
FAdo Documentation

Release 2.0

Rogério Reis & Nelma Moreira

Jan 31, 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	Module: Finite Automata (fa)	5
2.1	Classes	5
2.2	Functions	7
3	Module: Conversion of Finite Automata: (FAdo.conversions)	9
3.1	Classes	9
3.2	Functions	9
4	Module: Common Definitions (common)	13
4.1	Classes	13
5	Module: FAdo IO Functions (fio)	15
5.1	Class BuildFadoObject (Semantics of the FAdo grammars' objects)	15
5.2	Functions	15
5.3	Constants	17
6	Module: Regular Expressions (reex)	19
6.1	Classes	19
6.2	Functions	24
7	Module: Transducers (transducers)	27
7.1	Classes	27
7.2	Functions	28
8	Module: Finite Languages (fl)	31
8.1	Classes	31
8.2	Functions	32
9	Module: graphs (graph creation and manipulation)	35
10	Module: Context Free Grammars Manipulation (cfg)	37
10.1	Classes	37
10.2	Functions	38
10.3	Constants	38

11 Module: Random DFA Generator (rndfap)	39
11.1 Classes	39
12 Module: Random ADFA Generator (rndadfa)	41
12.1 Classes	41
13 Module: Combo Operations (comboperations)	43
13.1 Functions	43
14 Module: Codes (codes)	47
14.1 Classes	47
14.2 Functions	49
15 A small tutorial for FAdo	53
16 More classes and modules	59
17 Indices and tables	61
Python Module Index	63
Index	65

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

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 (regex) 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

MODULE: FINITE AUTOMATA (FA)

Finite automata manipulation.

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

2.1 Classes

class FAdo.fa.FA

Base class for Finite Automata.

States

set of states.

Type list

Sigma

alphabet set.

Type set

Initial

the initial state index.

Type int

Final

set of final states indexes.

Type set

delta

the transition function.

Type dict

Note: This is just an abstract class. **Not to be used directly!!**

class FAdo.fa.SemiDFA

Class of automata without initial or final states

States

list of states.

Type list

delta

transition function.

Type `dict`

Sigma

alphabet.

Type `set`

class `FAdo.fa.OFA`

Base class for one-way automata .. inheritance-diagram:: OFA

class `FAdo.fa.DFA`

Class for Deterministic Finite Automata.



class `FAdo.fa.NFA`

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



Return type `object`

class `FAdo.fa.NFAr`

Class for Non-deterministic Finite Automata with reverse delta function added by construction.



Note: Includes efficient methods for merging states.

Return type `object`

class `FAdo.fa.SSemiGroup`

Class support for the Syntactic SemiGroup.

Variables

- **elements** – list of tuples representing the transformations

- **words** – a list of pairs (index of the prefix transformation, index of the suffix char)
- **gen** – a list of the max index of each generation
- **Sigma** – set of symbols

class FAdo.fa.**EnumL**(*aut, store=False*)

Class for enumerate FA languages

Variables

- **aut** (**FA**) – 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

New in version 0.9.8.

See also:

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>

2.2 Functions

FAdo.fa.**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**) – the FA
- **sep** (*str*) – separation between *lines*

Returns the representation

Return type *str*

FAdo.fa.**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**

Changed in version 0.9.8: symbols are converted to str

MODULE: CONVERSION OF FINITE AUTOMATA: (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* <http://fado.dcc.fc.up.pt>.

3.1 Classes

class FAdo.conversions.GFA

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



3.2 Functions

FAdo.conversions.FA2GFA(*aut*)

Creates a GFA equivalent to NFA

Parameters *aut* (OFA) – the automaton

Returns a GFA deep copy

Return type GFA

FAdo.conversions.FAallRegExps(*aut*)

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

Parameters *aut* (OFA) – the automaton

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

FAdo.conversions.cutPoints(*aut*)

Set of FA's cut points

Parameters *aut* (OFA) – the automaton

Returns set of states

Return type set of int

`FAdo.conversions.FA2regexpSE(aut)`

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

Parameters *aut* (OFA) – the automaton

Returns the equivalent regular expression

Return type *reex.regexp*

`FAdo.conversions.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) – the automaton

Returns equivalent regular expression

Return type *reex.regexp*

Raises **NotSP** – 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

`FAdo.conversions.FAeliminateSingles(aut)`

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

Parameters *aut* (OFA) – the automaton

Returns GFA after eliminating states

Return type *GFA*

`FAdo.conversions.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) – the automaton

Returns the equivalent regular expression

Return type *reex.regexp*

`FAdo.conversions.FA2regexpCG_nn(aut: FAdo.fa.OFA)`

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) – the automaton

Returns the equivalent regular expression

Return type *reex.regexp*

`FAdo.conversions.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*) – the automaton
- **order** (*list*) – state elimination sequence

Returns the equivalent regular expression

Return type *reex.regexp*

`FAdo.conversions.FA2regexDynamicCycleHeuristic(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*) – the automaton

Returns an equivalent regular expression

Return type *reex.regexp*

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

`FAdo.conversions.FA2regexStaticCycleHeuristic(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*) – the automaton

Returns a equivalent regular expression

Return type *reex.regexp*

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

`FAdo.conversions.FA2regexSE_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*) – the automaton
- **order** (*list*) – state elimination sequence

Returns the equivalent regular expression

Return type *reex.regexp*

`FAdo.conversions.DFA2regexDijkstra(aut) → FAdo.reex.regexp`

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

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

Returns a regexp equivalent to the current DFA

Return type *reex.regexp*

MODULE: COMMON DEFINITIONS (COMMON)

Common definitions for FAdo files

4.1 Classes

class FAdo.common.Word(*data=None, it=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

class FAdo.common.Drawable

Any FAdo object that is drawable

MODULE: FADO IO FUNCTIONS (FI0)

In/Out.

FAdo IO.

5.1 Class BuildFadoObject (Semantics of the FAdo grammars' objects)

```
class FAdo.fio.BuildFadoObject(visit_tokens=True)
    Semantics of the FAdo grammars' objects
```

5.2 Functions

```
FAdo.fio.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 a 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.

FAdo.fio.readOneFromFile(fileName)

Read the first of the FAdo objects from File

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

Return type *DFA|FA|STF|SST*

FAdo.fio.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|NFA|SFT*

FAdo.fio.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

FAdo.fio.saveToJson(fileName, aut, mode='w')

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

FAdo.fio.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

`FAdo.fio.toJson(aut)`

Json for a FA

Parameters `aut` (FA) – the automaton

Return type `str`

5.3 Constants

const *FAdo.fio.FAdoGrammar*

MODULE: REGULAR EXPRESSIONS (REEX)

Regular expressions manipulation

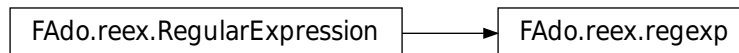
Regular expression classes and manipulation

6.1 Classes

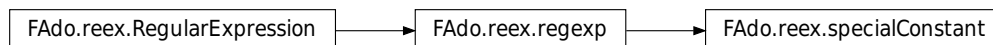
class `FAdo.reex.RegularExpression`
Abstract base class for all regular expression objects

class `FAdo.reex.regexp`(*sigma=None*)
Base class for regular expressions.

Variables *Sigma* – alphabet set of strings

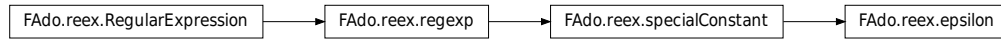


class `FAdo.reex.specialConstant`(*sigma=None*)
Base class for Epsilon and EmptySet



Parameters *sigma* – alphabet

class `FAdo.reex.epsilon`(*sigma=None*)
Class that represents the empty word.



Parameters **sigma** – alphabet

class `FAdo.reex.emptyset(sigma=None)`
 Class that represents the empty set.



Parameters **sigma** – alphabet

class `FAdo.reex.sigmaP(sigma=None)`

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

sigmaP: Class that represents the complement of the emptyset word (Σ^+)



Parameters **sigma** – alphabet

class `FAdo.reex.sigmaS(sigma=None)`

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

sigmaS: Class that represents the complement of the emptyset set (Σ^*)



Parameters **sigma** – alphabet

class `FAdo.reex.connective(arg1, arg2, sigma=None)`
 Base class for (binary) operations: concatenation, disjunction, etc



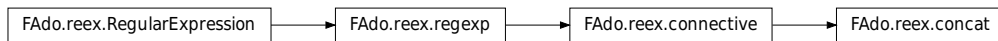
class `FAdo.reex.star`(*arg*, *sigma=None*)

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



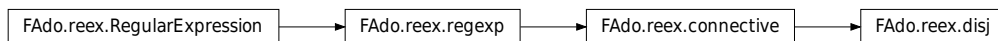
class `FAdo.reex.concat`(*arg1*, *arg2*, *sigma=None*)

Class for catenation operation on regular expressions.



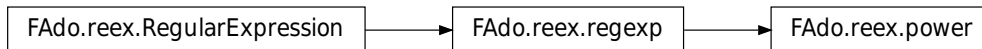
class `FAdo.reex.disj`(*arg1*, *arg2*, *sigma=None*)

Class for disjunction operation on regular expressions.



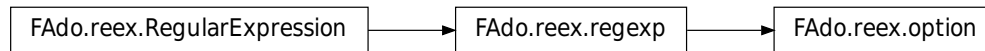
class `FAdo.reex.power`(*arg*, *n=1*, *sigma=None*)

Class for power operation on regular expressions.



class `FAdo.reex.option`(*arg*, *sigma=None*)

Class for option operation on regular expressions.



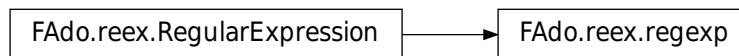
class `FAdo.reex.conj`(*arg1*, *arg2*, *sigma=None*)
Intersection operation of regexps

class `FAdo.reex.shuffle`(*arg1*, *arg2*, *sigma=None*)
Shuffle operation of regexps

class `FAdo.reex.atom`(*val*, *sigma=None*)
Simple atom (symbol)

Variables

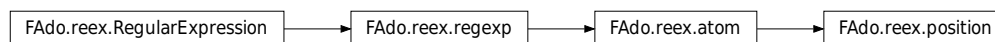
- **Sigma** – alphabet set of strings
- **val** – the actual symbol



Constructor of a regular expression symbol.

Parameters **val** – the actual symbol

class `FAdo.reex.position`(*val*, *sigma=None*)
Class for marked regular expression symbols.



Constructor of a regular expression symbol.

Parameters **val** – the actual symbol

class `FAdo.reex.sconnective`(*arg*, *sigma=None*)

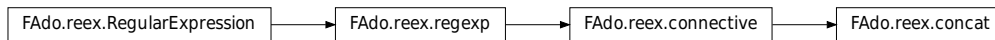
Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;

associativity of concatenation; identities Σ^* and Σ^+ . Connectives are: `sdisj`: disjunction
`sconj`: intersection `sconcat`: concatenation

For parsing use `str2sre`



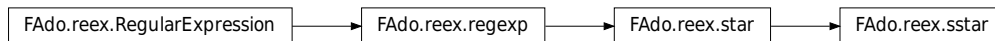
class `FAdo.reex.sconcat`(*arg*, *sigma=None*)
 Class that represents the concatenation operation.



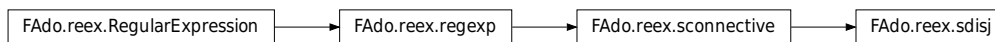
class `FAdo.reex.sstar`(*arg*, *sigma=None*)

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

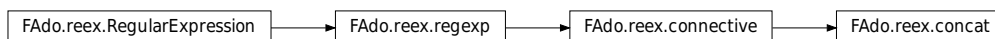
sstar: Class that represents Kleene star



class `FAdo.reex.sdisj`(*arg*, *sigma=None*)
 Class that represents the disjunction operation for special regular expressions.



class `FAdo.reex.sconj`(*arg*, *sigma=None*)
 Class that represents the conjunction operation.



class `FAdo.reex.snot`(*arg*, *sigma=None*)

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



class FAdo.reex.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.

Variables *reg* (*reex*) – regular expression

class FAdo.reex.dnode(*op*, *arg1=None*, *arg2=None*)

class FAdo.reex.m_atom(*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

class FAdo.reex.BuildRegexp(*context=None*)

Semantics of the FAdo grammars' regexps

class FAdo.reex.BuildRPNRegexp(*context=None*)

class FAdo.reex.BuildRPNSRE(*context=None*)

class FAdo.reex.BuildSRE(*context=None*)

Parser for sre

6.2 Functions

FAdo.reex.str2regexp(*s*, *parser=Lark(open('<string>'), 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*

`FAdo.reex.str2sre(s, parser=Lark(open('<string>'), parser='lalr', lexer='contextual', ...), sigma=None, strict=False)`

Reads a sre from string. Arguments as str2regex.

Return type *reex.sre*

`FAdo.reex.rpn2regex(s, sigma=None, strict=False)`

Reads a (simple) regex 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*

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

`FAdo.reex.to_s(r)`

Returns a sre from FAdo regex.

Parameters **r** (*regexp*) – the FAdo representation regex for a regular expression.

Return type *regexp*

MODULE: TRANSDUCERS (TRANSDUCERS)

Finite Tranducer Support

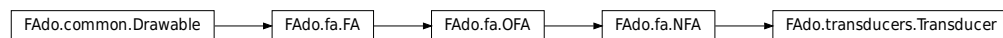
Transducer manipulation.

New in version 1.0.

7.1 Classes

class `FAdo.transducers.Transducer`

Base class for Transducers



Return type `object`

class `FAdo.transducers.SFT`

Standard Form Tranducer

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



Return type `object`

class `FAdo.transducers.NFT`

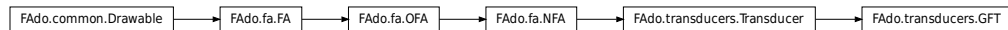
Normal Form Tranducer.

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



Return type `object`

class `FAdo.transducers.GFT`
General Form Transducer



Return type `object`

7.2 Functions

`FAdo.transducers.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*

`FAdo.transducers.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*

`FAdo.transducers.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 $N \cdot N \cdot M$ 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`

`FAdo.transducers.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*

`FAdo.transducers.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*

`FAdo.transducers.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*

MODULE: FINITE LANGUAGES (FL)

Finite languages and related automata manipulation

Finite languages manipulation

8.1 Classes

class `FAdo.fl.FL`(*wordsList=None, Sigma=None*)
Finite Language Class

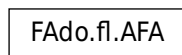
Variables

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

class `FAdo.fl.DFCA`
Deterministic Cover Automata class

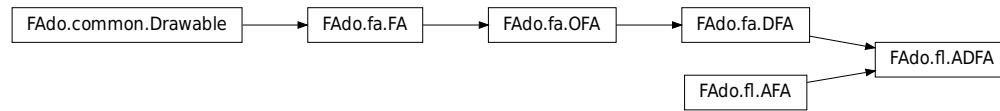


class `FAdo.fl.AFA`
Base class for Acyclic Finite Automata



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

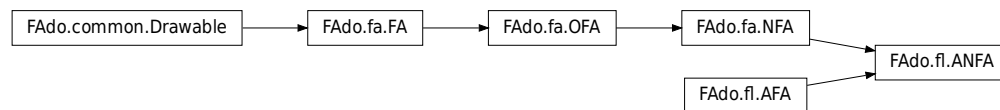
class `FAdo.fl.ADFA`
Acyclic Deterministic Finite Automata class



Changed in version 1.3.3.

class `FAdo.f1.ANFA`

Acyclic Nondeterministic Finite Automata class



Return type `object`

class `FAdo.f1.RndWGen(aut)`

Word random generator class

New in version 1.2.

Parameters `aut (ADFA)` – automata recognizing the language

8.2 Functions

`FAdo.f1.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

`FAdo.f1.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

`FAdo.f1.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*

`FAdo.fl.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

`FAdo.fl.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

`FAdo.fl.DFAtoADFA(aut)`

Transforms an acyclic DFA into a ADFA

Parameters `aut (DFA)` – the automaton to be transformed

Raises `notAcyclic` – if the DFA is not acyclic

Returns the converted automaton

Return type *ADFA*

`FAdo.fl.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.

MODULE: GRAPHS (GRAPH CREATION AND MANIPULATION)

Graph support

Basic Graph object support and manipulation

class `FAdo.graphs.Graph`

Graph base class

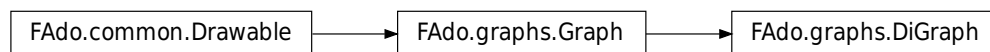
Variables

- **Vertices** (*list*) – Vertices' names
- **Edges** (*set*) – set of pairs (always sorted)



class `FAdo.graphs.DiGraph`

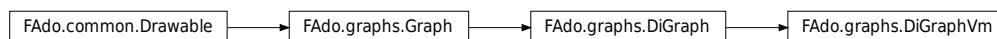
Directed graph base class



class `FAdo.graphs.DiGraphVm`

Directed graph with marked vertices

Variables **MarkedV** (*set*) – set of marked vertices



MODULE: CONTEXT FREE GRAMMARS MANIPULATION (CFG)

Context Free Grammars Manipulation.

Basic context-free grammars manipulation for building uniform random generetors

10.1 Classes

class `FAdo.cfg.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

class `FAdo.cfg.CNF(gram, mark='A@')`

No useless nonterminals or epsilon 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 nedded.

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

class `FAdo.cfg.cfgGenerator(cfgr, size)`

CFG uniform genetaror

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

class `FAdo.cfg.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: `reStringR-Generator (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

10.2 Functions

`FAdo.cfg.gRules(rules_list, rulesym='->', rhssep=None, rulesep='|')`
Transforms a list of rules into a grammar description.

Parameters

- **rules_list** – is a list of rule where rule is a string of the form: Word rulesym Word1 ... Word2 or Word rulesym []
- **rulesym** – LHS and RHS rule separator
- **rhssep** – RHS values separator (None for white chars)

Returns a grammar description

`FAdo.cfg.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*

10.3 Constants

const reGrammar

MODULE: RANDOM DFA GENERATOR (RNDFAP)

Random DFA generation (alternative version in python)

ICDFA Random generation binding

New in version 1.0.

11.1 Classes

class `FAdo.rndfap.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

class `FAdo.rndfap.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 FAdo.rndfap.ICDFArndIncomplete(*n*, *k*, *bias*=None, *seed*=0)
Incomplete IC DFA 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** – if a bias ≥ 1 or ≤ 0 is provided

Changed in version 1.3.4: seed added to the random generator

MODULE: RANDOM ADFA GENERATOR (RNDADFA)

Random ADFA generation

ADFA Random generation binding

New in version 1.2.1.

12.1 Classes

class `FAdo.rndadfa.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

MODULE: COMBO OPERATIONS (COMBOOPERATIONS)

Several combined operations for DFAs

Combined operations

13.1 Functions

`FAdo.comboperations.starConcat(fa1, fa2, strict=False)`

Star of concatenation of two languages: $(L1.L2)^*$

Parameters

- **fa1** (**DFA**) – first automaton
- **fa2** (**DFA**) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA*

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.

`FAdo.comboperations.concatWStar(fa1, fa2, strict=False)`

Concatenation combined with star: $(L1.L2)^*$

Parameters

- **fa1** (**DFA**) – first automaton
- **fa2** (**DFA**) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA*

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.

`FAdo.comboperations.starWConcat(fa1, fa2, strict=False)`

Star combined with concatenation: $(L1^*.L2)$

Parameters

- **fa1** (**DFA**) – first automaton

- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA*

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.comboperations.starDisj(fa1, fa2, strict=False)`
Star of Union of two DFAs: $(L1 + L2)^*$

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA*

See also:

Arto Salomaa, Kai Salomaa, and Sheng Yu. ‘State complexity of combined operations’. Theor. Comput. Sci., 383(2-3):140–152, 2007.

`FAdo.comboperations.starInter0(fa1, fa2, strict=False)`
Star of Intersection of two DFAs: $(L1 \& L2)^*$

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA*

See also:

Arto Salomaa, Kai Salomaa, and Sheng Yu. ‘State complexity of combined operations’. Theor. Comput. Sci., 383(2-3):140–152, 2007.

`FAdo.comboperations.starInter(fa1, fa2, strict=False)`
Star of Intersection of two DFAs: $(L1 \& L2)^*$

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA*

`FAdo.comboperations.disjWStar(f1, f2, strict=True)`
Union with star: $(L1 + L2^*)$

Parameters

- **f1** (DFA) – first automaton
- **f2** (DFA) – second automaton

- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA*

See also:

Yuan Gao and Sheng Yu. ‘State complexity of union and intersection combined with star and reversal’. CoRR, abs/1006.3755, 2010.

`FAdo.comboperations.interWStar(f1, f2, strict=True)`

Intersection with star: $(L1 \ \& \ L2^*)$

Parameters

- **f1** (*DFA*) – first automaton
- **f2** (*DFA*) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type *DFA*

See also:

Yuan Gao and Sheng Yu. ‘State complexity of union and intersection combined with star and reversal’. CoRR, abs/1006.3755, 2010.

MODULE: CODES (CODES)

Code theory module

New in version 1.0.

14.1 Classes

class `FAdo.codes.CodeProperty`(*name*, *alph*)

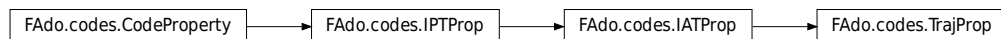
See also:

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

class `FAdo.codes.TrajProp`(*aut*, *Sigma*)

Class of trajectory properties



Constructor

Parameters

- **aut** (`DFA/NFA`) – regular expression over $\{0,1\}$
- **Sigma** (`set`) – the alphabet

class `FAdo.codes.IPTProp`(*aut*, *name=None*)

Input Preserving Transducer Property

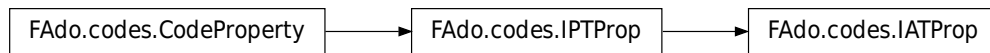


Variables

- **Aut** ([SFT](#)) – the transducer defining the property
- **Sigma** ([set](#)) – alphabet

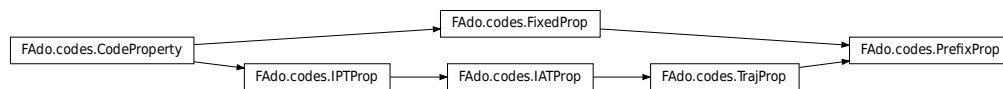
Constructor :param SFT aut: Input preserving transducer

class FAdo.codes.**IATProp**(aut, name=None)
Input Altering Transducer Property



Constructor :param SFT aut: Input preserving transducer

class FAdo.codes.**PrefixProp**(t)
Prefix Property



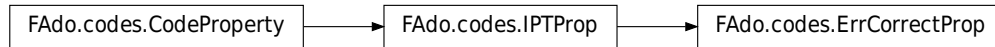
Constructor

Parameters

- **aut** ([DFA/NFA](#)) – regular expression over {0,1}
- **Sigma** ([set](#)) – the alphabet

FAdo.codes.**ErrDetectProp**
alias of [FAdo.codes.IPTProp](#)

class FAdo.codes.**ErrCorrectProp**(t)
Error Correcting Property



Constructor :param SFT aut: Input preserving transducer

14.2 Functions

`FAdo.codes.buildTrajPropS(regex, sigma)`

Builds a TrajProp from a string regexp

Parameters

- **regex** (*str*) – the regular expression
- **sigma** (*set*) – alphabet

Return type *TrajProp*

`FAdo.codes.buildIATPropF(fname)`

Builds a IATProp from a FAdo SFT file

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

Return type *IATProp*

`FAdo.codes.buildIPTPropF(fname)`

Builds a IPTProp from a FAdo SFT file

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

Return type *IPTProp*

`FAdo.codes.buildIATPropS(s)`

Builds a IATProp from a FAdo SFT string

Parameters **s** (*str*) – string containing SFT

Return type *IATProp*

`FAdo.codes.buildIPTPropS(s)`

Builds a IPTProp from a FAdo SFT string

Parameters **s** (*str*) – file name

Return type *IPTProp*

`FAdo.codes.buildErrorDetectPropF(fname)`

Builds an Error Detecting Property

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

Return type *ErrDetectProp*

`FAdo.codes.buildErrorCorrectPropF(fname)`

Builds an Error Correcting Property

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

Return type *ErrCorrectProp*

`FAdo.codes.buildErrorDetectPropS(s)`

Builds an Error Detecting Property from string

Parameters *s* (*str*) – transducer string

Return type *ErrDetectProp*

`FAdo.codes.buildErrorCorrectPropS(s)`

Builds an Error Correcting Property from string

Parameters *s* (*str*) – transducer string

Return type *ErrCorrectProp*

`FAdo.codes.buildPrefixProperty(alphabet)`

Builds a Prefix Code Property

Parameters *alphabet* (*set*) – alphabet

Return type *PrefixProp*

`FAdo.codes.editDistanceW(auto)`

Compute the edit distance of a given regular language accepted by the NFA via Input-altering transducer.

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]

Parameters *auto* (*NFA*) – language recogniser

Returns The edit distance of the given regular language plus a witness pair

Return type *tuple*

`FAdo.codes.exponentialDensityP(aut)`

Checks if language density is exponential

Using breadth first search (BFS)

Attention: *aut* should not have Epsilon transitions

Parameters *aut* (*NFA*) – the representation of the language

Return type *bool*

`FAdo.codes.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*

```
# ===== # Module: Set Specification
Transducers and Automata (sst) # =====
# # .. automodule:: sst # :no-members: # # ----- # Class PSP # ----- # # .. autoclass:: PSP # =====
Small Tutorial =====
```


A SMALL TUTORIAL FOR FADO

FAdo system is a set tools for regular languages manipulation.

Regular languages can be represented by regular expressions (regexp) 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:

```
>>> 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.

General conventions

Methods which name ends in `P` test if the object verifies a given property and return `True` or `False`.

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.

DFAs

The following example shows how to build a DFA that accepts the words of $\{0,1\}^*$ that are multiples of 3.

```
>>> m3= DFA()
>>> m3.setSigma(['0','1'])
>>> m3.addState('s1')
>>> m3.addState('s2')
```

(continues on next page)

(continued from previous page)

```
>>> 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.

```
>>> m3
DFA((['s1', 's2', 's3'], ['1', '0'], 's1', ['s1'], "[(('s1', '1', 's2'), ('s1', '0', 's1'
→ '), ('s2', '1', 's1'), ('s2', '0', 's3'), ('s3', '1', 's3'), ('s3', '0', 's2'))]"))
>>> m3.evalWordP("011")
True
>>> m3.evalWordP("1011")
False
>>>
```

If graphviz is installed it is also possible to display the diagram of an automaton as follows:

```
>>>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

```
>>> m3=readFromFile("mul3.fa")[0]
```

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:

```
>>> len(m3.States)
3
```

For the number of Transitions the countTransitions() method must be used

```
>>> m3.countTransitions()
6
```

To minimize a DFA any of the minimization algorithms implemented can be used:

```
>>> min=m3.minimalHopcroft()
```

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).

```
>>> u = m3 | ~m3
>>> u
DFA(([(1, 1), (0, 0), (2, 2)], set(['1', '0']), 0, set([0, 1, 2]), {0: {'1': 1, '0': 0}, 1: {'1': 0, '0': 2}, 2: {'1': 2, '0': 1}}))
```

```
>>> m = u.minimal()
>>> m
DFA((['(1, 1)'], ['1', '0'], '(1, 1)', ['(1, 1)'], "['(1, 1)', '1', '(1, 1)'], ('(1, 1)', '0', '(1, 1)'))
```

State names can be renamed in-place using:

```
>>> m.renameStates(range(len(m)))
```

```
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)

```
>>> u.witness()
'@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.

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.

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.

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:

```
>>> r = star(disj(regex("a"), concat(regex("b"), regex("a"))))
>>> print r
(a + (b a))*
```

But we can convert a string to a regexp class or subclass, using the method `str2regexp`.

```
>>> 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 class `regexp` does not deal with regular expressions module ACI properties (associativity, commutativity and idempotence of the union) (see class `xre`), 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.

```
>>> r.compare(str2regexp("(a*(ba)*a*)*\"))
True
>>>
```

Converting Finite Automata to Regular Expressions

For pedagogical purposes, it is implemented a recursive method that constructs a regular expression equivalent to a given DFA (`regexp`).

```
>>> print m3.regexp()
((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.

```
>>> print m3.reCG()
((0 + (1 1)) + (((1 0) (1 + (0 0))* (0 1)))*
```

Converting Regular Expressions to Finite Automata

Several methods to convert between regular expressions and NFAs are implemented. With the Thompson construction a NFA with epsilon transitions is obtained (`nfaThompson`). Epsilon free NFAs can be obtained by the Glushkov method (Position automata) (`nfaPosition`), the partial derivatives method (`nfaPD` – several implementations) or by the follow method (`nfaFollow`). The two last methods usually allows to obtain smaller NFAs.

```
>>> r.nfaThompson()
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')]]")
```

```
>>> r.nfaPosition()
NFA((['Initial', "('a', 1)", "('b', 2)", "('a', 3)"], ['a', 'b'], ['Initial'], ['Initial
↳ ', "('a', 1)", "('a', 3)"], '[(\'Initial\', \'a\', "('a\', 1)"), (\'Initial\', \'b\',
↳ (\'b\', 2)"), ("(\'a\', 1)", \'a\', "(\'a\', 1)"), ("(\'a\', 1)", \'b\', "(\'b\', 2)
↳ ")", ("(\'b\', 2)", \'a\', "(\'a\', 3)"), ("(\'a\', 3)", \'a\', "(\'a\', 1)"), ("(\'a\',
↳ 3)", \'b\', "(\'b\', 2)"))])
```

```
>>> r.nfaPD()
NFA((['(a + (b a))*', 'a (a + (b a))*'], ['a', 'b'], ['(a + (b a))*'], ['(a + (b a))*'],
↳ "[star(disj(regexp(a),concat(regexp(b),regexp(a)))), 'a', star(disj(regexp(a),
↳ concat(regexp(b),regexp(a))))), (star(disj(regexp(a),concat(regexp(b),regexp(a))))), 'b
↳ ', concat(regexp(a),star(disj(regexp(a),concat(regexp(b),regexp(a))))),
↳ (concat(regexp(a),star(disj(regexp(a),concat(regexp(b),regexp(a))))), 'a',
↳ star(disj(regexp(a),concat(regexp(b),regexp(a))))))"])
```

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.

```
>>> r.nfaPosition().toDFA().minimal(complete=False)
DFA((['\0', '2'], ['a', 'b'], '\0', ['\0'], "[('0', 'a', '0'), ('0', 'b', '2'), ('2', 'a',
↳ '0')])")
>>> m3 == m3.reCG().nfaPD().toDFA().minimal()
True
>>>
```


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 grail.py: compatibility with GRAIL

module transducers.py: several classes and methods for transducers

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, [37](#)
- FAdo.codes, [47](#)
- FAdo.comboperations, [43](#)
- FAdo.common, [13](#)
- FAdo.conversions, [9](#)
- FAdo.fa, [5](#)
- FAdo.fio, [15](#)
- FAdo.fl, [31](#)
- FAdo.graphs, [35](#)
- FAdo.reex, [19](#)
- FAdo.rndadfa, [41](#)
- FAdo.rndfap, [39](#)
- FAdo.transducers, [27](#)

A

ADFA (class in *FAdo.fl*), 31
 ADFArnd (class in *FAdo.rndadfa*), 41
 AFA (class in *FAdo.fl*), 31
 ANFA (class in *FAdo.fl*), 32
 atom (class in *FAdo.reex*), 22

B

buildErrorCorrectPropF() (in module *FAdo.codes*), 49
 buildErrorCorrectPropS() (in module *FAdo.codes*), 50
 buildErrorDetectPropF() (in module *FAdo.codes*), 49
 buildErrorDetectPropS() (in module *FAdo.codes*), 50
 BuildFadoObject (class in *FAdo.fio*), 15
 buildIATPropF() (in module *FAdo.codes*), 49
 buildIATPropS() (in module *FAdo.codes*), 49
 buildIPTPropF() (in module *FAdo.codes*), 49
 buildIPTPropS() (in module *FAdo.codes*), 49
 buildPrefixProperty() (in module *FAdo.codes*), 50
 BuildRegex (class in *FAdo.reex*), 24
 BuildRPNRegex (class in *FAdo.reex*), 24
 BuildRPNSRE (class in *FAdo.reex*), 24
 BuildSRE (class in *FAdo.reex*), 24
 buildTrajPropS() (in module *FAdo.codes*), 49

C

cfgGenerator (class in *FAdo.cfg*), 37
 CFGrammar (class in *FAdo.cfg*), 37
 CNF (class in *FAdo.cfg*), 37
 CodeProperty (class in *FAdo.codes*), 47
 concat (class in *FAdo.reex*), 21
 concatWStar() (in module *FAdo.comboperations*), 43
 conj (class in *FAdo.reex*), 22
 connective (class in *FAdo.reex*), 20
 createInputAlteringSIDTrans() (in module *FAdo.codes*), 50
 cutPoints() (in module *FAdo.conversions*), 9

D

dag (class in *FAdo.reex*), 24
 delta (*FAdo.fa.FA* attribute), 5
 delta (*FAdo.fa.SemiDFA* attribute), 5
 DFA (class in *FAdo.fa*), 6
 DFA2regexDijkstra() (in module *FAdo.conversions*), 11
 DFAtoADFA() (in module *FAdo.fl*), 33
 DFCA (class in *FAdo.fl*), 31
 DiGraph (class in *FAdo.graphs*), 35
 DiGraphVm (class in *FAdo.graphs*), 35
 disj (class in *FAdo.reex*), 21
 disjWStar() (in module *FAdo.comboperations*), 44
 dnode (class in *FAdo.reex*), 24
 Drawable (class in *FAdo.common*), 13

E

editDistanceW() (in module *FAdo.codes*), 50
 emptyset (class in *FAdo.reex*), 20
 EnumL (class in *FAdo.fa*), 7
 epsilon (class in *FAdo.reex*), 19
 ErrCorrectProp (class in *FAdo.codes*), 48
 ErrDetectProp (in module *FAdo.codes*), 48
 exponentialDensityP() (in module *FAdo.codes*), 50

F

FA (class in *FAdo.fa*), 5
 FA2GFA() (in module *FAdo.conversions*), 9
 FA2regexCG() (in module *FAdo.conversions*), 10
 FA2regexCG_nn() (in module *FAdo.conversions*), 10
 FA2regexDynamicCycleHeuristic() (in module *FAdo.conversions*), 11
 FA2regexSE() (in module *FAdo.conversions*), 10
 FA2regexSE_nn() (in module *FAdo.conversions*), 11
 FA2regexSEO() (in module *FAdo.conversions*), 10
 FA2regexStaticCycleHeuristic() (in module *FAdo.conversions*), 11
 FAllRegExps() (in module *FAdo.conversions*), 9
 FAdo.cfg
 module, 37
 FAdo.codes
 module, 47

[FAdo.comboperations](#)
 [module](#), 43
[FAdo.common](#)
 [module](#), 13
[FAdo.conversions](#)
 [module](#), 9
[FAdo.fa](#)
 [module](#), 5
[FAdo.fio](#)
 [module](#), 15
[FAdo.fl](#)
 [module](#), 31
[FAdo.graphs](#)
 [module](#), 35
[FAdo.reex](#)
 [module](#), 19
[FAdo.rndadfa](#)
 [module](#), 41
[FAdo.rndfap](#)
 [module](#), 39
[FAdo.transducers](#)
 [module](#), 27
[FAeliminateSingles\(\)](#) (in module *FAdo.conversions*), 10
[Final](#) (*FAdo.fa.FA* attribute), 5
[FL](#) (class in *FAdo.fl*), 31

G

[genRandomTrie\(\)](#) (in module *FAdo.fl*), 33
[genRndTrieBalanced\(\)](#) (in module *FAdo.fl*), 32
[genRndTriePrefix\(\)](#) (in module *FAdo.fl*), 33
[genRndTrieUnbalanced\(\)](#) (in module *FAdo.fl*), 32
[GFA](#) (class in *FAdo.conversions*), 9
[GFT](#) (class in *FAdo.transducers*), 28
[Graph](#) (class in *FAdo.graphs*), 35
[gRules\(\)](#) (in module *FAdo.cfg*), 38

H

[hypercodeTransducer\(\)](#) (in module *FAdo.transducers*), 28

I

[IATProp](#) (class in *FAdo.codes*), 48
[ICDFArgen](#) (class in *FAdo.rndfap*), 39
[ICDFArnd](#) (class in *FAdo.rndfap*), 39
[ICDFArndIncomplete](#) (class in *FAdo.rndfap*), 39
[infixTransducer\(\)](#) (in module *FAdo.transducers*), 28
[Initial](#) (*FAdo.fa.FA* attribute), 5
[interWStar\(\)](#) (in module *FAdo.comboperations*), 45
[IPTProp](#) (class in *FAdo.codes*), 47
[isLimitExceed\(\)](#) (in module *FAdo.transducers*), 28

M

[m_atom](#) (class in *FAdo.reex*), 24

[module](#)
 [FAdo.cfg](#), 37
 [FAdo.codes](#), 47
 [FAdo.comboperations](#), 43
 [FAdo.common](#), 13
 [FAdo.conversions](#), 9
 [FAdo.fa](#), 5
 [FAdo.fio](#), 15
 [FAdo.fl](#), 31
 [FAdo.graphs](#), 35
 [FAdo.reex](#), 19
 [FAdo.rndadfa](#), 41
 [FAdo.rndfap](#), 39
 [FAdo.transducers](#), 27

N

[NFA](#) (class in *FAdo.fa*), 6
[NFAR](#) (class in *FAdo.fa*), 6
[NFT](#) (class in *FAdo.transducers*), 27

O

[OFA](#) (class in *FAdo.fa*), 6
[option](#) (class in *FAdo.reex*), 21
[outfixTransducer\(\)](#) (in module *FAdo.transducers*), 28

P

[position](#) (class in *FAdo.reex*), 22
[power](#) (class in *FAdo.reex*), 21
[PrefixProp](#) (class in *FAdo.codes*), 48
[prefixTransducer\(\)](#) (in module *FAdo.transducers*), 29

R

[readFromFile\(\)](#) (in module *FAdo.fio*), 15
[readOneFromFile\(\)](#) (in module *FAdo.fio*), 16
[readOneFromString\(\)](#) (in module *FAdo.fio*), 16
[regex](#) (class in *FAdo.reex*), 19
[RegularExpression](#) (class in *FAdo.reex*), 19
[reStringRGenerator](#) (class in *FAdo.cfg*), 37
[RndWGen](#) (class in *FAdo.fl*), 32
[rpn2regex\(\)](#) (in module *FAdo.reex*), 25

S

[saveToFile\(\)](#) (in module *FAdo.fio*), 16
[saveToJson\(\)](#) (in module *FAdo.fio*), 16
[saveToString\(\)](#) (in module *FAdo.fa*), 7
[saveToString\(\)](#) (in module *FAdo.fio*), 16
[sconcat](#) (class in *FAdo.reex*), 23
[sconj](#) (class in *FAdo.reex*), 23
[sconnective](#) (class in *FAdo.reex*), 22
[sdisj](#) (class in *FAdo.reex*), 23
[SemiDFA](#) (class in *FAdo.fa*), 5
[SFT](#) (class in *FAdo.transducers*), 27
[shuffle](#) (class in *FAdo.reex*), 22

`Sigma` (*FAdo.fa.FA attribute*), 5
`Sigma` (*FAdo.fa.SemiDFA attribute*), 6
`sigmaInitialSegment()` (*in module FAdo.fl*), 32
`sigmaP` (*class in FAdo.reex*), 20
`sigmaS` (*class in FAdo.reex*), 20
`smallAlphabet()` (*in module FAdo.cfg*), 38
`snot` (*class in FAdo.reex*), 23
`SP2regexp()` (*in module FAdo.conversions*), 10
`specialConstant` (*class in FAdo.reex*), 19
`SSemiGroup` (*class in FAdo.fa*), 6
`sstar` (*class in FAdo.reex*), 23
`star` (*class in FAdo.reex*), 21
`starConcat()` (*in module FAdo.comboperations*), 43
`starDisj()` (*in module FAdo.comboperations*), 44
`starInter()` (*in module FAdo.comboperations*), 44
`starInter0()` (*in module FAdo.comboperations*), 44
`starWConcat()` (*in module FAdo.comboperations*), 43
`States` (*FAdo.fa.FA attribute*), 5
`States` (*FAdo.fa.SemiDFA attribute*), 5
`str2regexp()` (*in module FAdo.reex*), 24
`str2sre()` (*in module FAdo.reex*), 25
`stringToADFA()` (*in module FAdo.fl*), 33
`stringToDFA()` (*in module FAdo.fa*), 7
`suffixTransducer()` (*in module FAdo.transducers*), 29

T

`to_s()` (*in module FAdo.reex*), 25
`toJson()` (*in module FAdo.fio*), 16
`TrajProp` (*class in FAdo.codes*), 47
`Transducer` (*class in FAdo.transducers*), 27

W

`Word` (*class in FAdo.common*), 13