the output labels in the transducer. Translate it into an SSFA

Return type [SSFA](#) (page 148)

toOutNFA()

Returns the result of considering the output symbols of the transducer as input symbols of a NFA (ignoring the input symbol, thus)

Returns the NFA

Return type [NFA](#) (page 27)

toOutSSFA()

Returns the result of considering the output symbols of the transducer as input symbols of a SSFA (ignoring the input symbol, thus)

Returns the SSFA

Return type [SSFA](#) (page 148)

toSFT()

Expands a SST to an SFT

Return type [SFT](#) (page 106)

toXSSFA(*side*)

Skeleton of a method that extracts both left & right language of a PSP

class SetSpec

Set Specification labels

Polynomial Random Approximation Algorithms

class `GenWordDis`(*f*, *alf*, *e*)

Word generator according to a given distribution funtion (used for sizes), for prax test

Variables

- **sigma** (*list*) – alphabet
- **pf** (*function*) – distribution function
- **e** (*float*) – acceptable error
- **n_tries** (*int*) – size of the sample
- **max_length** (*int*) – maximal size of the words sampled
- **dist** (*list*) – cumulative probability for each size consedered (up to max_lengt)

f_dirichlet(*n*, *d=1*, *t=2.000001*)

Dirichlet distribution function

Parameters

- **n** (*int*) – evaluation point
- **d** (*int* | *float*) – displacement
- **t** (*int* | *float*) –

Return type *float*

New in version 2.0.4.

f_laplace(*n*, *d=1*, *z=0.99*)

Laplace distribution function

Parameters

- **n** (*int*) – evaluation point
- **d** (*int*) – displacement
- **z** (*float*) – a number $9 < z < 1$

Return type *float*

Raises *FAdoGeneralError* (page 50) – if *z* is null

minI(*a*, *t*, *u=None*)

An operator that returns a *t*-independent language containing *L*(*a*)

Parameters

- **a** ([FA](#) (page 21)) – the initial automaton
- **t** ([Transducer](#) (page 111)) – input-altering transducer
- **u** ([FA](#) (page 21) / *None*) – universe to consider

Return type [NFA](#) (page 27)

prax_parameters(*g*)

Prax parameters for a given experiment

prax_univ_nfa(*g, a, debug=False*)

Polynomial Randomized Approximation (PRAX) to NFA universality

Parameters

- **a** ([FA](#) (page 21)) – the automaton being tested
- **e** (*float*) – admissible error
- **prob_func** – probability function
- **alpha** (*set*) – alphabet of the language

Return type *bool*

See also:

S.Konstantinidis, M.Mastnak, N.Moreira, R.Reis. Approximate NFA Universality and Related Problems Motivated by Information Theory, arXiv, 2022.

New in version 2.0.4.

random() → *x* in the interval [0, 1).

unive_index(*g, aut, prop*)

Universality index of a automaton for a given distribution

Parameters

- **g** ([GenWordDis](#) (page 153)) – distribution
- **aut** ([FA](#) (page 21)) – automaton

Returns universality index

Return type *float*

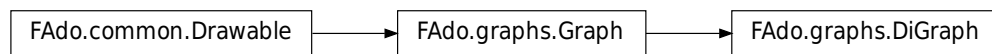
FADO.GRAPHS

Graph support

Basic Graph object support and manipulation

class DiGraph

Directed graph base class



addEdge(v1, v2)

Adds an edge

Parameters

- **v1** (*int*) – vertex 1 index
- **v2** (*int*) – vertex 2 index

static dotDrawEdge(st1, st2, sep='\n')

Draw a transition in Dot Format

Parameters

- **st1** (*str*) – starting state
- **st2** (*str*) – ending state
- **sep** (*str*) – separator

Return type *str*

dotDrawVertex(sti, sep='\n')

Draw a Vertex in Dot Format

Parameters

- **sti** (*int*) – index of the state
- **sep** (*str*) – separator

Return type *str*

dotFormat(*size*='20,20', *direction*='LR', *sep*='\n', *strict*=False, *maxLblSz*=10)

A dot representation

Parameters

- **direction** (*str*) – direction of drawing
- **size** (*str*) – size of image
- **sep** (*str*) – line separator
- **maxLblSz** – max size of labels before getting removed
- **strict** – use limitations of label sizes

Returns the dot representation

Return type *str*

New in version 0.9.6.

Changed in version 0.9.8.

inverse()

Inverse of a digraph

class DiGraphVm

Directed graph with marked vertices

Variables **MarkedV** (*set*) – set of marked vertices



markVertex(*v*)

Mark vertex *v*

Parameters **v** (*int*) – vertex

class Graph

Graph base class

Variables

- **Vertices** (*list*) – Vertices' names
- **Edges** (*set*) – set of pairs (always sorted)



addEdge(*v1*, *v2*)

Adds an edge :param int *v1*: vertex 1 index :param int *v2*: vertex 2 index :raises GraphError: if edge is loop

addVertex(*vname*)

Adds a vertex (by name)

Parameters *vname* – vertex name

Returns vertex index

Return type `int`

Raises [*DuplicateName*](#) (page 50) – if *vname* already exists

abstract dotFormat(*size*)

Some dot representation

Parameters

- **size** (*str*) – size parameter for dotviz
- **filename** (*str*) – filename
- **direction** (*str*) –
- **strict** (*bool*) –
- **maxlblsz** (*int*) –
- **sep** (*str*) –

Returns: str:

vertexIndex(*vname*, *autoCreate=False*)

Return vertex index

Parameters

- **autoCreate** (*bool*) – auto creation of non existing states
- **vname** – vertex name

Return type `int`

Raises [*GraphError*](#) (page 50) – if *vname* not found

SMALL TUTORIAL

17.1 A small tutorial for FAdo

FAdo system is a set tools for regular languages manipulation.

Regular languages can be represented by regular expressions (regex) or finite automata, among other formalisms. Finite automata may be deterministic (DFA) or non-deterministic (NFA). In FAdo these representations are implemented as Python classes. A full documentation of all classes and methods is [here](#).

To work with FAdo, after installation, import the following modules on a Python interpreter:

```
[52]: >>> from FAdo.fa import *
>>> from FAdo.reex import *
>>> from FAdo.fio import *
```

The module `fa` implements the classes for finite automata and the module `reex` the classes for regular expressions. The module `fio` implements methods for IO of automata and related models.

17.1.1 General conventions

Methods which name ends in `P` test if the object verifies a given property and return `True` or `False`.

17.1.2 Finite Automata

The top class for finite automata is the class `FA`, which has two main subclasses: `OFA` for one way finite automata and the class `TFA` for two-way finite automata. The class `OFA` implements the basic structure of a finite automaton shared by DFAs and NFAs. This class defines the following attributes:

Sigma: the input alphabet (set)

States: the list of states. It is a list such that each state is referred by its index whenever it is used (transitions, `Final`, etc).

Initial: the initial state (or a set of initial states for NFA). It is an index or list of indexes.

Final: the set of final states. It is a list of indexes.

In general, one should not create instances (objects) of class `OFA`. The class `DFA` and `NFA` implement DFAs and NFAs, respectively. The class `GFA` implements generalized NFAs that are used in the conversion between finite automata and regular expressions. All three classes inherit from class `OFA`.

For each class there are special methods for add/delete/modify alphabet symbols, states and transitions.

17.1.3 DFAs

The following example shows how to build a DFA that accepts the words of $\{0,1\}^*$ that are multiples of 3.

```
[53]: >>> m3 = DFA()
>>> m3.setSigma(['0','1'])
>>> m3.addState('s1')
>>> m3.addState('s2')
>>> m3.addState('s3')
>>> m3.setInitial(0)
>>> m3.addFinal(0)
>>> m3.addTransition(0, '0', 0)
>>> m3.addTransition(0, '1', 1)
>>> m3.addTransition(1, '0', 2)
>>> m3.addTransition(1, '1', 0)
>>> m3.addTransition(2, '0', 1)
>>> m3.addTransition(2, '1', 2)
```

It is now possible, for instance, to see the structure of the automaton or to test if a word is accepted by it.

```
[54]: >>> m3
```

```
[54]: DFA((['s1', 's2', 's3'], ['1', '0'], 's1', ['s1'], "(['s1', '0', 's1'), ('s1', '1', 's2', '1', 's3'])")
```

```
[55]: >>> m3.evalWordP("011")
```

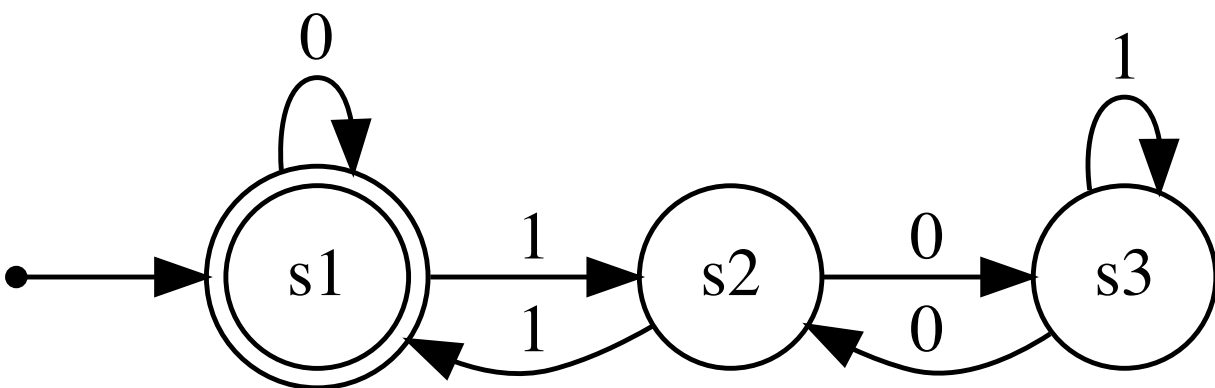
```
[55]: True
```

```
[56]: >>> m3.evalWordP("1011")
```

```
[56]: False
```

If graphviz is installed it is also possible to display the diagram of an automaton as follows:

```
[57]: >>> m3.display()
```



Instead of constructing the DFA directly we can load (and save) it in a simple text format. For the previous automaton the description will be:

```
@DFA 0
0 1 1
0 0 0
1 1 0
1 0 2
2 1 2
2 0 1
```

Then, if this description is saved in file `mul3.fa`, we have

```
[58]: a = "@DFA 0 \n 0 1 1 \n 0 0 0 \n 1 1 0 \n 1 0 2 \n 2 1 2 \n 2 0 1\n"
with open("mul3.fa", 'w') as f: f.write(a)
```

```
[59]: >>> m3 = readFromFile('mul3.fa')
```

As the set of states is represented by a Python list, the list method `len` can be used to determine the number of states of a FA:

```
[60]: >>> len(m3.States)
```

```
[60]: 3
```

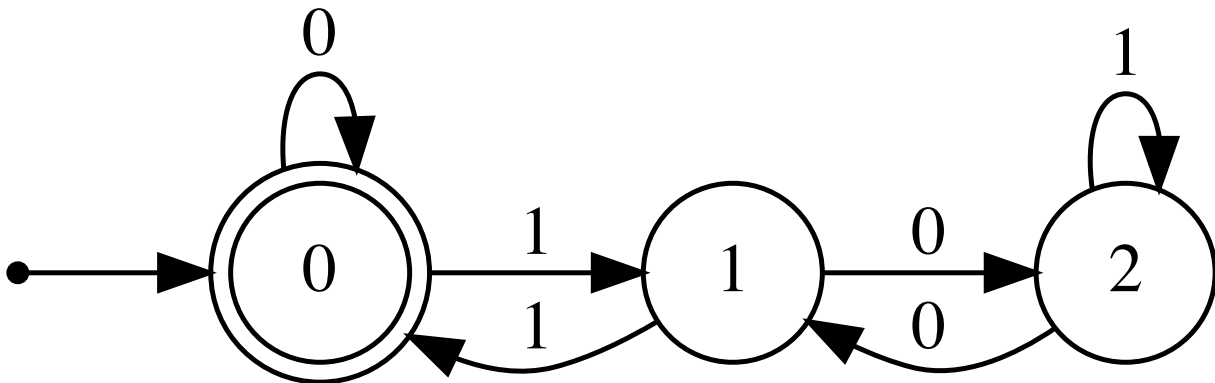
For the number of Transitions the `countTransitions()` method must be used

```
[61]: >>> m3.countTransitions()
```

```
[61]: 6
```

Automata can be displayed

```
[62]: >>> m3.display()
```



To minimize a DFA any of the minimization algorithms implemented can be used:

```
[63]: >>> min = m3.minimalHopcroft()
>>> min
```

```
[63]: DFA((['0', '1', '2'], ['1', '0'], '0', ['0'], "(['0', '1', '1'), ('0', '0', '0'), ('1',
→ '1', '0'), ('1', '0', '2'), ('2', '1', '2'), ('2', '0', '1'))")
```

In this case, the DFA was already minimal so `min` has the same number of states as `m3`.

Several (regularity preserving) operations of DFAs are implemented in FAdo: boolean (union (`|` or **or**), intersection (`&` or **and**) and complementation (`~` or **invert**)), concatenation (`concat`), reversal (`reversal`) and star (`star`).

```
[64]: >>> u = m3 | ~m3
>>> u
[64]: DFA((['0', '5', '10'], ['1', '0'], '0', ['0', '5', '10'], "[(('0', '1', '5'), ('0', '0',
↪ '0'), ('5', '1', '0'), ('5', '0', '10'), ('10', '1', '10'), ('10', '0', '5'))]"))

[65]: >>> m = u.minimal()
>>> m
[65]: DFA((['0'], ['1', '0'], '0', ['0'], "[(('0', '1', '0'), ('0', '0', '0'))]"))
```

State names can be renamed in-place using:

```
[66]: >>> m.renameStates(range(len(m)))
[66]: DFA((['0'], ['1', '0'], '0', ['0'], "[(0, '1', 0), (0, '0', 0)]"))
```

Notice that `m` recognize all words over the alphabet `{0, 1}`. It is possible to generate a word recognisable by an automata (witness)

```
[67]: >>> u.witness()
[67]: '@epsilon'
```

In this case this allows to ensure that `u` recognizes the empty word.

This method is also useful for obtain a witness for the difference of two DFAs (`witnessDiff`).

To test if two DFAs are equivalent the the operator `==` (`equivalenceP`) can be used.

17.1.4 NFAs

NFAs can be built and manipulated in a similar way. There is no distinction between NFAs with and without epsilon-transitions. But it is possible to test if a NFA has epsilon-transitions and convert between a NFA with epsilon-transitions to a (equivalent) NFA without them.

17.1.5 Converting between NFAs and DFAs

The method `toDFA` allows to convert a NFA to an equivalent DFA by the subset construction method. The method `toNFA` migrates trivially a DFA to a NFA.

17.1.6 Regular Expressions

A regular expression can be a symbol of the alphabet, the empty set (@emptyset), the empty word (@epsilon) or the concatenation or the union (+) or the Kleene star (*) of a regular expression. Examples of regular expressions are $a+b$, $(a+ba)^*$, and $(@epsilon+a)(ba+ab+@emptyset)$.

The class `regexp` is the base class for regular expressions and is used to represent an alphabet symbol. The classes `epsilon` and `emptyset` are the subclasses used for the empty set and empty word, respectively. Complex regular expressions are `concat`, `disj`, and `star`.

As for DFAs (and NFAs) we can build directly a regular expressions as a Python class:

```
[68]: >>> r = CStar(CDisj(CAtom("a"), CConcat(CAtom("b"), CAtom("a"))))
>>> print(r)

(a + (b a))*
```

But we can convert a string to a `regexp` class or subclass, using the method `str2regexp`.

```
[69]: >>> r = str2regexp("(a+ba)*")
>>> print(r)

(a + (b a))*
```

For regular expressions there are several measures available: alphabetic size, (parse) tree size, string length, number of epsilons and star height. It is also possible to explicitly associate an alphabet to regular expression (even if some symbols do not appear in it) (`setSigma`).

There are several algebraic properties that can be used to obtain equivalent regular expressions of a smaller size. The method `reduced` transforms a regular expression into one equivalent without some obvious unnecessary epsilons, emptysets or stars.

Several methods that allows the manipulation of derivatives (or partial derivatives) by a symbol or by a word are implemented. However, the basic class `RegEx` does not deal with regular expressions module ACI properties (associativity, commutativity and idempotence of the union), so it is not possible to obtain all word derivatives of a given regular expression. This is not the case for partial derivatives.

To test if two regular expressions are equivalent the method `compare` can be used.

```
[70]: >>> r.compare(str2regexp("(a*(ba)*a)*"))
[70]: True
```

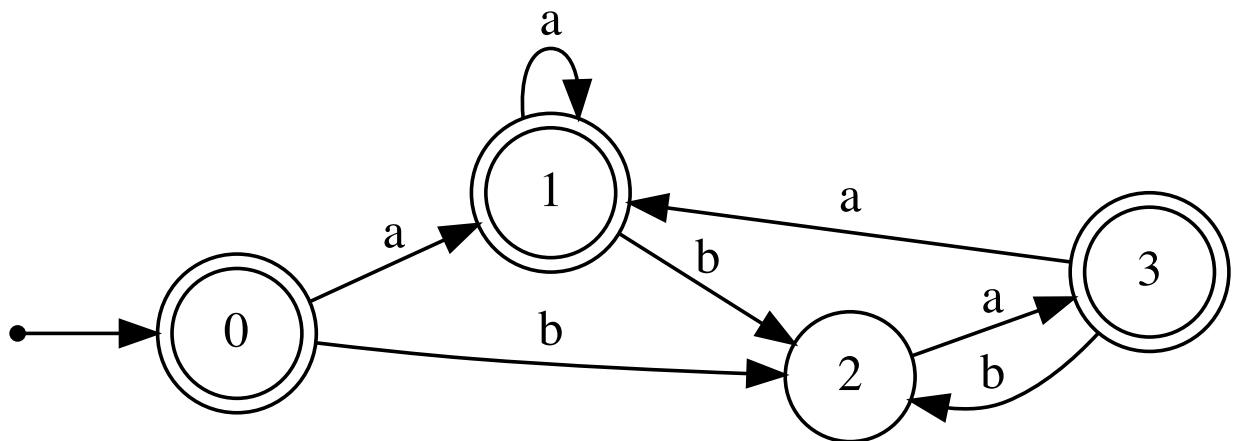
There several methods to convert regular expressions into equivalent nfes or dfes: Thompson, Position/Glushkov, Partial Derivatives (PD), etc.

```
[71]: >>> r.nfaThompson()
[71]: NFA([["(0, (0, '0'))", "(0, (0, '1'))", "(0, (1, 0))", "(0, (1, 1))", "(1, 0)", "(1, 1)",
→ "(1, 2)", "(1, 3)", '8', '9'], ['a', 'b'], ['8'], ['9'], ["((0, (1, 0)), 'a', (0, (1, 1))),
→ ((0, (0, '0')), '@epsilon', (0, (1, 0))), ((0, (0, '0')), '@epsilon', (1, 0)),
→ ((0, (1, 1)), '@epsilon', (0, (0, '1'))), ((1, 0), 'b', (1, 1)), ((1, 2), 'a', (1, 3)),
→ ((1, 1), '@epsilon', (1, 2)), ((1, 3), '@epsilon', (0, (0, '1'))), ('8', '@epsilon',
→ (0, (0, '0'))), ('8', '@epsilon', '9'), ('9', '@epsilon', '8'), ((0, (0, '1')),
→ '@epsilon', '9')]])
```

```
[72]: >>> r.nfaPosition()
```

```
[72]: NFA((['Initial', "('a', 1)", "('b', 2)", "('a', 3)"], ['a', 'b'], ['Initial'], ['Initial',
  ↳ '(', "('a', 1)", "('a', 3)"], '(\Initial\, \'a\', "("\'a\', 1)"), (\Initial\, \'b\',
  ↳ "("\'b\', 2)"), ("\'b\', 2)", \'a\', "("\'a\', 3)"), ("\'a\', 3)", \'b\', "("\'b\', 2)
  ↳ "), ("\'a\', 3)", \'a\', "("\'a\', 1)"), ("\'a\', 1)", \'a\', "("\'a\', 1)"), ("\'a\',
  ↳ 1)", \'b\', "("\'b\', 2)"))])
```

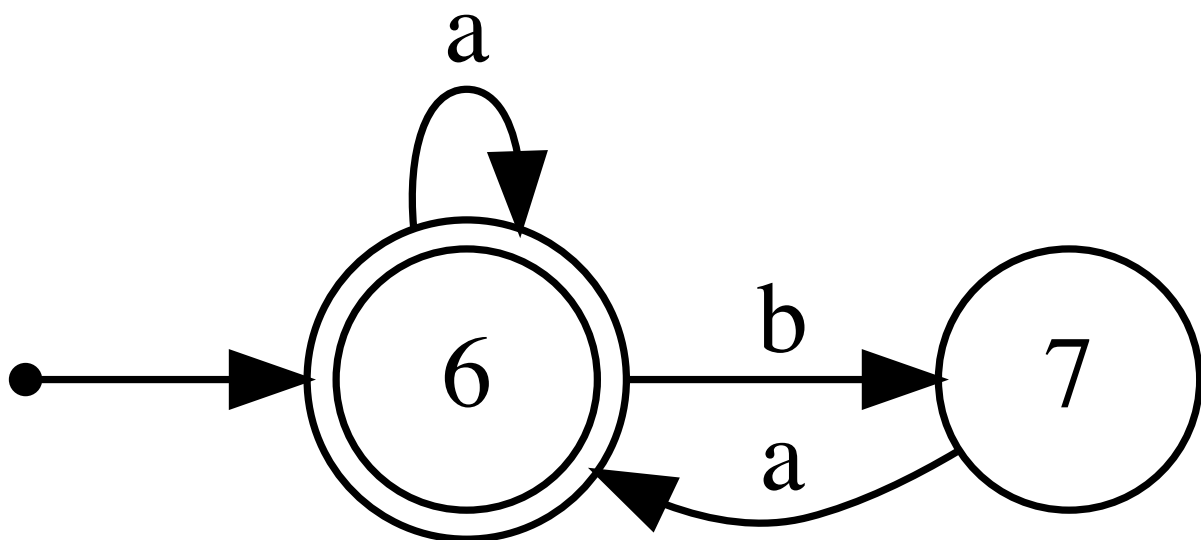
```
[73]: >>> r.nfaPosition().display()
```



```
[74]: >>> r.nfaPD()
```

```
[74]: NFA((['6', '7'], ['a', 'b'], ['6'], ['6'], "[ (6, 'a', 6), (6, 'b', 7), (7, 'a', 6)]"))
```

```
[75]: >>> r.nfaPD().display()
```



17.1.7 Converting Finite Automata to Regular Expressions

Import module conversions

```
[76]: from FAdo.conversions import *
```

For pedagogical purposes, it is implemented a recursive method that constructs a regular expression equivalent to a given DFA (DFA2regexpDijkstra).

```
[77]: >>> print(DFA2regexpDijkstra(m3))

((0 + ((@epsilon + 0) (0* (@epsilon + 0)))) + ((1 + ((@epsilon + 0) (0* 1))) ((1 (0* 1))*) (1 + (1 (0* (@epsilon + 0)))))) + (((1 + ((@epsilon + 0) (0* 1))) ((1 (0* 1))*) (0)) ((1 + (0 ((1 (0* 1))*) 0))) (0 ((1 (0* 1))*) (1 + (1 (0* (@epsilon + 0))))))
```

Methods based on state elimination techniques are usually more efficient, and produces much smaller regular expressions. We have implemented several heuristics for the elimination order.

```
[78]: >>> print(FA2regexpCG(m3))

((0 + (1 1)) + (((1 0) (1 + (0 0))*) (0 1)))*
```

NFA(((['', '', '', '0', '1', '2', '3', '8', '9'], ['a', 'b'], ['8'], ['9'], ["('', '@epsilon', ' '), ('', '@epsilon', 0), ('', '@epsilon', '9'), ('', 'a', ' '), ('', '@epsilon', ' '), (0, 'b', 1), (1, '@epsilon', 2), (2, 'a', 3), (3, '@epsilon', ' '), ('8', '@epsilon', ' '), ('8', '@epsilon', '9'), ('9', '@epsilon', '8')]))

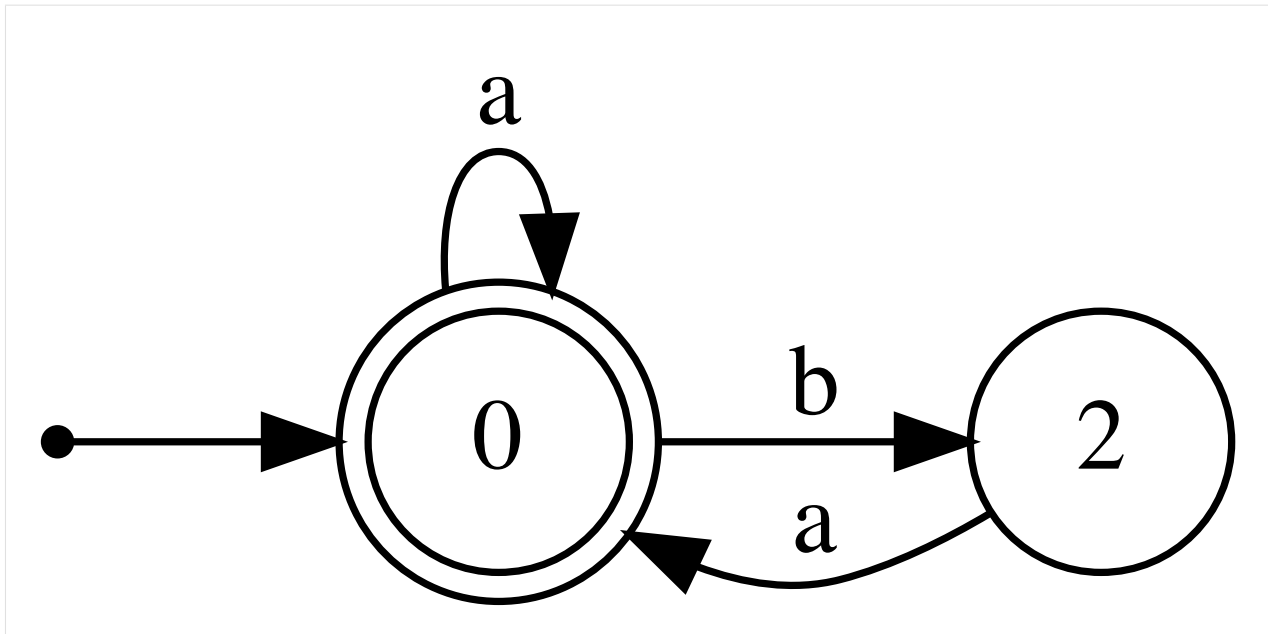
17.1.8 General Example

Considering the several methods described before it is possible to convert between the different equivalent representations of regular languages, as well to perform several regularity preserving operations.

```
[79]: >>> r.nfaPosition().toDFA().minimal(complete=False)
```

```
[79]: DFA(((['0', '2'], ['a', 'b'], '0', ['0'], "[( '0', 'a', '0'), ('0', 'b', '2'), ('2', 'a', '0') ]"))
```

```
[80]: >>> r.nfaPosition().toDFA().minimal(complete=False).display()
```



```
[81]: >>> m3 == FA2regexpCG(m3).nfaPD().toDFA().minimal()
```

```
[81]: True
```

17.1.9 More classes and modules

Several other classes and modules are also available, including:

class `ICDFArnd` (module `rndfa.py`): Random DFA generation

class `FL` (module `fl.py`): special methods for finite languages

module `comboperations.py`: implementation of several algorithms for several combined operations with DFAs and NFAs

module `transducers.py`: several classes and methods for transducers in standard form

module `codes.py`: language tests for a property (set of languages) specified by a transducer

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

f

- FAdo.cfg, [121](#)
- FAdo.codes, [135](#)
- FAdo.comboperations, [131](#)
- FAdo.common, [49](#)
- FAdo.conversions, [43](#)
- FAdo.fa, [5](#)
- FAdo.fio, [55](#)
- FAdo.fl, [113](#)
- FAdo.graphs, [155](#)
- FAdo.prax, [153](#)
- FAdo.reex, [57](#)
- FAdo.rndadfa, [127](#)
- FAdo.rndfap, [125](#)
- FAdo.sst, [147](#)
- FAdo.transducers, [105](#)

A

acyclicP() (*OFA method*), 38
 addEdge() (*DiGraph method*), 155
 addEdge() (*Graph method*), 156
 addEpsilonLoops() (*NFA method*), 28
 addEpsilonLoops() (*SFT method*), 106
 addEpsilonLoops() (*SSFA method*), 148
 addEpsilonLoops() (*SST method*), 149
 addFinal() (*FA method*), 21
 addInitial() (*NFA method*), 28
 addOutput() (*GFT method*), 105
 addSigma() (*FA method*), 21
 addState() (*AFA method*), 116
 addState() (*FA method*), 22
 addSuffix() (*ADFA method*), 113
 addToCode() (*IPTProp method*), 138
 addToSigma() (*SST method*), 149
 addTransition() (*DFA method*), 6
 addTransition() (*GFA method*), 45
 addTransition() (*GFT method*), 105
 addTransition() (*NFA method*), 28
 addTransition() (*NFAr method*), 36
 addTransition() (*OFA method*), 38
 addTransition() (*SFT method*), 106
 addTransition() (*SSFA method*), 148
 addTransition() (*SST method*), 149
 addTransitionProductQ() (*SFT method*), 106
 addTransitionProductQ() (*SST method*), 149
 addTransitionQ() (*NFA method*), 28
 addTransitionQ() (*SFT method*), 106
 addVertex() (*Graph method*), 157
 addWord() (*FL method*), 118
 addWords() (*FL method*), 118
 ADFA (*class in FAdo.fl*), 113
 ADFArnd (*class in FAdo.rndadfa*), 127
 aEquiv() (*DFA method*), 6
 AFA (*class in FAdo.fl*), 115
 AllWords (*class in FAdo.common*), 49
 alpha() (*ADFArnd method*), 127
 alpha0() (*ADFArnd method*), 127
 alphabet() (*PSPVanila method*), 147
 alphabeticLength() (*CAtom static method*), 57

alphabeticLength() (*Connective method*), 79
 alphabeticLength() (*Power method*), 82
 alphabeticLength() (*RegExp static method*), 84
 alphabeticLength() (*SConnective method*), 93
 alphabeticLength() (*SNot method*), 96
 alphabeticLength() (*SpecialConstant static method*), 99
 alphabeticLength() (*Unary method*), 101
 ANFA (*class in FAdo.fl*), 116
 assignLow() (*GFA method*), 45
 assignNum() (*GFA method*), 45
 autobisimulation() (*NFA method*), 29
 autobisimulation2() (*NFA method*), 29

B

behaviour() (*PSPVanila method*), 148
 beta() (*ADFArnd method*), 127
 beta0() (*ADFArnd method*), 128
 binomial() (*in module FAdo.common*), 51
 binomial() (*in module FAdo.rndadfa*), 129
 buildErrorCorrectPropF() (*in module FAdo.codes*), 142
 buildErrorCorrectPropS() (*in module FAdo.codes*), 142
 buildErrorDetectPropF() (*in module FAdo.codes*), 142
 buildErrorDetectPropS() (*in module FAdo.codes*), 142
 BuildFadoObject (*class in FAdo.fio*), 55
 buildHypercodeProperty() (*in module FAdo.codes*), 142
 buildIATPropF() (*in module FAdo.codes*), 142
 buildIATPropS() (*in module FAdo.codes*), 143
 buildInfixProperty() (*in module FAdo.codes*), 143
 buildIPTPropF() (*in module FAdo.codes*), 143
 buildIPTPropS() (*in module FAdo.codes*), 143
 buildOutfixProperty() (*in module FAdo.codes*), 143
 buildPrefixProperty() (*in module FAdo.codes*), 143
 BuildRegExp (*class in FAdo.reex*), 57
 BuildRPNRegExp (*class in FAdo.reex*), 57
 BuildRPNSRE (*class in FAdo.reex*), 57
 BuildSRE (*class in FAdo.reex*), 57

`buildSuffixProperty()` (in module *FAdo.codes*), 143
`buildTrajPropS()` (in module *FAdo.codes*), 143
`buildUDCodeProperty()` (in module *FAdo.codes*), 143

C

`catLF()` (DAG method), 81
`CAtom` (class in *FAdo.reex*), 57
`CConcat` (class in *FAdo.reex*), 63
`CConj` (class in *FAdo.reex*), 64
`CDisj` (class in *FAdo.reex*), 65
`CEmptySet` (class in *FAdo.reex*), 67
`CEpsilon` (class in *FAdo.reex*), 68
`CFGError`, 49
`CFGGenerator` (class in *FAdo.cfg*), 121
`CFGgrammarError`, 49
`CFGGrammar` (class in *FAdo.cfg*), 121
`CFGterminalError`, 49
`chomsky()` (CNF method), 122
`closeEpsilon()` (NFA method), 29
`CNF` (class in *FAdo.cfg*), 121
`codeOfTransducer()` (GFT method), 105
`CodeProperty` (class in *FAdo.codes*), 135
`CodesError`, 49
`CodingTheoryError`, 49
`compare()` (RegExp method), 84
`compareMinimalDFA()` (RegExp method), 85
`compat()` (DFA method), 6
`Compl` (class in *FAdo.reex*), 78
`complete()` (ADFA method), 113
`complete()` (DFA method), 6
`completeDelta()` (GFA method), 45
`completeMinimal()` (DFA method), 7
`completeP()` (DFA method), 7
`completeProduct()` (DFA method), 7
`composition()` (SFT method), 107
`computeFollowNames()` (NFA method), 29
`computeKernel()` (DFA method), 7
`concat()` (DFA method), 7
`concat()` (NFA method), 29
`concat()` (SFT method), 107
`concatI()` (DFA method), 7
`concatN()` (in module *FAdo.transducers*), 111
`concatWStar()` (in module *FAdo.comboperations*), 131
`conjunction()` (FA method), 22
`Connective` (class in *FAdo.reex*), 79
`constructCode()` (in module *FAdo.codes*), 144
`COption` (class in *FAdo.reex*), 69
`countadfa()` (in module *FAdo.rndadfa*), 129
`countAdfaBySources()` (ADFArnd method), 128
`countadfaL()` (in module *FAdo.rndadfa*), 129
`countafa()` (in module *FAdo.rndadfa*), 129
`countafaL()` (in module *FAdo.rndadfa*), 129
`countTransitions()` (FA method), 22
`countTransitions()` (NFA method), 29

`createInputAlteringSIDTrans()` (in module *FAdo.codes*), 144
`cross()` (SDisj static method), 95
`CShuffle` (class in *FAdo.reex*), 71
`CShuffleU` (class in *FAdo.reex*), 72
`CSigmaP` (class in *FAdo.reex*), 73
`CSigmaS` (class in *FAdo.reex*), 74
`CStar` (class in *FAdo.reex*), 75
`cutPoints()` (in module *FAdo.conversions*), 47
`CYKParserTable()` (in module *FAdo.cfg*), 122

D

`DAG` (class in *FAdo.reex*), 81
`deleteState()` (FA method), 22
`deleteState()` (GFA method), 46
`deleteState()` (SFT method), 107
`deleteStates()` (DFA method), 8
`deleteStates()` (NFA method), 30
`deleteStates()` (NFAr method), 36
`deleteStates()` (SFT method), 107
`delFinal()` (FA method), 22
`delFinals()` (FA method), 22
`delFromList()` (in module *FAdo.common*), 51
`Delta()` (DFA method), 5
`delTransition()` (DFA method), 8
`delTransition()` (NFA method), 29
`delTransition()` (NFAr method), 36
`delTransition()` (SFT method), 107
`dememoize()` (in module *FAdo.common*), 52
`derivative()` (CAtom method), 58
`derivative()` (CSigmaP method), 73
`derivative()` (CSigmaS method), 74
`derivative()` (SConcat method), 91
`derivative()` (SConj method), 92
`derivative()` (SDisj method), 95
`derivative()` (SNot method), 96
`derivative()` (SpecialConstant method), 99
`derivative()` (SStar method), 98
`deterministicP()` (DFA static method), 8
`deterministicP()` (NFA method), 30
`detSet()` (NFA method), 30
`DFA` (class in *FAdo.fa*), 5
`DFA2regexDijkstra()` (in module *FAdo.conversions*), 43
`dfaAuPoint()` (RegExp method), 85
`dfaBrzozowski()` (RegExp method), 85
`DFAdifferentSigma`, 49
`DFAEmptyDFA`, 49
`DFAEmptySigma`, 49
`DFAepsilonRedefinition`, 49
`DFAequivalent`, 49
`DFAerror`, 49
`DFAfileError`, 49
`DFAfound`, 49

DFAinputError, 49
 DFAMarkedError, 49
 dfaNaiveFollow() (*RegExp method*), 85
 DFAnoInitial, 49
 DFAnotComplete, 49
 DFAnotMinimal, 49
 DFAnotNFA, 49
 DFAstateUnknown, 49
 DFAstopped, 49
 DFAsymbolUnknown, 49
 DFASyncWords() (*in module FAdo.conversions*), 43
 DFASyntaticError, 49
 DFAtoADFA() (*in module FAdo.fl*), 117
 dfaYMG() (*RegExp method*), 85
 DFCA (*class in FAdo.fl*), 118
 DFS() (*GFA method*), 45
 dfs_visit() (*GFA method*), 46
 diff() (*FL method*), 118
 DiGraph (*class in FAdo.graphs*), 155
 DiGraphVm (*class in FAdo.graphs*), 156
 directRank() (*AFA method*), 116
 disj() (*FA method*), 22
 disjunction() (*FA method*), 22
 disjWStar() (*in module FAdo.comboperations*), 131
 display() (*Drawable method*), 50
 diss() (*ADFA method*), 113
 dissMin() (*ADFA method*), 114
 dist() (*DFA method*), 8
 distDerivative() (*SpecialConstant method*), 99
 distMin() (*DFA method*), 9
 distR() (*DFA method*), 9
 distRMin() (*DFA method*), 9
 distTS() (*DFA method*), 9
 dotDrawEdge() (*DiGraph static method*), 155
 dotDrawState() (*FA method*), 23
 dotDrawState() (*SemiDFA method*), 40
 dotDrawTransition() (*FA static method*), 23
 dotDrawTransition() (*OFA method*), 38
 dotDrawTransition() (*SemiDFA static method*), 40
 dotDrawVertex() (*DiGraph method*), 155
 dotFormat() (*DiGraph method*), 155
 dotFormat() (*Drawable method*), 50
 dotFormat() (*FA method*), 23
 dotFormat() (*Graph method*), 157
 dotFormat() (*NFA method*), 30
 dotFormat() (*SemiDFA method*), 41
 dotLabel() (*Drawable method*), 50
 Drawable (*class in FAdo.common*), 49
 dump() (*OFA method*), 39
 dup() (*ADFA method*), 114
 dup() (*DFA method*), 9
 dup() (*GFA method*), 46
 dup() (*NFA method*), 30
 dup() (*OFA method*), 39

dup() (*SFT method*), 107
 DuplicateName, 50

E

editDistanceW() (*in module FAdo.codes*), 144
 elim_unitary() (*CNF method*), 122
 elimEpsilon() (*NFA method*), 30
 elimEpsilon0() (*NFAr method*), 36
 eliminate() (*GFA method*), 46
 eliminateAll() (*GFA method*), 46
 eliminateDeadName() (*FA method*), 23
 eliminateEpsilonTransitions() (*NFA method*), 31
 eliminateState() (*GFA method*), 46
 eliminateStout() (*OFA method*), 39
 eliminateTSymbol() (*NFA method*), 31
 emptyDFA() (*in module FAdo.fa*), 41
 emptyP() (*OFA method*), 39
 emptyP() (*SFT method*), 107
 emptyP() (*SSFA method*), 148
 emptysetP() (*CEmptySet static method*), 67
 emptysetP() (*RegExp static method*), 86
 ensureDead() (*AFA method*), 116
 enum() (*EnumL method*), 20
 enumCrossSection() (*EnumL method*), 20
 EnumDFA (*class in FAdo.fa*), 19
 enumDFA() (*DFA method*), 9
 EnumL (*class in FAdo.fa*), 19
 EnumNFA (*class in FAdo.fa*), 20
 enumNFA() (*NFA method*), 31
 epsilonClosure() (*NFA method*), 31
 epsilonLength() (*CAtom static method*), 58
 epsilonLength() (*CEmptySet static method*), 67
 epsilonLength() (*CEpsilon static method*), 68
 epsilonLength() (*Compl method*), 78
 epsilonLength() (*Connective method*), 79
 epsilonLength() (*COption method*), 69
 epsilonLength() (*CShuffleU method*), 72
 epsilonLength() (*CStar method*), 75
 epsilonLength() (*Power method*), 83
 epsilonLength() (*RegExp static method*), 86
 epsilonLength() (*SConnective method*), 93
 epsilonLength() (*SNot method*), 96
 epsilonLength() (*SpecialConstant static method*), 99
 epsilonLength() (*Unary method*), 101
 epsilonOutP() (*SFT method*), 108
 epsilonOutP() (*SST method*), 150
 epsilonP() (*CEmptySet static method*), 67
 epsilonP() (*CEpsilon static method*), 68
 epsilonP() (*NFA method*), 31
 epsilonP() (*RegExp static method*), 86
 epsilonP() (*SFT method*), 108
 epsilonP() (*SSFA method*), 149
 epsilonP() (*SST method*), 150
 epsilonPaths() (*NFA method*), 31

`equal()` (*DFA method*), 9
`equivalentP()` (*FA method*), 24
`equivalentP()` (*in module FAdo.reex*), 103
`equivalentP()` (*RegExp method*), 86
`equivP()` (*RegExp method*), 86
`equivReduced()` (*NFA method*), 31
`ErrCorrectProp` (*class in FAdo.codes*), 135
`ErrDetectProp` (*in module FAdo.codes*), 136
`evalNumberOfStateCycles()` (*GFA method*), 46
`evalRank()` (*AFA method*), 116
`evalSymbol()` (*DFA method*), 9
`evalSymbol()` (*FA method*), 24
`evalSymbol()` (*GFA method*), 46
`evalSymbol()` (*NFA method*), 32
`evalSymbol()` (*OFA method*), 39
`evalSymbolI()` (*DFA method*), 10
`evalSymbolL()` (*DFA method*), 10
`evalSymbolLI()` (*DFA method*), 10
`evalWord()` (*DFA method*), 10
`evalWordP()` (*DFA method*), 11
`evalWordP()` (*NFA method*), 32
`evalWordP()` (*RegExp method*), 86
`evalWordP()` (*SFT method*), 108
`evalWordSlowP()` (*SFT method*), 108
`ewp()` (*CConcat method*), 63
`ewp()` (*CConj method*), 64
`ewp()` (*CDisj method*), 65
`ewp()` (*CEmptySet method*), 67
`ewp()` (*CEpsilon method*), 68
`ewp()` (*Compl method*), 78
`ewp()` (*COption method*), 69
`ewp()` (*CShuffle method*), 71
`ewp()` (*CShuffleU method*), 72
`ewp()` (*CSigmaP method*), 73
`ewp()` (*CSigmaS method*), 74
`ewp()` (*CStar method*), 75
`ewp()` (*RegExp static method*), 86
`ewp()` (*SConcat method*), 91
`ewp()` (*SConj method*), 92
`ewp()` (*SDisj method*), 95
`ewp()` (*SNot method*), 96
`ewp()` (*Unary method*), 101
`exponentialDensityP()` (*in module FAdo.codes*), 144

F

`f_dirichlet()` (*in module FAdo.prax*), 153
`f_laplace()` (*in module FAdo.prax*), 153
`FA` (*class in FAdo.fa*), 21
`FA2GFA()` (*in module FAdo.conversions*), 43
`FA2regexpCG()` (*in module FAdo.conversions*), 43
`FA2regexpCG_nn()` (*in module FAdo.conversions*), 43
`FA2regexpDynamicCycleHeuristic()` (*in module FAdo.conversions*), 43
`FA2regexpSE()` (*in module FAdo.conversions*), 44

`FA2regexpSE_nn()` (*in module FAdo.conversions*), 44
`FA2regexpSE0()` (*in module FAdo.conversions*), 44
`FA2regexpStaticCycleHeuristic()` (*in module FAdo.conversions*), 44
`FAallRegExps()` (*in module FAdo.conversions*), 45
`FAdo.cfg`
 module, 121
`FAdo.codes`
 module, 135
`FAdo.comboperations`
 module, 131
`FAdo.common`
 module, 49
`FAdo.conversions`
 module, 43
`FAdo.fa`
 module, 5
`FAdo.fio`
 module, 55
`FAdo.fl`
 module, 113
`FAdo.graphs`
 module, 155
`FAdo.prax`
 module, 153
`FAdo.reex`
 module, 57
`FAdo.rndadfa`
 module, 127
`FAdo.rndfap`
 module, 125
`FAdo.sst`
 module, 147
`FAdo.transducers`
 module, 105
`FAdoError`, 50
`FAdoGeneralError`, 50
`FAdoNotImplemented`, 50
`FAdoSyntacticError`, 50
`FAeliminateSingles()` (*in module FAdo.conversions*), 45
`FAException`, 50
`fillStack()` (*EnumDFA method*), 19
`fillStack()` (*EnumL method*), 20
`fillStack()` (*EnumNFA method*), 21
`filter()` (*FL method*), 118
`finalCompP()` (*DFA method*), 11
`finalCompP()` (*NFA method*), 32
`finalP()` (*FA method*), 24
`finalsP()` (*FA method*), 24
`first()` (*CAtom method*), 58
`first()` (*CConcat method*), 63
`first()` (*CDisj method*), 65
`first()` (*Compl method*), 78

[first\(\)](#) (*Connective method*), 79
[first\(\)](#) (*COption method*), 69
[first\(\)](#) (*CShuffle method*), 71
[first\(\)](#) (*CShuffleU method*), 72
[first\(\)](#) (*CStar method*), 75
[first\(\)](#) (*Power method*), 83
[first\(\)](#) (*RegExp method*), 86
[first\(\)](#) (*SConnective method*), 93
[first\(\)](#) (*SDisj method*), 95
[first\(\)](#) (*SNot method*), 96
[first\(\)](#) (*SpecialConstant static method*), 99
[first\(\)](#) (*Unary method*), 102
[first_l\(\)](#) (*CAtom method*), 58
[first_l\(\)](#) (*CConcat method*), 63
[first_l\(\)](#) (*CConj method*), 64
[first_l\(\)](#) (*CDisj method*), 65
[first_l\(\)](#) (*COption method*), 69
[first_l\(\)](#) (*CShuffle method*), 71
[first_l\(\)](#) (*CStar method*), 75
[fixedHierSubset\(\)](#) (*in module FAdo.codes*), 144
[FixedProp](#) (*class in FAdo.codes*), 136
[FL](#) (*class in FAdo.fl*), 118
[fnhException](#), 52
[follow_l\(\)](#) (*CAtom method*), 59
[follow_l\(\)](#) (*CConcat method*), 63
[follow_l\(\)](#) (*CConj method*), 64
[follow_l\(\)](#) (*CDisj method*), 65
[follow_l\(\)](#) (*COption method*), 69
[follow_l\(\)](#) (*CShuffle method*), 71
[follow_l\(\)](#) (*CStar method*), 76
[followFromPosition\(\)](#) (*NFA method*), 32
[followLists\(\)](#) (*CAtom method*), 58
[followLists\(\)](#) (*CConcat method*), 63
[followLists\(\)](#) (*CDisj method*), 65
[followLists\(\)](#) (*Compl method*), 78
[followLists\(\)](#) (*Connective method*), 80
[followLists\(\)](#) (*COption method*), 69
[followLists\(\)](#) (*CStar method*), 75
[followLists\(\)](#) (*Power method*), 83
[followLists\(\)](#) (*RegExp method*), 86
[followLists\(\)](#) (*SConnective method*), 93
[followLists\(\)](#) (*SDisj method*), 95
[followLists\(\)](#) (*SNot method*), 96
[followLists\(\)](#) (*SpecialConstant method*), 99
[followLists\(\)](#) (*Unary method*), 102
[followListsD\(\)](#) (*CAtom method*), 58
[followListsD\(\)](#) (*CConcat method*), 63
[followListsD\(\)](#) (*CDisj method*), 65
[followListsD\(\)](#) (*Compl method*), 78
[followListsD\(\)](#) (*Connective method*), 80
[followListsD\(\)](#) (*COption method*), 69
[followListsD\(\)](#) (*CShuffle method*), 71
[followListsD\(\)](#) (*CShuffleU method*), 72
[followListsD\(\)](#) (*CStar method*), 76

[followListsD\(\)](#) (*Power method*), 83
[followListsD\(\)](#) (*RegExp method*), 87
[followListsD\(\)](#) (*SConnective method*), 93
[followListsD\(\)](#) (*SNot method*), 97
[followListsD\(\)](#) (*SpecialConstant method*), 99
[followListsD\(\)](#) (*Unary method*), 102
[followListsStar\(\)](#) (*CAtom method*), 59
[followListsStar\(\)](#) (*COption method*), 69
[followListsStar\(\)](#) (*SDisj method*), 95
[followListsStar\(\)](#) (*SpecialConstant static method*), 100
[forceIterable\(\)](#) (*in module FAdo.common*), 52
[forceToDFA\(\)](#) (*ADFA method*), 114
[forceToDFCA\(\)](#) (*ADFA method*), 114
[functionalP\(\)](#) (*SFT method*), 108

G

[gamma\(\)](#) (*ADFArnd method*), 128
[generate\(\)](#) (*CFGGenerator method*), 121
[genFinalities\(\)](#) (*ICDFArgen method*), 125
[genRandomTrie\(\)](#) (*in module FAdo.fl*), 120
[genRndTrieBalanced\(\)](#) (*in module FAdo.fl*), 120
[genRndTriePrefix\(\)](#) (*in module FAdo.fl*), 120
[genRndTrieUnbalanced\(\)](#) (*in module FAdo.fl*), 120
[GenWordDis](#) (*class in FAdo.prax*), 153
[getAtomIdx\(\)](#) (*DAG method*), 81
[getIdx\(\)](#) (*DAG method*), 81
[getLeaves\(\)](#) (*AFA method*), 116
[GFA](#) (*class in FAdo.conversions*), 45
[GFT](#) (*class in FAdo.transducers*), 105
[Graph](#) (*class in FAdo.graphs*), 156
[GraphError](#), 50
[graphvizTranslate\(\)](#) (*in module FAdo.common*), 52
[gRules\(\)](#) (*in module FAdo.cfg*), 122

H

[half\(\)](#) (*NFA method*), 32
[hasStateIndexP\(\)](#) (*FA method*), 24
[hasTransitionP\(\)](#) (*NFA method*), 32
[hasTrapStateP\(\)](#) (*DFA method*), 11
[head\(\)](#) (*SConcat method*), 91
[head_rev\(\)](#) (*SConcat method*), 91
[HKeqP\(\)](#) (*DFA method*), 5
[HKeqP\(\)](#) (*NFA method*), 28
[homogeneousFinalityP\(\)](#) (*NFA method*), 32
[homogeneousP\(\)](#) (*in module FAdo.common*), 52
[homogenousP\(\)](#) (*NFA method*), 32
[homogenousP\(\)](#) (*NFAr method*), 37
[HypercodeProp](#) (*class in FAdo.codes*), 137
[hypercodeTransducer\(\)](#) (*in module FAdo.transducers*), 111
[hyperMinimal\(\)](#) (*DFA method*), 11

I

IATProp (class in FAdo.codes), 137
ICDFArgen (class in FAdo.rndfap), 125
ICDFArnd (class in FAdo.rndfap), 125
ICDFArndIncomplete (class in FAdo.rndfap), 125
iCompletePC (EnumL method), 20
IllegalBias, 50
images() (FA method), 24
inDegree() (DFA method), 11
indexList() (FA method), 24
infix() (DFA method), 11
InfixProp (class in FAdo.codes), 139
infixTransducer() (in module FAdo.transducers), 112
inIntersection() (PSPDiff method), 147
inIntersection() (PSPEqual method), 147
inIntersection() (PSPVanila method), 148
inIntersection() (SFT method), 108
inIntersection() (SST method), 150
inIntersectionSlow() (SFT method), 108
initialComp() (DFA method), 11
initialComp() (NFA method), 33
initialPC() (DFA method), 11
initialPC() (FA method), 24
initialSet() (DFA method), 12
initialSet() (FA method), 25
initStack() (EnumDFA method), 19
initStack() (EnumL method), 20
initStack() (EnumNFA method), 21
inputS() (FA method), 25
interLF() (DAG method), 81
intersection() (FL method), 119
interWStar() (in module FAdo.comboperations), 131
inverse() (DiGraph method), 156
inverse() (PSPVanila method), 148
inverse() (SFT method), 109
inverse() (SST method), 150
IPTProp (class in FAdo.codes), 138
isAInvariant() (PSPVanila method), 148
isLimitExceed() (in module FAdo.transducers), 112
isSubclass() (in module FAdo.codes), 145

J

joinStates() (DFA method), 12

L

last() (CAtom method), 59
last() (CConcat method), 63
last() (CDisj method), 65
last() (Compl method), 78
last() (Connective method), 80
last() (COption method), 69
last() (CShuffleU method), 72
last() (CStar method), 76

last() (Power method), 83
last() (RegExp method), 87
last() (SConnective method), 93
last() (SDisj method), 95
last() (SNot method), 97
last() (SpecialConstant method), 100
last() (Unary method), 102
last_l() (CAtom method), 59
last_l() (CConcat method), 63
last_l() (CConj method), 64
last_l() (CDisj method), 65
last_l() (COption method), 70
last_l() (CShuffle method), 71
last_l() (CStar method), 76
length (DFCA property), 118
lEquivNFA() (NFA method), 33
level() (ADFA method), 114
linearForm() (CAtom method), 59
linearForm() (CConcat method), 63
linearForm() (CConj method), 64
linearForm() (CDisj method), 65
linearForm() (Compl method), 78
linearForm() (Connective method), 80
linearForm() (COption method), 70
linearForm() (CShuffle method), 71
linearForm() (CShuffleU method), 72
linearForm() (CSigmaP method), 73
linearForm() (CSigmaS method), 74
linearForm() (CStar method), 76
linearForm() (Power method), 83
linearForm() (RegExp method), 87
linearForm() (SConcat method), 91
linearForm() (SConj method), 92
linearForm() (SConnective method), 93
linearForm() (SDisj method), 95
linearForm() (SNot method), 97
linearForm() (SpecialConstant method), 100
linearForm() (SStar method), 98
linearForm() (Unary method), 102
linearFormC() (CAtom method), 59
linearFormC() (CSigmaP method), 73
linearFormC() (CSigmaS method), 74
linearFormC() (SConcat method), 91
linearFormC() (SDisj method), 95
linearFormC() (SNot method), 97
linearPC() (CAtom method), 59
list2string() (in module FAdo.codes), 145
listOfTransitions() (GFT method), 105
long2base() (in module FAdo.codes), 145
lrEquivNFA() (NFA method), 33
lSet() (in module FAdo.common), 52

M

MADFA() (FL method), 118

- `make_prefix_free()` (*DFA method*), 12
- `makeCode()` (*IPTProp method*), 138
- `makeCode0()` (*IPTProp method*), 138
- `makenonterminals()` (*CFGGrammar method*), 121
- `makePNG()` (*Drawable method*), 50
- `makeReversible()` (*DFA method*), 12
- `maketerminals()` (*CFGGrammar method*), 121
- `mark()` (*CAtom method*), 60
- `mark()` (*CConcat method*), 63
- `mark()` (*CConj method*), 64
- `mark()` (*CDisj method*), 66
- `mark()` (*Compl method*), 78
- `mark()` (*Connective method*), 80
- `mark()` (*CShuffle method*), 71
- `mark()` (*CShuffleU method*), 72
- `mark()` (*Power method*), 83
- `mark()` (*RegExp method*), 87
- `mark()` (*SConnective method*), 94
- `mark()` (*SNot method*), 97
- `mark()` (*SpecialConstant method*), 100
- `mark()` (*Unary method*), 102
- `marked()` (*RegExp method*), 87
- `markNonEquivalent()` (*DFA method*), 12
- `markVertex()` (*DiGraphVm method*), 156
- `MAtom` (*class in FAdo.reex*), 82
- `maximalP()` (*CodeProperty method*), 135
- `maximalP()` (*FixedProp method*), 136
- `maximalP()` (*IPTProp method*), 139
- `maximalP()` (*UDCodeProp method*), 142
- `measure()` (*CAtom static method*), 60
- `measure()` (*CEmptySet static method*), 67
- `measure()` (*CEpsilon static method*), 68
- `memoize()` (*in module FAdo.common*), 52
- `memoized` (*class in FAdo.common*), 52
- `mergeInitial()` (*ANFA method*), 117
- `mergeLeaves()` (*ANFA method*), 117
- `mergeStates()` (*ANFA method*), 117
- `mergeStates()` (*DFA method*), 12
- `mergeStates()` (*NFAr method*), 37
- `mergeStatesSet()` (*NFAr method*), 37
- `minDFCA()` (*ADFA method*), 114
- `minI()` (*in module FAdo.prax*), 153
- `minimal()` (*ADFA method*), 114
- `minimal()` (*DFA method*), 12
- `minimal()` (*NFA method*), 33
- `minimalBrzozowski()` (*OFA method*), 39
- `minimalBrzozowskiP()` (*OFA method*), 39
- `minimalDFA()` (*NFA method*), 33
- `minimalHopcroft()` (*DFA method*), 13
- `minimalHopcroftP()` (*DFA method*), 13
- `minimalIncremental()` (*DFA method*), 13
- `minimalIncrementalP()` (*DFA method*), 13
- `minimalMoore()` (*DFA method*), 13
- `minimalMooreSq()` (*DFA method*), 13
- `minimalMooreSqP()` (*DFA method*), 14
- `minimalNCompleteP()` (*DFA method*), 14
- `minimalNotEquivP()` (*DFA method*), 14
- `minimalP()` (*ADFA method*), 114
- `minimalP()` (*DFA method*), 14
- `minimalWatson()` (*DFA method*), 14
- `minimalWatsonP()` (*DFA method*), 14
- `minReversible()` (*ADFA method*), 114
- `minWord()` (*EnumL method*), 20
- `minWordT()` (*EnumDFA method*), 19
- `minWordT()` (*EnumL method*), 20
- `minWordT()` (*EnumNFA method*), 21
- `module`
 - `FAdo.cfg`, 121
 - `FAdo.codes`, 135
 - `FAdo.comboperations`, 131
 - `FAdo.common`, 49
 - `FAdo.conversions`, 43
 - `FAdo.fa`, 5
 - `FAdo.fio`, 55
 - `FAdo.fl`, 113
 - `FAdo.graphs`, 155
 - `FAdo.prax`, 153
 - `FAdo.reex`, 57
 - `FAdo.rndadfa`, 127
 - `FAdo.rndfap`, 125
 - `FAdo.sst`, 147
 - `FAdo.transducers`, 105
- `moveFinal()` (*ANFA method*), 117
- `multiLineAutomaton()` (*FL method*), 119
- `MyhillNerodePartition()` (*DFA method*), 6

N

- `nextWord()` (*EnumDFA method*), 19
- `nextWord()` (*EnumL method*), 20
- `nextWord()` (*EnumNFA method*), 21
- `NFA` (*class in FAdo.fa*), 27
- `NFA()` (*DAG method*), 81
- `NFA2SSFA()` (*in module FAdo.sst*), 147
- `NFAEmpty`, 50
- `NFAerror`, 50
- `nfaFollow()` (*RegExp method*), 87
- `nfaFollowEpsilon()` (*RegExp method*), 87
- `nfaGlushkov()` (*RegExp method*), 88
- `nfaLoc()` (*RegExp method*), 88
- `nfaNaiveFollow()` (*RegExp method*), 88
- `nfaPD()` (*CEmptySet method*), 67
- `nfaPD()` (*CSigmaP method*), 73
- `nfaPD()` (*CSigmaS method*), 75
- `nfaPD()` (*RegExp method*), 88
- `nfaPD()` (*SConnective method*), 94
- `nfaPD()` (*SNot method*), 97
- `nfaPD()` (*SStar method*), 98
- `nfaPDDAG()` (*RegExp method*), 88

nfaPDNaive() (*RegExp method*), 88
nfaPDO() (*RegExp method*), 89
nfaPosition() (*RegExp method*), 89
nfaPre() (*RegExp method*), 89
nfaPSNF() (*RegExp method*), 89
NFAR (*class in FAdo.fa*), 36
nfaThompson() (*CAtom method*), 60
nfaThompson() (*CDisj method*), 66
nfaThompson() (*CEpsilon method*), 68
nfaThompson() (*COption method*), 70
nfaThompson() (*CStar method*), 76
NFT (*class in FAdo.transducers*), 106
NImplemented, 51
noBlankNames() (*FA method*), 25
notEmptyW() (*SFT method*), 109
notEmptyW() (*SST method*), 150
nonFunctionalW() (*SFT method*), 109
NonPlanar, 51
normalize() (*GFA method*), 46
notAcyclic, 52
notEmptyW() (*RegExp method*), 89
notequal() (*DFA method*), 14
notMaximalW() (*CodeProperty method*), 135
notMaximalW() (*ErrCorrectProp method*), 136
notMaximalW() (*FixedProp method*), 136
notMaximalW() (*IPTProp method*), 139
notMaxStatW() (*IPTProp method*), 139
notSatisfiesW() (*CodeProperty method*), 135
notSatisfiesW() (*ErrCorrectProp method*), 136
notSatisfiesW() (*FixedProp method*), 137
notSatisfiesW() (*IATProp method*), 137
notSatisfiesW() (*IPTProp method*), 139
notSatisfiesW() (*UDCodeProp method*), 142
NotSP, 51
notUniversalStatW() (*in module FAdo.codes*), 145
NULLABLE() (*CFGGrammar method*), 121

O

OFA (*class in FAdo.fa*), 38
ordered() (*AFA method*), 116
orderedStrConnComponents() (*DFA method*), 14
OutfixProp (*class in FAdo.codes*), 140
outfixTransducer() (*in module FAdo.transducers*), 112
outIntersection() (*SFT method*), 109
outIntersection() (*SST method*), 150
outIntersectionDerived() (*SFT method*), 109
outputS() (*SFT method*), 109
overlapFreeP() (*in module FAdo.common*), 52

P

pairGraph() (*DFA method*), 14
partialDerivatives() (*CAtom method*), 60
partialDerivatives() (*CEmptySet method*), 67

partialDerivatives() (*CEpsilon method*), 68
partialDerivatives() (*CSigmaP method*), 73
partialDerivatives() (*CSigmaS method*), 75
partialDerivatives() (*SConcat method*), 91
partialDerivatives() (*SConj method*), 92
partialDerivatives() (*SDisj method*), 95
partialDerivatives() (*SNot method*), 97
partialDerivatives() (*SStar method*), 98
partialDerivativesC() (*CAtom method*), 60
partialDerivativesC() (*CEmptySet method*), 67
partialDerivativesC() (*CEpsilon method*), 68
partialDerivativesC() (*CSigmaP method*), 74
partialDerivativesC() (*CSigmaS method*), 75
partialDerivativesC() (*SConcat method*), 91
partialDerivativesC() (*SConj method*), 92
partialDerivativesC() (*SDisj method*), 95
partialDerivativesC() (*SNot method*), 97
partialDerivativesC() (*SpecialConstant method*), 100
partialDerivativesC() (*SStar method*), 98
PD() (*CAtom method*), 57
PDAerror, 51
PDAsymbolUnknown, 51
PEGError, 51
pickFrom() (*in module FAdo.codes*), 145
plus() (*FA method*), 25
plusLF() (*DAG static method*), 81
Position (*class in FAdo.reex*), 82
possibleToReverse() (*ADFA method*), 115
possibleToReverse() (*DFA method*), 15
Power (*class in FAdo.reex*), 82
powerset() (*in module FAdo.reex*), 103
prax_parameters() (*in module FAdo.prax*), 154
prax_univ_nfa() (*in module FAdo.prax*), 154
pref() (*DFA method*), 15
prefix_free_p() (*DFA method*), 15
PrefixProp (*class in FAdo.codes*), 140
prefixTransducer() (*in module FAdo.transducers*), 112
print_data() (*DFA method*), 15
product() (*DFA method*), 15
product() (*NFA method*), 33
productInput() (*SFT method*), 109
productInput() (*SST method*), 150
productInputSlow() (*SFT method*), 109
productSlow() (*DFA method*), 15
PropertyNotSatisfied, 51
PSP (*class in FAdo.sst*), 147
PSPDiff (*class in FAdo.sst*), 147
PSPEqual (*class in FAdo.sst*), 147
PSPVanila (*class in FAdo.sst*), 147

R

random() (*in module FAdo.prax*), 154

[readFromFile\(\)](#) (in module *FAdo.fio*), 55
[readOneFromFile\(\)](#) (in module *FAdo.fio*), 56
[readOneFromString\(\)](#) (in module *FAdo.fio*), 56
[reduced\(\)](#) (*CAtom* method), 61
[RegExp](#) (class in *FAdo.reex*), 84
[regexpInvalid](#), 53
[regexpInvalidMethod](#), 53
[regexpInvalidSymbols](#), 53
[RegularExpression](#) (class in *FAdo.reex*), 90
[renameState\(\)](#) (*FA* method), 25
[renameStates\(\)](#) (*FA* method), 25
[renameStatesFromPosition\(\)](#) (*NFA* method), 34
[reorder\(\)](#) (*DFA* method), 15
[reorder\(\)](#) (*GFA* method), 46
[reorder\(\)](#) (*NFA* method), 34
[rEquivNFA\(\)](#) (*NFA* method), 34
[REStringRGenerator](#) (class in *FAdo.cfg*), 122
[reversal\(\)](#) (*CAtom* method), 61
[reversal\(\)](#) (*CConcat* method), 64
[reversal\(\)](#) (*CDisj* method), 66
[reversal\(\)](#) (*FA* method), 26
[reversal\(\)](#) (*NFA* method), 34
[reversal\(\)](#) (*Power* method), 83
[reversal\(\)](#) (*SFT* method), 110
[reversal\(\)](#) (*SpecialConstant* method), 100
[reversal\(\)](#) (*SST* method), 151
[reversal\(\)](#) (*Unary* method), 102
[reverseTransitions\(\)](#) (*DFA* method), 16
[reverseTransitions\(\)](#) (*NFA* method), 34
[reversibleP\(\)](#) (*DFA* method), 16
[rndAdfa\(\)](#) (*ADFArnd* method), 128
[rndNumberSecondSources\(\)](#) (*ADFArnd* method), 128
[rndTransitionsFromSources\(\)](#) (*ADFArnd* method), 129
[RndWGen](#) (class in *FAdo.fl*), 119
[rpn\(\)](#) (*CAtom* method), 61
[rpn\(\)](#) (*CConcat* method), 64
[rpn\(\)](#) (*CConj* method), 64
[rpn\(\)](#) (*CDisj* method), 66
[rpn\(\)](#) (*CEmptySet* method), 67
[rpn\(\)](#) (*CEpsilon* method), 68
[rpn\(\)](#) (*Compl* method), 78
[rpn\(\)](#) (*Connective* method), 80
[rpn\(\)](#) (*COption* method), 70
[rpn\(\)](#) (*CShuffle* method), 71
[rpn\(\)](#) (*CShuffleU* method), 72
[rpn\(\)](#) (*CSigmaP* method), 74
[rpn\(\)](#) (*CSigmaS* method), 75
[rpn\(\)](#) (*CStar* method), 77
[rpn\(\)](#) (*Power* method), 83
[rpn\(\)](#) (*RegExp* method), 89
[rpn\(\)](#) (*SConnective* method), 94
[rpn\(\)](#) (*SNot* method), 97
[rpn\(\)](#) (*SpecialConstant* method), 100

[rpn\(\)](#) (*Unary* method), 102
[rpn2regexp\(\)](#) (in module *FAdo.reex*), 103
[runOnNFA\(\)](#) (*SFT* method), 110
[runOnWord\(\)](#) (*SFT* method), 110

S

[same_nullability\(\)](#) (*FA* method), 26
[satisfiesP\(\)](#) (*CodeProperty* method), 135
[satisfiesP\(\)](#) (*ErrCorrectProp* method), 136
[satisfiesP\(\)](#) (*FixedProp* method), 137
[satisfiesP\(\)](#) (*IPTProp* method), 139
[satisfiesP\(\)](#) (*UDCodeProp* method), 142
[satisfiesPrefixP\(\)](#) (*PrefixProp* method), 140
[saveToFile\(\)](#) (in module *FAdo.fio*), 56
[saveToJson\(\)](#) (in module *FAdo.fio*), 56
[saveToString\(\)](#) (in module *FAdo.fa*), 41
[saveToString\(\)](#) (in module *FAdo.fio*), 56
[SConcat](#) (class in *FAdo.reex*), 91
[sConcat\(\)](#) (in module *FAdo.common*), 53
[SConj](#) (class in *FAdo.reex*), 92
[SConnective](#) (class in *FAdo.reex*), 93
[SDisj](#) (class in *FAdo.reex*), 94
[SemiDFA](#) (class in *FAdo.fa*), 40
[setDeadState\(\)](#) (*AFA* method), 116
[setFinal\(\)](#) (*FA* method), 26
[setInitial\(\)](#) (*FA* method), 26
[setInitial\(\)](#) (*NFA* method), 34
[setInitial\(\)](#) (*SFT* method), 110
[setOfSymbols\(\)](#) (*CAtom* method), 61
[setOfSymbols\(\)](#) (*Connective* method), 80
[setOfSymbols\(\)](#) (*COption* method), 70
[setOfSymbols\(\)](#) (*Position* method), 82
[setOfSymbols\(\)](#) (*Power* method), 83
[setOfSymbols\(\)](#) (*RegExp* static method), 89
[setOfSymbols\(\)](#) (*SConnective* method), 94
[setOfSymbols\(\)](#) (*SNot* method), 97
[setOfSymbols\(\)](#) (*SpecialConstant* static method), 100
[setOfSymbols\(\)](#) (*Unary* method), 102
[setOutput\(\)](#) (*Transducer* method), 111
[setSigma\(\)](#) (*FA* method), 26
[setSigma\(\)](#) (*FL* method), 119
[setSigma\(\)](#) (*RegExp* method), 90
[SetSpec](#) (class in *FAdo.sst*), 151
[SFT](#) (class in *FAdo.transducers*), 106
[show\(\)](#) (in module *FAdo.fio*), 56
[shuffle\(\)](#) (*DFA* method), 16
[shuffle\(\)](#) (*NFA* method), 34
[shuffleLF\(\)](#) (*DAG* method), 81
[sigmaInitialSegment\(\)](#) (in module *FAdo.fl*), 120
[sigmaStarDFA\(\)](#) (in module *FAdo.fa*), 41
[simDiff\(\)](#) (*DFA* method), 16
[smallAlphabet\(\)](#) (in module *FAdo.cfg*), 123
[sMonoid\(\)](#) (*DFA* method), 16
[snf\(\)](#) (*CAtom* method), 61

snf() (*CConcat method*), 64
snf() (*CConj method*), 64
snf() (*CDisj method*), 66
snf() (*CEmptySet method*), 68
snf() (*CEpsilon method*), 68
snf() (*Compl method*), 79
snf() (*Connective method*), 80
snf() (*COption method*), 71
snf() (*CShuffle method*), 72
snf() (*CShuffleU method*), 72
snf() (*CStar method*), 77
snf() (*Power method*), 84
snf() (*RegExp method*), 90
snf() (*SConnective method*), 94
snf() (*SNot method*), 97
snf() (*SpecialConstant method*), 100
snf() (*Unary method*), 102
SNot (*class in FAdo.reex*), 96
sop() (*DFA method*), 16
SP2regexp() (*in module FAdo.conversions*), 47
SpecialConstant (*class in FAdo.reex*), 99
SPLabel (*class in FAdo.common*), 51
square() (*SFT method*), 110
square_fv() (*SFT method*), 110
SSAnyOf (*class in FAdo.sst*), 148
SSBadTransition, 51
SSConditionalNoneOf() (*in module FAdo.sst*), 148
sSemigroup() (*DFA method*), 16
SSEmpty (*class in FAdo.sst*), 148
SSEpsilon (*class in FAdo.sst*), 148
SSError, 51
SSFA (*class in FAdo.sst*), 148
SSMissAlphabet, 51
SSNoneOf (*class in FAdo.sst*), 149
SSOneOf (*class in FAdo.sst*), 149
SST (*class in FAdo.sst*), 149
SStar (*class in FAdo.reex*), 98
star() (*DFA method*), 16
star() (*NFA method*), 34
star() (*SFT method*), 110
starConcat() (*in module FAdo.comboperations*), 132
starDisj() (*in module FAdo.comboperations*), 132
starHeight() (*CAtom static method*), 61
starHeight() (*Compl method*), 79
starHeight() (*Connective method*), 80
starHeight() (*COption method*), 71
starHeight() (*CShuffleU method*), 72
starHeight() (*CStar method*), 77
starHeight() (*Power method*), 84
starHeight() (*RegExp static method*), 90
starHeight() (*SConnective method*), 94
starHeight() (*SNot method*), 97
starHeight() (*SpecialConstant static method*), 100
starHeight() (*Unary method*), 102
starI() (*DFA method*), 17
starInter() (*in module FAdo.comboperations*), 132
starInter0() (*in module FAdo.comboperations*), 132
starWConcat() (*in module FAdo.comboperations*), 133
stateAlphabet() (*FA method*), 26
stateChildren() (*DFA method*), 17
stateChildren() (*GFA method*), 46
stateChildren() (*NFA method*), 34
stateChildren() (*OFA method*), 39
stateIndex() (*FA method*), 26
stateName() (*FA method*), 27
statePairEquiv() (*ADFA method*), 115
statePP() (*in module FAdo.fa*), 41
str2regexp() (*in module FAdo.reex*), 103
str2sre() (*in module FAdo.reex*), 104
stringLength() (*CAtom method*), 62
stringToADFA() (*in module FAdo.fl*), 120
stringToDFA() (*in module FAdo.fa*), 42
stronglyConnectedComponents() (*DFA method*), 17
stronglyConnectedComponents() (*NFA method*), 35
subword() (*DFA method*), 17
subword() (*NFA method*), 35
succintTransitions() (*DFA method*), 17
succintTransitions() (*FA method*), 27
succintTransitions() (*GFA method*), 47
succintTransitions() (*NFA method*), 35
succintTransitions() (*OFA method*), 39
succintTransitions() (*Transducer method*), 111
suff() (*DFA method*), 17
suffixClosedP() (*FL method*), 119
suffixes() (*in module FAdo.common*), 53
SuffixProp (*class in FAdo.codes*), 141
suffixTransducer() (*in module FAdo.transducers*), 112
support() (*CAtom method*), 62
support() (*CConcat method*), 64
support() (*CDisj method*), 66
support() (*Compl method*), 79
support() (*COption method*), 71
support() (*CSigmaP method*), 74
support() (*CSigmaS method*), 75
support() (*CStar method*), 77
support() (*Power method*), 84
support() (*RegExp method*), 90
support() (*SConcat method*), 91
support() (*SConj method*), 92
support() (*SConnective method*), 94
support() (*SDisj method*), 96
support() (*SNot method*), 98
support() (*SpecialConstant method*), 100
support() (*SStar method*), 99
supportlast() (*CAtom method*), 62
supportlast() (*SpecialConstant method*), 100
surj() (*in module FAdo.rndadfa*), 129

[symmAndRefl\(\)](#) (in module *FAdo.codes*), 145
[syncPower\(\)](#) (DFA method), 17
[syntacticLength\(\)](#) (CAtom static method), 62
[syntacticLength\(\)](#) (SConnective method), 94
[syntacticLength\(\)](#) (SNot method), 98

T

[tail\(\)](#) (SConcat method), 91
[tail_rev\(\)](#) (SConcat method), 92
[tailForm\(\)](#) (CAtom method), 62
[tailForm\(\)](#) (CConcat method), 64
[tailForm\(\)](#) (CConj method), 64
[tailForm\(\)](#) (CDisj method), 67
[tailForm\(\)](#) (Compl method), 79
[tailForm\(\)](#) (COption method), 71
[tailForm\(\)](#) (CShuffle method), 72
[tailForm\(\)](#) (CShuffleU method), 73
[tailForm\(\)](#) (CStar method), 77
[tailForm\(\)](#) (Power method), 84
[tailForm\(\)](#) (RegExp method), 90
[tailForm\(\)](#) (SConcat method), 92
[tailForm\(\)](#) (SConj method), 92
[tailForm\(\)](#) (SConnective method), 94
[tailForm\(\)](#) (SDisj method), 96
[tailForm\(\)](#) (SNot method), 98
[tailForm\(\)](#) (SpecialConstant method), 101
[TFAAccept](#), 51
[TFARreject](#), 51
[TFARrejectBlocked](#), 51
[TFARrejectLoop](#), 51
[TFARrejectNonFinal](#), 51
[TFASignal](#), 51
[to_s\(\)](#) (in module *FAdo.reex*), 104
[toADFA\(\)](#) (DFA method), 17
[toANFA\(\)](#) (ADFA method), 115
[toDFA\(\)](#) (DFA method), 18
[toDFA\(\)](#) (FL method), 119
[toDFA\(\)](#) (NFA method), 35
[toDFA\(\)](#) (RegExp method), 90
[toInNFA\(\)](#) (SFT method), 110
[toInNFA\(\)](#) (SST method), 151
[toInSSFA\(\)](#) (SST method), 151
[toJson\(\)](#) (in module *FAdo.fio*), 56
[toNFA\(\)](#) (ADFA method), 115
[toNFA\(\)](#) (DFA method), 18
[toNFA\(\)](#) (FL method), 119
[toNFA\(\)](#) (NFA method), 35
[toNFA\(\)](#) (NFAR method), 37
[toNFA\(\)](#) (RegExp method), 90
[toNFA\(\)](#) (SSFA method), 149
[toNFAR\(\)](#) (NFA method), 35
[toNFT\(\)](#) (SFT method), 110
[toOutNFA\(\)](#) (SFT method), 111
[toOutNFA\(\)](#) (SST method), 151

[toOutSSFA\(\)](#) (SST method), 151
[topoSort\(\)](#) (OFA method), 40
[toSFT\(\)](#) (GFT method), 105
[toSFT\(\)](#) (SFT method), 111
[toSFT\(\)](#) (SST method), 151
[toXSSFA\(\)](#) (SST method), 151
[TrajProp](#) (class in *FAdo.codes*), 141
[trajToTransducer\(\)](#) (TrajProp static method), 141
[Transducer](#) (class in *FAdo.transducers*), 111
[transitions\(\)](#) (DFA method), 18
[transitionsA\(\)](#) (DFA method), 18
[treeLength\(\)](#) (CAtom static method), 62
[treeLength\(\)](#) (Compl method), 79
[treeLength\(\)](#) (Connective method), 80
[treeLength\(\)](#) (Power method), 84
[treeLength\(\)](#) (RegExp static method), 90
[treeLength\(\)](#) (SConnective method), 94
[treeLength\(\)](#) (SNot method), 98
[treeLength\(\)](#) (SpecialConstant static method), 101
[treeLength\(\)](#) (Unary method), 103
[TSError](#), 51
[trieFA\(\)](#) (FL method), 119
[trim\(\)](#) (ADFA method), 115
[trim\(\)](#) (OFA method), 40
[trim\(\)](#) (SFT method), 111
[trimP\(\)](#) (OFA method), 40
[TstError](#), 51
[twDict](#) (class in *FAdo.common*), 53
[TypeError](#), 51

U

[UDCodeProp](#) (class in *FAdo.codes*), 141
[Unary](#) (class in *FAdo.reex*), 101
[union\(\)](#) (FA method), 27
[union\(\)](#) (FL method), 119
[union\(\)](#) (SFT method), 111
[unionOfIDs\(\)](#) (in module *FAdo.codes*), 146
[unionSigma\(\)](#) (RegExp method), 90
[unique\(\)](#) (in module *FAdo.common*), 53
[uniqueRepr\(\)](#) (DFA method), 18
[uniqueRepr\(\)](#) (NFA method), 35
[unive_index\(\)](#) (in module *FAdo.prax*), 154
[universalP\(\)](#) (DFA method), 18
[unlinkSoleIncoming\(\)](#) (NFAR method), 37
[unlinkSoleOutgoing\(\)](#) (NFAR method), 38
[unmark\(\)](#) (CConcat method), 64
[unmark\(\)](#) (CDisj method), 67
[unmark\(\)](#) (Compl method), 79
[unmark\(\)](#) (CShuffleU method), 73
[unmark\(\)](#) (DFA method), 18
[unmark\(\)](#) (MAtom method), 82
[unmark\(\)](#) (SpecialConstant method), 101
[unmark\(\)](#) (Unary method), 103
[unmarked\(\)](#) (CAtom method), 62

`unmarked()` (*Position method*), 82
`unmarked()` (*SpecialConstant method*), 101
`usefulStates()` (*DFA method*), 18
`usefulStates()` (*NFA method*), 35
`uSet()` (*in module FAdo.common*), 53

V

`VersoError`, 51
`vertexIndex()` (*Graph method*), 157
`VertexNotInGraph`, 51

W

`weight()` (*GFA method*), 47
`weightWithCycles()` (*GFA method*), 47
`witness()` (*DFA method*), 18
`witness()` (*NFA method*), 35
`witness()` (*SSFA method*), 149
`witnessDiff()` (*DFA method*), 18
`Word` (*class in FAdo.common*), 51
`wordDerivative()` (*RegExp method*), 90
`wordDerivative()` (*SpecialConstant method*), 101
`wordGenerator()` (*ADFA method*), 115
`wordImage()` (*NFA method*), 36
`words()` (*FA method*), 27

Z

`ZERO`, 111