FAdo Documentation

Release 1.3.4

Rogério Reis & Nelma Moreira

Aug 02, 2017

CONTENTS

1 What is FAdo? 3

1.1 Regular Languages . . . 3
1.2 Finite Languages . . . 3
1.3 Transducers . . . 4
1.4 Codes . . . 4

2 Module: Finite Automata (`fa`) 5

2.1 Class FA (abstract class for Finite Automata) . . . 5
2.2 Class OFA (one-way finite automata class) . . . 11
2.3 Class DFA (Deterministic Finite Automata) . . . 15
2.4 Class NFA (Nondeterministic Finite Automata) . . . 27
2.5 Class NFAr (Nondeterministic Finite Automata w/ reverse transition f.) . . . 35
2.6 Class GFA (Generalized Finite Automata) . . . 37
2.7 Class SSemiGroup (Syntactic SemiGroup) . . . 39
2.8 Class EnumL (Language Enumeration) . . . 39
2.9 Functions . . . 40

3 Module: Common Definitions (`common`) 43

3.1 Class Word . . . 43

4 Module: FAdo IO Functions (`fio`) 45

4.1 Class ParserFAdo (Yappy parser for FAdo FA files) . . . 45
4.2 Functions . . . 45

5 Module: Regular Expressions (`reex`) 47

5.1 Class regexp (regular expression) . . . 47
5.2 Class specialConstant . . . 51
5.3 Class epsilon . . . 52
5.4 Class emptyset . . . 53
5.5 Class sigmaP . . . 54
5.6 Class sigmaS . . . 55
5.7 Class connective . . . 56
5.8 Class star . . . 56
5.9 Class concat . . . 57
5.10 Class disj . . . 58
5.11 Class power . . . 59
5.12 Class option . . . 60
5.13 Class conj (intersection) . . . 61
5.14 Class shuffle . . . 61
5.15 Class atom . . . 61
5.16 Class position . . . 65
5.17 Class ParseReg . . . 66
5.18 Class sconnective (special connective) . . . 66
5.19 Class sconcat . . . 67
5.20 Class sstar . . . 68

5.21 Class sdisj 68
5.22 Class sconj 69
5.23 Class snot 70
5.24 Functions 71

6 Module: Transducers (`transducers`) 73
6.1 Class Transducer 73
6.2 Class SFT (Standard Form Transducers) 73
6.3 Functions 78

7 Module: Finite Languages (`fl`) 79
7.1 Class FL (Finite Language) 79
7.2 Class DFCA (Deterministic Finite Cover Automata) 80
7.3 Class AFA (Acyclic Finite Automata) 81
7.4 Class ADFA (Acyclic Deterministic Finite Automata) 82
7.5 Class ANFA (Acyclic Non-deterministic Finite Automata) 84
7.6 Class RndWGen (Random Word Generator) 85
7.7 Functions 86

8 Module: graphs (graph creation and manipulation) 87

9 Module: Context Free Grammars Manipulation (`cfg`) 91
9.1 Class CFGrammar (Context Free Grammar) 91
9.2 Class CNF 91
9.3 Class cfgGenerator 92
9.4 Class reStringRGenerator (Reg Exp Generator) 92
9.5 Functions 92

10 Module: Random DFA Generator (`rndfa`) 95
10.1 Class ICDFArgen (Generator container) 95
10.2 Class ICDFArnd (Complete ICDFA random generator) 95
10.3 Class ICDFArndIncomple (Incomplete ICDFA generator) 96

11 Module: Random ADFA Generator (`rndadfa`) 97
11.1 Class ADFArnd (ADFA random generator) 97

12 Module: Combo Operations (`combooperations`) 101

13 Module: Codes (`codes`) 105
13.1 Class CodeProperty 105
13.2 Class TrajProp 105
13.3 Class IPTProp 106
13.4 Class IATProp 107
13.5 Class PrefixProp 108
13.6 Class ErrDetectProp 108
13.7 Class ErrCorrectProp 108
13.8 Functions 109

14 Module: Grail Compatibility (`grail`) 113
14.1 Class ParserGrail 113
14.2 Class Grail 113
14.3 Functions 114

15 Module: Verso Language (`verso`) 115

16 A small tutorial for FAdo 117

17 Indices and tables 123

Python Module Index 125

FAdo: Tools for Language Models Manipulation

Authors: Rogério Reis & Nelma Moreira

The support of transducers and all its operations, 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: 1.3.4

Faculdade de Ciências da Universidade do Porto

Centro de Matemática da Universidade do Porto

CHAPTER

ONE

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

CHAPTER

TWO

MODULE: FINITE AUTOMATA (`FA`)

Finite automata manipulation.

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

2.1 Class FA (abstract class for Finite Automata)

class `fa.`**`FA`**

Bases: `common.Drawable`

Base class for Finite Automata.

Note: 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 **`DFAepsilonRedefenition`** – 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*)

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

Parameters **`name`** (*`object`*) – Name of the state to be added

Returns Current number of states (the new state index)

Return type int

Raises **DuplicateName** – if a state with that name already exists

conjunction (*other*)
A simple literate invocation of __and__

Parameters **other** – the other FA

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

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

Raises **DFAstateUnknown** – if state index does not exist

disj (*other*)
Another simple literate invocation of __or__

Parameters **other** – the other FA

New in version 0.9.6.

disjunction (*other*)
A simple literate invocation of __or__

Parameters **other** – the other FA

dotDrawState (*sti*, *sep='\n'*, *strict=False*, *maxLblSz=6*)
Draw a state in dot format

Parameters

- **sti** (*int*) – index of the state
- **sep** (*str*) – separator
- **maxLblSz** – max size of labels before getting removed
- **strict** – use limitations of label sizes

Return type str

dotDrawTransition (*st1*, *sym*, *st2*, *sep*)
Draw a transition in dot format

Parameters

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

- **sep** (*str*) – separator

dotFormat (*size='20, 20'*, *direction='LR'*, *sep='\n'*, *strict=False*, *maxLblSz=6*)
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 1.2.1.

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

Attention: works inplace

New in version 1.2.

equivalentP (*other*)
Test equivalence

Parameters **other** – the other automata

Return type bool

New in version 0.9.6.

evalSymbol ()
Evaluation of a single symbol

finalP (*state*)
Tests if a state is final

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

Return type bool

finalsP (*states*)
Tests if al the states in a set are final

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

Return type bool

New in version 1.0.

hasStateIndexP (*st*)
Checks if a state index pertains to an FA

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

Return type bool

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 of int

Raises **DFAstateUnknown** – if a state name is unknown

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 of States

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

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

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

Attention: the object is modified in place

renameStates (*nameList=None*)
Renames all states using a new list of names.

Parameters **nameList** (*list*) – list of new names

Raises **DFAerror** – if provided list is too short

Returns self

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*

same_nullability(*s1*, *s2*)
Tests if this two states have the same nullability

Parameters

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

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: it 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(*symbolSet*)
Defines the alphabet for the FA.

Parameters **symbolSet** (*list | set*) – alphabet symbols

stateIndex(*name*, *autoCreate=False*)
Index of given state name.

Parameters

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

Returns state index

Return type int

Raises **DFAstateUnknown** – 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 (**args*, ***kwargs*)
Index of given state name.

Parameters

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

Returns state index

Return type int

Raises DFAstateUnknown – if the state name is unknown and autoCreate==False

Deprecated since version 1.0: Use: `stateIndex()` instead

succintTransitions ()
Colapsed transitions

union (*other*)
A simple literate invocation of __or__

Parameters other – right hand operand

words (*stringo=True*)
Lexicografical word generator

> **Attention:** does not generate the empty word

Parameters stringo (*bool*) – are words strings?

New in version 0.9.8.

2.1.1 Class SemiDFA (Semi-Automata class)

class fa.**SemiDFA**
Bases: `common.Drawable`

Class of automata without initial or final states

Variables

- **States** (*list*) – list of states
- **delta** (*dict*) – transition function
- **Sigma** (*set*) – alphabet

dotDrawState (*sti*, *sep='\n'*)
Dot representation of a state

Parameters

- **sti** (*int*) – state index
- **sep** (*str*) – separator

Return type str

static dotDrawTransition (*st1*, *lbl1*, *st2*, *sep='\n'*)
Draw a transition in dot format

Parameters

- **st1** (*str*) – departing state
- **lbl1** (*str*) – label

- **st2** (*str*) – arriving state
- **sep** (*str*) – separator

Return type str

dotFormat (*size='20, 20', direction='LR', sep='\n'*)
Dot format of automata

Parameters

- **size** (*str*) – image size
- **direction** (*str*) – direction of drawing
- **sep** (*str*) – separator

Return type str

2.2 Class OFA (one-way finite automata class)

class fa.**OFA**
Bases: *fa.FA*

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

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

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

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. http://www.dcc.fc.up.pt/~nam/publica/spa07.pdf

---

Note: Automata must be Serial-Parallel

---

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

Return type bool

addTransition (*st1, sym, st2*)
Add transition :param int st1: departing state :param str sym: label :param int st2: arriving state

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

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

cutPoints()
Set of FA's cut points

Returns set of states

Return type set of int

deleteStates(*del_states*)
To be implemented below

Parameters **del_states** (*list*) – states to be deleted

static **dotDrawTransition**(*st1*, *label*, *st2*, *sep='\n'*)
Draw a transition in Dot Format

Parameters

- **st1** (*str*) – starting state
- **st2** (*str*) – ending state
- **label** (*str*) – symbol
- **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

eliminateSingles()
Eliminates every state that only have one successor and one predecessor.

Returns GFA after eliminating states

Return type *GFA*

eliminateStout(*st*)
Eliminate all transitions outgoing from a given state

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

Attention: performs in place alteration of the automata

New in version 0.9.6.

emptyP()
Tests if the automaton accepts a empty language

Return type bool

New in version 1.0.

evalNumberOfStateCycles()
Evaluates the number of cycles each state participates

Returns state->list of cycle lengths

Return type dict

evalSymbol()
Eval symbol

finalCompP(*s*)
To be implemented below

Parameters **s** – state

Return type list

initialComp()
Initial component

Return type list

minimalBrzozowski()
Constructs the equivalent minimal DFA using Brzozowski's algorithm

Returns equivalent minimal DFA

Return type *DFA*

minimalBrzozowskiP()
Tests if the FA is minimal using Brzozowski's algorithm

Return type bool

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

Returns the equivalent regular expression

Return type *reex.regexp*

reCG_nn()
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.

Returns the equivalent regular expression

Return type *reex.regexp*

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

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 Descriptional Complexity of Formal Systems (DCFS10), pages 169-180.2010. DOI: 10.4204/EPTCS.31.16

reStaticCycleHeuristic()
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

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 Descriptional Complexity of Formal Systems (DCFS10), pages 169-180.2010. DOI: 10.4204/EPTCS.31.16

re_stateElimination(*order=None*)
Regular expression from state elimination whose language is recognised by the FA. The FA is normalized before the state elimination.

Parameters **order** (*list*) – state elimination sequence

Returns the equivalent regular expression

Return type *reex.regexp*

re_stateElimination_nn(*order=None*)
Regular expression from state elimination whose language is recognised by the FA. The FA is not normalized before the state elimination.

Parameters **order** (*list*) – state elimination sequence

Returns the equivalent regular expression

Return type *reex.regexp*

regexpSE()
A regular expression obtained by state elimination algorithm whose language is recognised by the FA.

Returns the equivalent regular expression

Return type *reex.regexp*

stateChildren(*s*)
To be implemented below

Parameters **s** – state

Return type list

succintTransitions()
Collapsed transitions

toGFA()
To be implemented below

topoSort()
Topological order for the FA

Returns List of state indexes

Return type list of int

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.

Attention: in place transformation

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

Returns bool

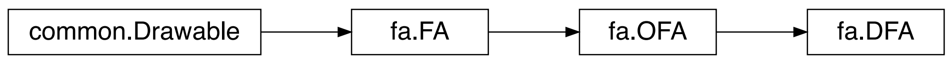
uniqueRepr()
Abstract method

usefulStates()
To be implemented below

2.3 Class DFA (Deterministic Finite Automata)

class fa.**DFA**
Bases: *fa.OFA*

Class for Deterministic Finite Automata.



Delta(*state*, *symbol*)
Evaluates the action of a symbol over a state

Parameters

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

Returns the action of symbol over state

Return type int

aEquiv()
Computes almost equivalence, used by hyperMinimial

Returns partition of states

Return type dictionary

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** (*str*) – symbol consumed

Raises DFAnotNFA – 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*

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.

Attention: The object is modified in place.

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 tey are not already a dead state. Only in the negative case a new (dead) state is added to the automaton.

Return type *DFA*

Attention: The object is modified in place. If the alphabet is empty nothing is done

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

Returns bool

completeProduct (*other*)
Product structure

Parameters **other** – the other DFA

computeKernel ()
The Kernel of a ICDFA 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`) – the second DFA

Returns the result of the concatenation

Return type *DFA*

Raises **DFAdifferentSigma** – if alphabet are not equal

concatI (*fa2*, *strict=False*)
Concatenation of two DFAs.

Parameters

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

Returns the result of the concatenation

Return type *DFA*

Raises DFAdifferentSigma – 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

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

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 int representing states

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.

dist ()

Evaluate the distinguishability language for a DFA

Return type *DFA*

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

Returns set of minimal distinguishing words

Return type *FL*

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*

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

distRMin()

Compute distRMin for DFA

:rtype FL

..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*

..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*

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`) – 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** – if symbol not in alphabet
- **DFAstopped** – 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** – 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 (*wrd*)
Evaluates a word

Parameters **wrd** (*Word*) – 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*)
Hyperminization of a minimal DFA

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

Returns an hyperminimal DFA

Return type *DFA*

See also:

M. Holzer and A. Maletti, An nlogn 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 infix(L(a))

Return type *DFA*

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 of States

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*

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*

minimalHKP ()
Tests the DFA's minimality using Hopcroft and Karp's state equivalence algorithm

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.

minimalHopcroft ()
Evaluates the equivalent minimal complete DFA using Hopcroft algorithm

Returns equivalent minimal DFA

Return type *DFA*

See also:

John Hopcroft,An nlog{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*

See also:

13. Almeida and N. Moreira and 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*

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*

minimalMooreSqP()

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

Return type bool

minimalNCompleteP()

Tests if a non necessarely 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*

Raises **DFAnotComplete** – 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) – 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*

New in version 1.1.

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 meaningfull.

Parameters **other** – the other DFA

Return type *DFA*

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 meaningfull state names

New in version 1.3.3.

Parameters

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

Return type *DFA*

regexp ()
Returns a regexp for the current DFA considering the recursive method. Very inefficent.

Returns a regexp equivalent to the current DFA

Return type *reex.regexp*

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) – DFA in which the reverse function will be stored

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*)
Shuffle of two languages: L1 W L2

Parameters

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

Return type *DFA*

See also:

C. Câmpeanu, K. Salomaa and S. Yu, *Tight lower bound for the state complexity of shuffle 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) – the other automaton

Return type *DFA*

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 Descriptional 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*

starI()
Star of an incomplete DFA.

Returns the Kleene closure DFA

Return type *DFA*

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()

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

Return type dfa

New in version 1.1.

succintTransitions()
Collects the transition information in a compact way suitable for graphical representation. :rtype: list of tupples

New in version 0.9.8.

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

Return type *DFA*

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*

syncWords()
Evaluates the regular expression corresponding to the synchronizing pwords of the automata.

Returns a regular expression of the sync words of the automata

Return type *reex.regexp*

toADFA()
Try to convert DFA to ADFA

Returns the same automaton as a ADFA

Return type *ADFA*

Raises **notAcyclic** – 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*

toGFA()
Creates a GFA equivalent to DFA

Returns GFA deep copy

Return type *GFA*

toNFA()
Migrates a DFA to a NFA as dup()

Returns DFA seen as new NFA

Return type *NFA*

uniqueRepr()
Normalise unique string for the string icdfa's representation.

See also:

TCS 387(2):93-102, 2007 http://www.ncc.up.pt/~nam/publica/tcsamr06.pdf

Returns normalised representation

Return type list

Raises **DFAnotComplete** – if DFA is not complete

unmark()

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

Returns a NFA

Return type *NFA*

usefulStates(*initial_states=None*)

Set of states reacheable 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

static vDescription()

Generation of Verso interface description

New in version 0.9.5.

Returns the interface list

witness()

Witness of non emptyness

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) – the other DFA

Returns a witness word

Return type list of symbols

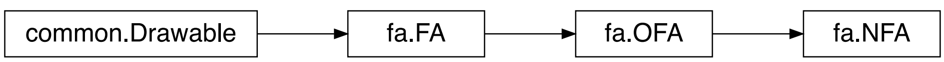
Raises **DFAequivalent** – if automata are equivalent

2.4 Class NFA (Nondeterministic Finite Automata)

class fa.**NFA**

Bases: *fa.OFA*

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



addEpsilonLoops()

Add epsilon loops to every state :return: self

> **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** – 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 of frozensets

See also:

Ilie&Yu, 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 of tuples

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

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** (*NFA | DFA*) – the other NFA

Returns the result of the concatenation

Return type *NFA*

countTransitions ()
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** (*str*) – 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*

deterministicP ()
Verify whether this NFA is actually deterministic

Return type bool

dotFormat (*size='20, 20'*, *direction='LR'*, *sep='\n'*, *strict=False*, *maxLblSz=6*)
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 1.2.1.

dup()

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

Return type *NFA*

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 epslilon-transitions with no state addition

> **Attention:** in-place modification

eliminateTSymbol(*symbol*)

Delete all trasitions through a given symbol

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

> **Attention:** in place alteration of the automata

New in version 0.9.6.

enumNFA(*n=None*)

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

Return type list of strings or None

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 of int

> **Attention:** `st` must exist.

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 of states

equivReduced (*equiv_classes*)
Equivalent NFA reduced according to given equivalence classes.

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

Returns Equivalent NFA

Return type *NFA*

evalSymbol (*stil*, *sym*)
Set of states reacheable 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** – 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

Returns :: bool

followFromPosition ()
computes follow automaton from a position automaton :rtype: NFA

half ()
Half operation

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 froms 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*

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*

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

minimal()

Evaluates the equivalent minimal DFA

Returns equivalent minimal DFA

Return type *DFA*

minimalDFA()

Evaluates the equivalent minimal complete DFA

Returns equivalent minimal DFA

Return type *DFA*

product(*other*)

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

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

Return type *NFA*

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*

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*

reorder(*dicti*)

Reorder states indexes according to given dictionary.

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

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*

reverseTransitions(*rev*)

Evaluate reverse transition function.

Parameters **rev** (NFA) – 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) – an FA

Returns the resulting NFA

Return type *NFA*

star(*flag=False*)

Kleene star of a NFA

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

Returns the resulting NFA

Return type *NFA*

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 children states

Return type Set of int

stronglyConnectedComponents()

Strong components

Return type list

New in version 1.0.

subword()

returns a nfa that recognizes subword(L(self))

Return type nfa

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*

Note: valid to epsilon-NFA

toGFA()

Creates a GFA equivalent to NFA

Returns a GFA deep copy

Return type *GFA*

toNFA()

Dummy identity function

Return type *NFA*

toNFAr()

NFA with the reverse mapping of the delta function.

Returns shallow copy with reverse delta function added

Return type *NFAr*

uniqueRepr()

Dummy representation. Used DFA.uniqueRepr() :rtype: tuple

usefulStates(*initial_states=None*)

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

Parameters **initial_states** (`set of int or list of int`) – set of initial states

Returns set of state indexes

Return type set of int

static vDescription()

Generation of Verso interface description

New in version 0.9.5.

Returns the interface list

witness()

Witness of non emptyness

Returns word

Return type str

wordImage(*word, ist=None*)

Evaluates the set of states reached consuming given word

Parameters

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

Returns the set of ending states

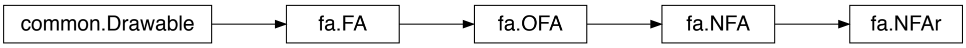
Return type Set of int

2.5 Class NFAr (Nondeterministic Finite Automata w/ reverse transition f.)

class `fa.`**`NFAr`**

Bases: *`fa.NFA`*

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



Variables **`deltaReverse`** – the reversed transition function

Note: 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`*) – 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 or list of int`*) – collection of int representing states

`elimEpsilonO`()

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

Returns itself

Return type

> **Attention:** performs inplace modification of automaton

homogenousP (*inplace=False*)
Checks is the automaton is homogenous, i.e.the transitions that reaches 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 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 (*tomerge*, *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

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

> **Note:** if target state is not given, the minimal index with be considered.

> **Attention:** The states of the list will become unreacheable, 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*

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 or 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 or None

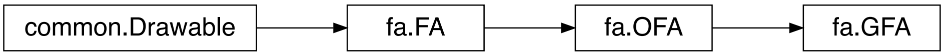
Note: if conditions aren't met, returned target state is None, and automaton remains unmodified.

2.6 Class GFA (Generalized Finite Automata)

class fa.**GFA**

Bases: *fa.OFA*

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 **DFAepsilonRedefenition** – if sym is Epsilon

assignLow (*st*)

Parameters **st** –

assignNum (*st*)

Parameters **st** –

completeDelta ()

Adds empty set transitions between the automatons 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 od states visited
- **io** –

dup ()

Returns a copy of a GFA

Return type *GFA*

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

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

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

2.7 Class SSemiGroup (Syntactic SemiGroup)

class fa.**SSemiGroup**

Bases: `object`

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

WordI (*i*)

Representative of an element given as index

Parameters **i** (*int*) – index of the element

Returns the first word originating the element

Return type str

WordPS (*pref*, *sym*)

Representative of an element given as prefix symb

Parameters

- **pref** (*int*) – prefix index
- **sym** (*int*) – symbol index

Returns word

Return type str

add (*tr*, *pref*, *sym*, *tmpLists*)

Try to add a new transformation to the monoid

Parameters

- **tr** (*tuple of int*) – transformation
- **pref** (*int or None*) – prefix of the generating word
- **sym** (*int*) – suffix symbol
- **tmpLists** (*pairs of lists as (elements,words)*) – this generation lists

addGen (*tmpLists*)

Add a new generation to the monoid

Parameters **tmpLists** (*pair of lists as (elements, words)*) – the new generation data

2.8 Class EnumL (Language Enumeration)

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

Bases: `object`

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

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 nth cross-section of L(A)

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

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

initStack ()
Abstract method

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

minWordT (*n*)
Abstract method :param int n: :type n: int

nextWord (*w*)
Abstract method :param w: :type w: str

2.9 Functions

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

`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

CHAPTER

THREE

MODULE: COMMON DEFINITIONS (COMMON)

Common definitions for FAdo files

3.1 Class Word

class `common.`**`Word`**(*data=None*, *it=None*)

Bases: `object`

Class to implement generic words as iterables with pretty-print

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

CHAPTER

FOUR

MODULE: FADO IO FUNCTIONS (`FIO`)

In/Out.

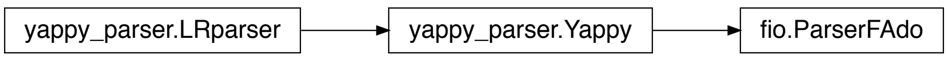
FAdo IO.

4.1 Class ParserFAdo (Yappy parser for FAdo FA files)

class `fio.`**`ParserFAdo`**(*no_table=1*, *table='.tableFAdo'*)

Bases: `yappy_parser.Yappy`

A parser for FAdo standard automata descriptions



4.2 Functions

`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, decrares 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.

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

CHAPTER

FIVE

MODULE: REGULAR EXPRESSIONS (REEX)

Regular expressions manipulation

Regular expression classes and manipulation

5.1 Class regexp (regular expression)

class `reex.regexp`(*sigma=None*)

Bases: `object`

Base class for regular expressions.

Variables **Sigma** – alphabet set of strings

alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression.

Return type integer

Attention: Doesn't include the empty word.

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

Compare with another regular expression for equivalence. :param r: :param cmp_method: :param nfa_method:

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

Compare with another regular expression for equivalence through minimal DFAs. :param r: :param nfa_method:

dfaAuPoint()

DFA "au-point" according to Nipkow

Returns "au-point" DFA

Return type *fa.DFA*

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

Returns word derivatives automaton

Return type *DFA*

See also:

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

dfaYMG()
DFA Yamada-McNaugthon-Gluskov according to Nipkow

Returns Y-M-G DFA

Return type *DFA*

See also:

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

static emptysetP()
Whether the regular expression is the empty set.

Return type Boolean

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

Return type integer

epsilonP()
Whether the regular expression is the empty word.

Return type Boolean

equivP(*r*)
Verifies if two regular expressions are equivalent.

Parameters **r** – regular expression

Return type boolean

equivalentP(*other*)
Tests equivalence

Parameters **other** –

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

Return type bool

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

Return type Boolean

first()

Return type set

last()

Return type set

linearForm()

Return type list

mark()

Make all atoms maked (tag False) :rtype: reex.regexp

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 *reex.regexp*

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, and construction.

Return type *NFA*

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** –

Returns NFA possibly with epsilon transitions

Return type NFAe

Note: The regular expression must be reduced

See also:

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

nfaGlushkov()

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

Returns NFA

nfaNaiveFollow()

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

Return type *NFA*

Note: Included for testing purposes.

See also:

Ilie & Yu (Follow Automata, 2003)

nfaPD()

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*

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.

Note: optimized version

Returns partial derivatives [or equation] automaton

Return type *NFA*

nfaPSNF()

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

Returns position automaton

Return type *NFA*

nfaPosition(*lstar=True*)

Position automaton of the regular expression.

Parameters **lstar** (*boolean*) – if not None followlists are computed dijunct

Returns position NFA

Return type *NFA*

rpn()

RPN representation :rtype: str :return: printable RPN representation

setOfSymbols()

Return type set

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

Set the alphabet for a regular expression and all its nodes

Parameters

- **symbolSet** (*list or set of str*) – 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()

starHeight()

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

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

Return type integer

toDFA()

DFA that accepts the regular expression's language

toNFA (*nfa_method='nfaPD'*)
NFA that accepts the regular expression's language. :param nfa_method:

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

Return type integer

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 of arbitrary symbols.

Return type regular expression

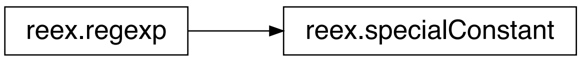
See also:

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

5.2 Class specialConstant

class reex.**specialConstant** (*sigma=None*)
Bases: *reex.regexp*

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 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

`partialDerivativesC` (*sigma*)

Parameters `sigma` –

Returns

`reversal` ()

Reversal of regexp

Return type *reex.regexp*

static `setOfSymbols` ()

Returns

`support` ()

Returns

`supportlast` ()

Returns

`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 epsilon() or the emptyset().

Return type (general) regular expression

`wordDerivative` (*word*)

Parameters `word` –

Returns

5.3 Class epsilon

class `reex.epsilon` (*sigma=None*)

Bases: `reex.specialConstant`

Class that represents the empty word.

Parameters **sigma** – alphabet

static **epsilonLength** ()

Return type int

static **epsilonP** ()

Return type bool

static **ewp** ()

Return type bool

static **measure** (*from_parent=None*)

Parameters **from_parent** –

Returns measures

nfaThompson ()

Return type *NFA*

static **partialDerivatives** (_)

Returns

partialDerivativesC (_)

Parameters **sigma** –

Returns

rpn ()

Returns str

snf (*_hollowdot=False*)

Parameters **_hollowdot** –

Returns

5.4 Class emptyset

class reex.**emptyset** (*sigma=None*)

Bases: *reex.specialConstant*

Class that represents the empty set.

Parameters **sigma** – alphabet

static **emptysetP**()

Returns

epsilonLength()

Returns

epsilonP()

Returns

ewp()

Returns

static **measure**(*from_parent=None*)

Parameters **from_parent** –

Returns

partialDerivativesC(_)

Parameters **sigma** –

Returns

rpn()

Returns

5.5 Class sigmaP

class reex.**sigmaP**(*sigma=None*)

Bases: *reex.specialConstant*

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

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

Parameters **sigma** – alphabet

derivative(*sigma*)

Parameters **sigma** –

Returns

ewp()

Returns

linearForm()

Returns

linearFormC()

Returns

partialDerivatives (*sigma*)

Parameters sigma –

Returns

static **partialDerivativesC** (_)

Parameters _ –

Returns

support ()

Returns

5.6 Class sigmaS

class reex.**sigmaS** (*sigma=None*)

Bases: *reex.specialConstant*

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

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

Parameters sigma – alphabet

derivative (*sigma*)

Parameters sigma –

Returns

ewp ()

Returns

linearForm ()

Returns

linearFormC ()

Returns

partialDerivatives (*sigma*)

Parameters sigma –

Returns

partialDerivativesC (*sigma*)

Parameters sigma –

Returns

support ()

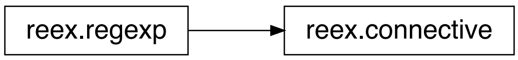
Returns

5.7 Class connective

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

Bases: *`reex.regexp`*

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

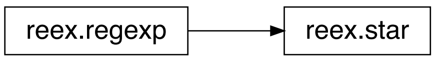


5.8 Class star

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

Bases: *`reex.regexp`*

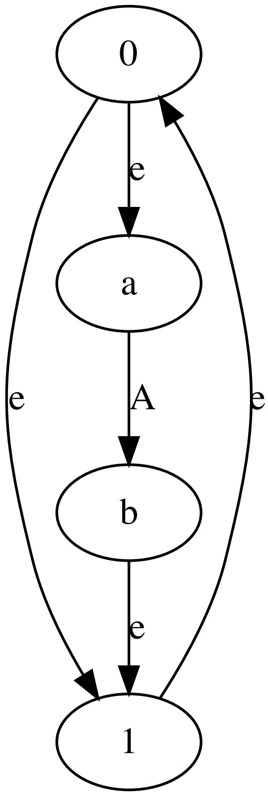
Class for iteration operation (aka Kleene star, or Kleene closure) on regular expressions.



`nfaThompson`()

Returns a NFA that accepts the RE.

Return type *NFA*



reversal()
Reversal of regexp

Return type *reex.regexp*

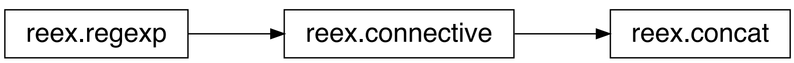
unmark()
Conversion back to regexp

Return type *reex.star*

5.9 Class concat

class reex.**concat**(*arg1*, *arg2*, *sigma=None*)
Bases: `reex.connective`

Class for catenation operation on regular expressions.



reversal()
Reversal of regexp

Return type *reex.regexp*

rpn()

Return type str

unmark()
Conversion back to unmarked atoms :rtype: concat

5.10 Class disj

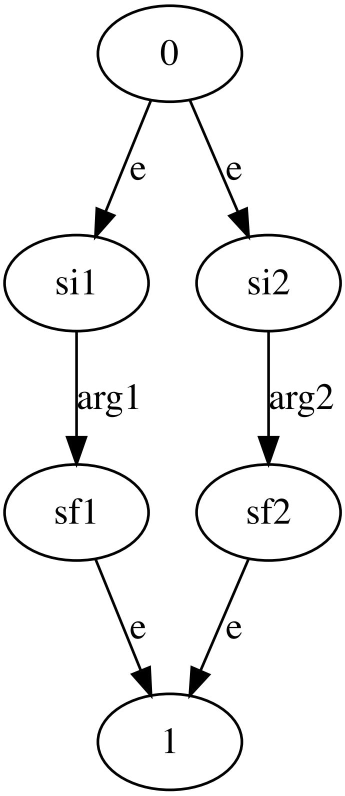
class reex.**disj**(*arg1*, *arg2*, *sigma=None*)
Bases: *reex.connective*

Class for disjuction operation on regular expressions.

mark()
Convertion to marked atoms :rtype: disj

nfaThompson()
Returns an NFA (Thompson) that accepts the RE.

Return type *NFA*



reversal()
Reversal of regexp

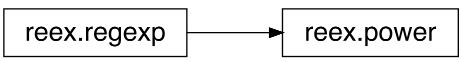
Return type *reex.regexp*

unmark()
Conversion back to unmarked atoms :rtype: disj

5.11 Class power

class reex.**power**(*arg*, *n=1*, *sigma=None*)
Bases: *reex.regexp*

Class for power operation on regular expressions.



reversal()
Reversal of regexp

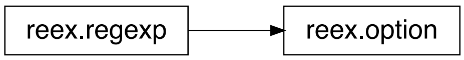
Return type *reex.regexp*

5.12 Class option

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

Bases: *reex.regexp*

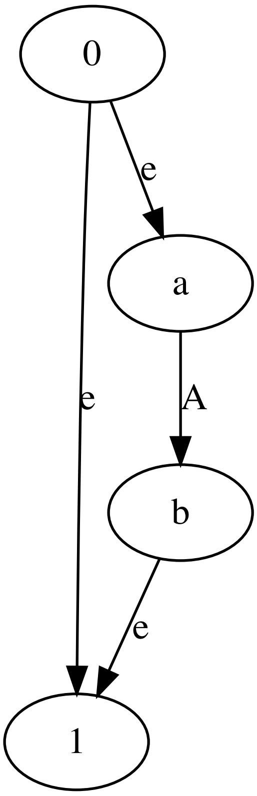
Class for option operation on regular expressions.



`nfaThompson()`

Returns a NFA that accepts the RE.

Return type *NFA*



`reversal()`

Reversal of regexp

Return type *reex.regexp*

5.13 Class conj (intersection)

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

Bases: *`reex.connective`*

Intersection operation of regexps

`support`()

5.14 Class shuffle

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

Bases: *`reex.connective`*

Shuffle operation of regexps

`support`()

`supportlast`()

5.15 Class atom

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

Bases: *`reex.regexp`*

Simple atom (symbol)

Variables

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

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 integer

Attention: Doesn't include the empty word.

derivative(*sigma*)
Derivative of the regular expression in relation to the given symbol.

Parameters **sigma** – an arbitrary symbol.

Return type regular expression

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:

10. (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 integer

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** –

Returns list of symbols

followLists(*lists=None*)
Map of each symbol's follow list in the regular expression.

Parameters **lists** –

Returns map of symbols' follow lists

Return type {symbol: list of symbols}

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 (Brüggemann-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** –

Returns map of symbols' follow lists

Return type {symbol: list of symbols}

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** –

Returns map of symbols' follow lists

Return type {symbol: list of symbols}

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** –

Returns list of symbols

Return type list

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 dictionary mapping heads to sets of tails

Return type {symbol: set([regular expressions])}

See also:

Antimirov, 95

linearFormC ()

Returns

linearP ()

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

Return type boolean

mark ()

Return type m_atom

static measure (*from_parent=None*)

A list with four measures for regular expressions.

Parameters **from_parent** –

Return type [int,int,int,int]

[alphabeticLength, treeLength, epsilonLength, starHeight]

1.alphabeticLength: number of occurences 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.

Return type *NFA*

See also:

11.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** – symbol in relation to which the derivative will be calculated.

Returns set of regular expressions

See also:

Antimirov, 95

partialDerivativesC(*sigma*)

Parameters **sigma** –

Returns

reduced(*hasEpsilon=False*)

Equivalent regular expression with the following cases simplified:

1.Epsilon.RE = RE.Epsilon = RE

2.EmptySet.RE = RE.EmptySet = EmptySet

3.EmptySet + RE = RE + EmptySet = RE

4.Epsilon + RE = RE + Epsilon = RE, where Epsilon is in L(RE)

5.RE** = RE*

6.EmptySet* = Epsilon* = Epsilon

7.Epsilon:RE = RE:Epsilon= RE

Parameters **hasEpsilon** – used internally to indicate that the language of which this term is a subterm has the empty word.

Returns regular expression

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

reversal()

Reversal of regexp

Return type *reex.regexp*

rpn()

RPN representation :return: printable RPN representation

setOfSymbols()

Set of symbols that occur in a regular expression..

Returns set of symbols

Return type set of symbols

snf (*hollowdot=False*)
Star Normal Form (SNF) of the regular expression.

Parameters **hollowdot** –

Returns regular expression in star normal form

static **starHeight** ()
Maximum level of nested regular expressions with a star operation applied.

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

Return type integer

stringLength ()
Length of the string representation of the regular expression.

Return type integer

support ()
'Support of a regular expression.

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)

supportlast ()
Subset of support such that elements have ewp

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

Return type integer

static **treeLength** ()
Number of nodes of the regular expression's syntactical tree.

Return type integer

unmarked ()
The unmarked form of the regular expression. Each leaf in its syntactical tree becomes a regexp(), the epsilon() or the emptyset().

Return type (general) regular expression

5.16 Class position

class reex.**position** (*val*, *sigma=None*)
Bases: *reex.atom*

Class for marked regular expression symbols.

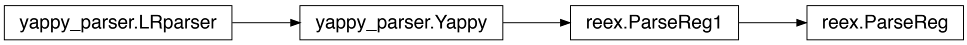
Constructor of a regular expression symbol.

Parameters **`val`** – the actual symbol

5.17 Class ParseReg

class `reex.`**`ParseReg`**(*no_table=1*, *table='tableambreg'*)

Bases: `reex.ParseReg1`



A parser for regular expressions with ambiguous rules: not working

5.18 Class sconnective (special connective)

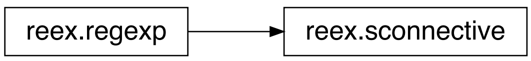
class `reex.`**`sconnective`**(*arg*, *sigma=None*)

Bases: *`reex.regexp`*

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

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

For parsing use str2sre



`alphabeticLength`()

Returns

`epsilonLength`()

Returns

`setOfSymbols`()

Returns

`syntacticLength`()

Returns

`treeLength`()

Returns

5.19 Class sconcat

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

Bases: `reex.sconnective`

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()

Returns

tail()

Returns

tail_rev()

Returns

5.20 Class sstar

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

Bases: *`reex.star`*

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

sstar: Class that represents Kleene star

`derivative`(*sigma*)

Parameters **`sigma`** –

Returns

`linearForm`()

Returns

`partialDerivatives`(*sigma*)

Parameters **`sigma`** –

Returns

`partialDerivativesC`(*sigma*)

Parameters **`sigma`** –

Returns

`support`()

Returns

5.21 Class sdisj

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

Bases: *`reex.sconnective`*

Class that represents the disjunction operation for special regular expressions.

`cross`(*ri*, *s*, *lists*)

Returns

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 ()

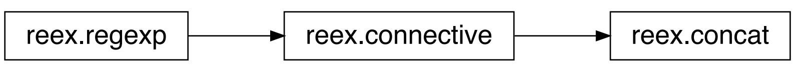
Returns

5.22 Class sconj

class reex.**sconj** (*arg, sigma=None*)

Bases: *reex.sconnective*

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 ()

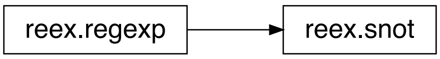
Returns

5.23 Class snot

class reex.**snot** (*arg*, *sigma=set([])*)

Bases: *reex.regexp*

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



alphabeticLength ()

Returns

derivative (*sigma*)
:param sigma :return:

epsilonLength ()

Returns

ewp ()

Returns

linearForm ()

Returns

linearFormC ()

Returns

partialDerivatives (*sigma*)

Parameters **sigma** –

Returns

partialDerivativesC (*sigma*)

Parameters **sigma** –

Returns

setOfSymbols ()

Returns

support ()

Returns

syntacticLength ()

Returns

treeLength ()

Returns

5.24 Functions

reex.**str2regexp** (*s*, *parser=<class 'reex.ParseReg1'>*, *no_table=1*, *sigma=None*, *strict=False*)

Reads a regexp from string.

Parameters

- **s** (*string*) – the string representation of the regular expression
- **parser** (*Yappy*) – a parser generator for regexps
- **no_table** (*integer*) – if 0 table is created
- **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*

reex.**str2sre** (*s*, *parser=<class 'reex.ParseS'>*, *no_table=1*, *sigma=None*, *strict=False*)

Reads a sre from string. Arguments as str2regexp.

Return type *regexp*

reex.**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** (*string*) – RPN representation

Return type *reex.regexp*

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

CHAPTER

SIX

MODULE: TRANSDUCERS (TRANSDUCERS)

Finite Tranducer Support

Transducer manipulation.

New in version 1.0.

6.1 Class Transducer

class `transducers.`**`Transducer`**

Bases: *`fa.NFA`*

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 tupples

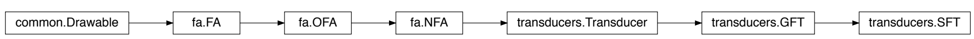
6.2 Class SFT (Standard Form Transducers)

class `transducers.`**`SFT`**

Bases: `transducers.GFT`

Standard Form Tranducer

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



addEpsilonLoops ()
Add a loop transition with epsilon input and output to every state in the 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*, *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) – the second transducer

Return type *SFT*

concat (*other*)
Concatenation of transducers

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

Return type *SFT*

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** – 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 meaningfull state names.

Note: The resulting transducer is not trim.

Parameters **other** (*DFA | NFA*) – the automata needs to be operated.

Return type *SFT*

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 | NFA*) – the automata needs to be operated.

Return type *SFT*

inverse()
Switch the input label with the output label.

No initial or final state changed.

Returns Transducer with transitions switched.

Return type *SFT*

nonEmptyW()
Witness of non emptyness

Returns pair (in-word, out-word)

Return type tuple

nonFunctionalW()
Returns a witness of non funcionality (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 | NFA*) – the automaton used as a filter of the output

Return type *SFT*

outIntersectionDerived(*other*)
Naive version of outIntersection

Parameters **other** (*DFA | NFA*) – the automaton used as a filter of the output

Return type *SFT*

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) – the automaton used as filter

Return type *SFT*

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 meaningfull names of states.

Parameters **other** (NFA) – the automaton used as filter

Return type *SFT*

reversal()
Returns a transducer that recognizes the reversal of the relation.

Returns Transducer recognizing reversal language

Return type *SFT*

runOnNFA(*nfa*)
Result of applying a transducer to an automaton

Parameters **nfa** (*DFA|NFA*) – input language to transducer

Returns resulting language

Return type *NFA*

runOnWord(*word*)
Returns the automaton accepting the outup of the transducer on the input word

Parameters **word** – the word

Return type *NFA*

setInitial(*sts*)
Sets the initial state of a Transducer

Parameters **sts** (*list*) – list of states

square()
Conjunction of transducer with itself

Return type *NFA*

square_fv()
Conjunction of transducer with itself (Fast Version)

Return type *NFA*

star (*flag=False*)
Kleene star

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

Returns the resulting Transducer

Return type *SFT*

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

Return type *NFA*

toNFT ()
Transformation into Nomal Form Transducer

Return type NFT

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*

toSFT ()
Pacifying rule

Return type *SFT*

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) – the other operand

Return type *SFT*

6.3 Functions

CHAPTER

SEVEN

MODULE: FINITE LANGUAGES (FL)

Finite languages and related automata manipulation

Finite languages manipulation

7.1 Class FL (Finite Language)

class `fl.FL`(*wordsList=None*, *Sigma=None*)

Bases: `object`

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*

`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`) – right hand operand

Return type *FL*

Raises **`FAdoGeneralError`** – 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 | NFA*) – 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) – right hand operand

Raises **FAdoGeneralError** – if both arguments are not FL

multiLineAutomaton ()
Generates the trivial linear ANFA equivalent to this language

Return type *ANFA*

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*

New in version 1.2.

toNFA ()
Generates a NFA recognizing the language

Return type *ANFA*

New in version 1.2.

trieFA ()
Generates the trie automaton that recognises this language

Returns the trie automaton

Return type *ADFA*

union (*other*)
union of FL: a | b

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

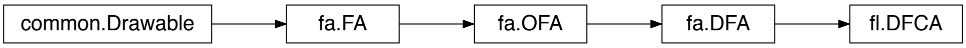
Return type *FL*

Raises **FAdoGeneralError** – if both arguments are not FL

7.2 Class DFCA (Deterministic Finite Cover Automata)

class fl.**DFCA**
Bases: *fa.DFA*

Deterministic Cover Automata class



length

Returns size of the longest word

Return type int

7.3 Class AFA (Acyclic Finite Automata)

class fl.**AFA**

Bases: object

Base class for Acyclic Finite Automata

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

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 accessability 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

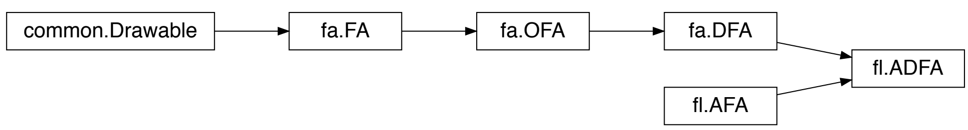
---

7.4 Class ADFA (Acyclic Deterministic Finite Automata)

class fl.**ADFA**

Bases: *fa.DFA*, *fl.AFA*

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) – 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*

Attention: The object is modified in place

Changed in version 1.3.3.

`diss()`
Evaluates the dissimilarity language

Return type *FL*

New in version 1.2.1.

`dissMin(witnesses=None)`
Evaluates the minimal dissimilarity language :param dict witnesses: optional witness dictionay :rtype: FL

New in version 1.2.1.

`dup()`
Duplicate the basic structure into a new ADFA. Basically a copy.deep.

Return type *ADFA*

`forceToDFA()`
Conversion to DFA

Return type *DFA*

`forceToDFCA()`
Conversion to DFCA

Return type *DFA*

`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*

New in version 0.9.8.

See also:

Cezar Campeanu, Andrei Päun, 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*

`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*

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*

toNFA ()
Converts the ADFA in a equivalent NFA

Return type *ANFA*

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

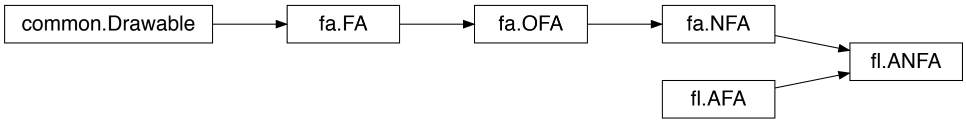
Return type *RndWGen*

New in version 1.2.

7.5 Class ANFA (Acyclic Non-deterministic Finite Automata)

class fl.**ANFA**
Bases: *fa.NFA*, *fl.AFA*

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 teh automaton

> **Attention:** the object is modified in place

moveFinal (*st*, *stf*)

Unsets a set as final transfering 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

7.6 Class RndWGen (Random Word Generator)

class fl.**RndWGen** (*aut*)

Bases: object

Word random generator class

New in version 1.2.

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

next()
Next word

Returns a new random word

7.7 Functions

fl.**sigmaInitialSegment**(*Sigma*, *l*, *exact=False*)
Generates the ADFA recognizing Sigma^i for i<=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

fl.**genRndTrieBalanced**(*maxL*, *Sigma*, *safe=True*)
Generates a random trie automaton for a binary language of balanced words of a given leght 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

fl.**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*

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

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

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*

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.

CHAPTER

EIGHT

MODULE: GRAPHS (GRAPH CREATION AND MANIPULATION)

Graph support

Basic Graph object support and manipulation

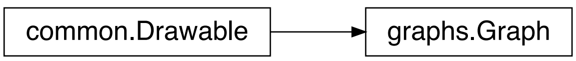
class `graphs.`**`Graph`**

Bases: `common.Drawable`

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`** – if vname already exists

`vertexIndex`(*vname*, *autoCreate=False*)

Return vertex index

Parameters

- **`autoCreate`** (*`bool`*) – auto creation of non existing states
- **`vname`** – vertex name

Return type int

Raises **`GraphError`** – if vname not found

class `graphs.`**`DiGraph`**

Bases: *`graphs.Graph`*

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.

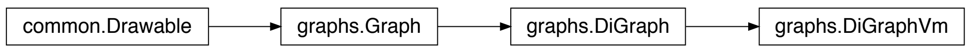
inverse ()
Inverse of a digraph

class graphs.**DiGraphVm**

Bases: *graphs.DiGraph*

Directed graph with marked vertices

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



markVertex(*v*)

Mark vertex v

Parameters **v** (*int*) – vertex

CHAPTER

NINE

MODULE: CONTEXT FREE GRAMMARS MANIPULATION (CFG)

Context Free Grammars Manipulation.

Basic context-free grammars manipulation for building uniform random generetors

9.1 Class CFGrammar (Context Free Grammar)

class `cfg.`**`CFGrammar`**(*gram*)

Bases: `object`

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 ->* []

`makenonterminals`()

Extracts C{nonterminals} from grammar rules.

`maketerminals`()

Extracts C{terminals} from the rules. Nonterminals must already exist

9.2 Class CNF

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

Bases: *`cfg.CFGrammar`*

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.

`Chomsky`()

Transform to CNF

elim_unitary()
Elimination of unitary rules

9.3 Class cfgGenerator

class cfg.**cfgGenerator**(*cfgr*, *size*)
Bases: object

CFG uniform genetaror

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

9.4 Class reStringRGenerator (Reg Exp Generator)

class cfg.**reStringRGenerator**(*Sigma=None*, *size=10*, *cfgr=None*, *epsilon=None*, *empty=None*, *ident='Ti'*)
Bases: *cfg.cfgGenerator*

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 arbitary 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

9.5 Functions

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

`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

CHAPTER

TEN

MODULE: RANDOM DFA GENERATOR (RNDFA)

Random DFA generation

ICDFA Random generation binding

Changed in version 0.9.4: Interface python to the C code

Changed in version 0.9.6: Working with incomplete automata

Changed in version 0.9.8: distinct classes for complete and incomplete ICDFA

10.1 Class ICDFArgen (Generator container)

class `rndfa.`**`ICDFArgen`**

Bases: `object`

Generic ICDFA random generator class

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

`next``()`

Get the next generated DFA

Returns a random generated ICDFA

Return type *DFA*

10.2 Class ICDFArnd (Complete ICDFA random generator)

class `rndfa.`**`ICDFArnd`**`(`*n*, *k*, *seed=0*`)`

Bases: *`rndfa.ICDFArgen`*

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 the 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

10.3 Class ICDFArndIncomple (Incomplete ICDFA generator)

class `rndfa.`**`ICDFArndIncomplete`**(*n*, *k*, *bias=None*, *seed=0*)

Bases: *`rndfa.ICDFArgen`*

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`** – if a bias >=1 or <=0 is provided

Changed in version 1.3.4: seed added to the random generator

CHAPTER

ELEVEN

MODULE: RANDOM ADFA GENERATOR (RNDADFA)

Random ADFA generation

ADFA Random generation binding

New in version 1.2.1.

11.1 Class ADFArnd (ADFA random generator)

class 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 ICDFA 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 souces

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 souces

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 souces
- **u** (*int*) – number of souces 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 souces

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

next ()
Generates a random adfa

Returns an dfa if number of sources is 1; otherwise self.transitions has the transitions of an adfa with s sources

Return type *DFA*

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*)

Uniformaly 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

CHAPTER

TWELVE

MODULE: COMBO OPERATIONS (COMBOPERATIONS)

Several combined operations for DFAs

Combined operations

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

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

Parameters

- **f a1** (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.

`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.

`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

`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.

`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.

`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*

`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.

`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.

CHAPTER

THIRTEEN

MODULE: CODES (CODES)

Code theory module

New in version 1.0.

13.1 Class CodeProperty

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

Bases: `object`

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.

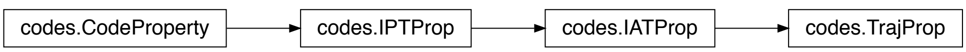
Variables **`Sigma`** – the alphabet

13.2 Class TrajProp

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

Bases: *`codes.IATProp`*

Class of trajectoty properties



Constructor

Parameters

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

static **`trajToTransducer`**(*traj*, *Sigma*)

Input Altering Tranducer corresponding to a Trajectory

Parameters

- **`traj`** (`NFA`) – trajectory language
- **`Sigma`** (*`set`*) – alphabet

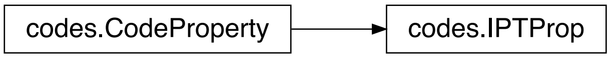
Return type *SFT*

13.3 Class IPTProp

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

Bases: *codes.CodeProperty*

Input Preserving Transducer Property



Variables

- **`Aut`** (SFT) – 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) – 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,...,s-1}. where s <= 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,...,s-1}. where s <= 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) – 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) – the automaton
- **U** (NFA) – Universe of permitted words (Sigma^* 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) – 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 | NFA*) – the automaton
- **U** (*DFA | NFA*) – Universe of permitted words (Sigma^* as default)

Return type bool

Raises PropertyNotSatisfied – if not satisfied

notSatisfiesW (*aut*)
Return a witness of non-satisfaction of the property by the automaton language

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

Returns word witness pair

Return type tuple

satisfiesP (*aut*)
Satisfaction of the property by the automaton language

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

Return type bool

13.4 Class IATProp

class codes.**IATProp** (*aut*, *name=None*)
Bases: *codes.IPTProp*

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 | NFA*) – the automaton

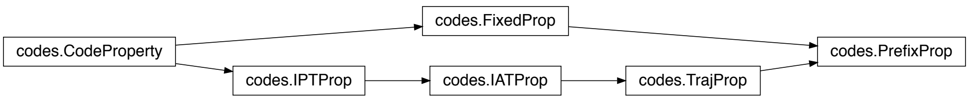
Returns word witness pair

Return type tuple

13.5 Class PrefixProp

class codes.**PrefixProp**(*t*)
Bases: *codes.TrajProp*, codes.FixedProp

Prefix Property



satisfiesPrefixP(*aut*)
Satisfaction of property by the automaton language: faster than satisfiesP

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

Return type bool

13.6 Class ErrDetectProp

codes.**ErrDetectProp**
alias of *IPTProp*

13.7 Class ErrCorrectProp

class codes.**ErrCorrectProp**(*t*)
Bases: *codes.IPTProp*

Error Correcting Property

notMaximalW(*aut*, *U=None*)
Tests if the language is maximal w.r.t. the property

Parameters

- **aut** (*DFA | NFA*) – the automaton
- **U** (*DFA | NFA*) – Universe of permitted words (Sigma^* as default)

Return type bool

notSatisfiesW(*aut*)
Satisfaction of the code property by the automaton language

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

Return type bool

13.8 Functions

codes.**buildTrajPropS**(*regex*, *sigma*)
Builds a TrajProp from a string regexp

Parameters

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

Return type *TrajProp*

codes.**buildIATPropF**(*fname*)
Builds a IATProp from a FAdo SFT file

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

Return type *IATProp*

codes.**buildIPTPropF**(*fname*)
Builds a IPTProp from a FAdo SFT file

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

Return type *IPTProp*

`codes.`**`buildIATPropS`**`(`*s*`)`
Builds a IATProp from a FAdo SFT string

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

Return type *IATProp*

`codes.`**`buildIPTPropS`**`(`*s*`)`
Builds a IPTProp from a FAdo SFT string

Parameters **`s`** (`str`) – file name

Return type *IPTProp*

`codes.`**`buildErrorDetectPropF`**`(`*fname*`)`
Builds an Error Detecting Property

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

Return type *ErrDetectProp*

`codes.`**`buildErrorCorrectPropF`**`(`*fname*`)`
Builds an Error Correcting Property

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

Return type *ErrCorrectProp*

`codes.`**`buildErrorDetectPropS`**`(`*s*`)`
Builds an Error Detecting Property from string

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

Return type *ErrDetectProp*

`codes.`**`buildErrorCorrectPropS`**`(`*s*`)`
Builds an Error Correcting Property from string

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

Return type *ErrCorrectProp*

`codes.`**`buildPrefixProperty`**`(`*alphabet*`)`
Builds a Prefix Code Property

Parameters **`alphabet`** (`set`) – alphabet

Return type *PrefixProp*

`codes.`**`buildTrajPropS`**`(`*regex, sigma*`)`
Builds a TrajProp from a string regexp

Parameters

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

Return type *TrajProp*

`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

`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

`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*

CHAPTER

FOURTEEN

MODULE: GRAIL COMPATIBILITY (GRAIL)

GRAIL support.

GRAIL formats support. This is an auxiliary module that sould be imported by fa.py

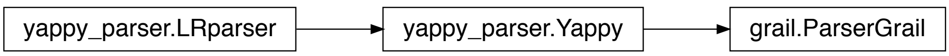
New in version 0.9.4.

14.1 Class ParserGrail

class `grail.`**`ParserGrail`**(*no_table=1*, *table='.tableGrail'*)

Bases: `yappy_parser.Yappy`

A parser form GRAIL standard automata descriptions



14.2 Class Grail

class `grail.`**`Grail`**

Bases: `object`

A class for Grail execution

Changed in version 0.9.8: tries to initialise execPath from fadorc

`do`(*cmd*, **args*)

Execute Grail command

Parameters

- **`cmd`** (*`string`*) – name of the command
- **`args`** – arguments

Raises

- **`GrailCommandError`** – if the syntax is not correct an exception is raised
- **`FAdoGeneralError`** – if Grail fails to execute something

`setExecPath`(*path*)

Sets the path to the Grail executables

Parameters **`path`** (*`str`*) – the path to Grail executables

14.3 Functions

`grail.`**`exportToGrail`**(*fileName*, *fa*)

Saves a finite automatom definition to a file using Grail format

Parameters

- **fileName** (*string*) – file name
- **fa** (FA) – the FA

`grail.`**`FAToGrail`**(*f*, *fa*)

Saves a finite automatom definition to an open file using Grail format

Parameters

- **f** (*file*) – opended file
- **fa** (FA) – the FA

`grail.`**`importFromGrailFile`**(*fileName*)

Imports a finite automaton from a file in GRAIL format

The type of the object returned depends on the transition definiion red as well as the number of initial states declared

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

Returns the automata red

Return type *FA*

`grail.`**`FAFromGrail`**(*buffer*)

Imports a finite automaton from a buffer in GRAIL format

The type of the object returned depends on the transition definiion red as well as the number of initial states declared

Parameters **buffer** (*str*) – buffer file

Returns the automata red

Return type *FA*

CHAPTER

FIFTEEN

MODULE: VERSO LANGUAGE (VERSO)

FAdo interface language and slave manager

Applications that want to use FAdo as a slave, just to process it objects should use this language to interface with it.

Note: Every object that is supposed to be available through this language, should be defined in objects and should have a method `vDescription`, returning the following list

0. A pair of descriptions, short and long, of the object
1. A list of pairs

1.0. A name of a format *(names should be unique)*

1.1. The function that returns the string representation of the object in that format

2. A tuple for each method provided

2.0. Name of the command in verso

2.1. A pair, short/long, descriptions of the method

2.2. Number (n) of arguments of the method

2.2+i. The type of the ith argument

2.1+n. The return type `None` if does not return (in place transformation)

2.2+n. The function implementing the method having a list as arguments

3. and so on...

---

class `verso.`**`ParserVerso`**(*vsession*, *objects=None*, *no_table=0*, *table='.tableVerso'*)

Bases: `yappy_parser.Yappy`

A parser for FAdo standard automata descriptions

Variables

- **`vi`** – virtual interaction session that knows how to communicate with the client
- **`objects`** – the list of objects known
- **`info`** – dictionary object -> (longdescription, [list of commands])
- **`fun`** – dictionary command -> (arity, return type, function)
- **`format`** – dictionary formatName -> function

Parameters

- **`no_table`** – ignore the table if it exists
- **`table`** – name of the table

CHAPTER

SIXTEEN

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')
>>> 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 (@epmtyset), 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(regexp("a"),concat(regexp("b"),regexp("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) , a 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) 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

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

c

cfg, 91
codes, 105
comboperations, 101
common, 43

f

fa, 5
fio, 45
fl, 79

g

grail, 113
graphs, 87

r

reex, 47
rndadfa, 97
rndfa, 95

t

transducers, 73

v

verso, 115

INDEX

A

acyclicP() (fa.OFA method), 11
add() (fa.SSemiGroup method), 39
addEdge() (graphs.DiGraph method), 88
addEdge() (graphs.Graph method), 87
addEpsilonLoops() (fa.NFA method), 27
addEpsilonLoops() (transducers.SFT method), 73
addFinal() (fa.FA method), 5
addGen() (fa.SSemiGroup method), 39
addInitial() (fa.NFA method), 28
addOutput() (transducers.SFT method), 74
addSigma() (fa.FA method), 5
addState() (fa.FA method), 5
addState() (fl.AFA method), 81
addSuffix() (fl.ADFA method), 82
addToCode() (codes.IPTProp method), 106
addTransition() (fa.DFA method), 15
addTransition() (fa.GFA method), 37
addTransition() (fa.NFA method), 28
addTransition() (fa.NFAr method), 35
addTransition() (fa.OFA method), 11
addTransition() (transducers.SFT method), 74
addTransitionProductQ() (transducers.SFT method), 74
addTransitionQ() (fa.NFA method), 28
addTransitionQ() (transducers.SFT method), 74
addVertex() (graphs.Graph method), 87
addWord() (fl.FL method), 79
addWords() (fl.FL method), 79
ADFA (class in fl), 82
ADFArnd (class in rndadfa), 97
aEquiv() (fa.DFA method), 15
AFA (class in fl), 81
allRegExps() (fa.OFA method), 11
alpha() (rndadfa.ADFArnd method), 97
alpha0() (rndadfa.ADFArnd method), 97
alphabeticLength() (reex.atom static method), 61
alphabeticLength() (reex.regexp method), 47
alphabeticLength() (reex.sconnective method), 66
alphabeticLength() (reex.snot method), 70
alphabeticLength() (reex.specialConstant static method), 51
ANFA (class in fl), 84
assignLow() (fa.GFA method), 37
assignNum() (fa.GFA method), 37
atom (class in reex), 61
autobisimulation() (fa.NFA method), 28
autobisimulation2() (fa.NFA method), 28

B

beta() (rndadfa.ADFArnd method), 98
beta0() (rndadfa.ADFArnd method), 98
buildErrorCorrectPropF() (in module codes), 110
buildErrorCorrectPropS() (in module codes), 110
buildErrorDetectPropF() (in module codes), 110
buildErrorDetectPropS() (in module codes), 110
buildIATPropF() (in module codes), 109
buildIATPropS() (in module codes), 109
buildIPTPropF() (in module codes), 109
buildIPTPropS() (in module codes), 110
buildPrefixProperty() (in module codes), 110
buildTrajPropS() (in module codes), 109, 110

C

cfg (module), 91
cfgGenerator (class in cfg), 92
CFGrammar (class in cfg), 91
Chomsky() (cfg.CNF method), 91
closeEpsilon() (fa.NFA method), 28
CNF (class in cfg), 91
CodeProperty (class in codes), 105
codes (module), 105
comboperations (module), 101
common (module), 43
compare() (reex.regexp method), 47
compareMinimalDFA() (reex.regexp method), 47
compat() (fa.DFA method), 15
complete() (fa.DFA method), 16
complete() (fl.ADFA method), 82
completeDelta() (fa.GFA method), 37
completeMinimal() (fa.DFA method), 16
completeP() (fa.DFA method), 16
completeProduct() (fa.DFA method), 16
composition() (transducers.SFT method), 74
computeFollowNames() (fa.NFA method), 28
computeKernel() (fa.DFA method), 16
concat (class in reex), 57
concat() (fa.DFA method), 16
concat() (fa.NFA method), 28
concat() (transducers.SFT method), 74
concatI() (fa.DFA method), 16
concatWStar() (in module comboperations), 101
conj (class in reex), 61

conjunction() (fa.FA method), 6
connective (class in reex), 56
countAdfaBySources() (rndadfa.ADFArnd method), 98
countTransitions() (fa.FA method), 6
countTransitions() (fa.NFA method), 29
createInputAlteringSIDTrans() (in module codes), 111
cross() (reex.sdisj method), 68
cutPoints() (fa.OFA method), 12

D

deleteState() (fa.FA method), 6
deleteState() (fa.GFA method), 37
deleteState() (transducers.SFT method), 75
deleteStates() (fa.DFA method), 17
deleteStates() (fa.NFA method), 29
deleteStates() (fa.NFAr method), 35
deleteStates() (fa.OFA method), 12
deleteStates() (transducers.SFT method), 75
delFinal() (fa.FA method), 6
delFinals() (fa.FA method), 6
Delta() (fa.DFA method), 15
delTransition() (fa.DFA method), 17
delTransition() (fa.NFA method), 29
delTransition() (fa.NFAr method), 35
delTransition() (transducers.SFT method), 74
derivative() (reex.atom method), 62
derivative() (reex.sconcat method), 67
derivative() (reex.sconj method), 69
derivative() (reex.sdisj method), 68
derivative() (reex.sigmaP method), 54
derivative() (reex.sigmaS method), 55
derivative() (reex.snot method), 70
derivative() (reex.specialConstant method), 51
derivative() (reex.sstar method), 68
deterministicP() (fa.NFA method), 29
detSet() (fa.NFA method), 29
DFA (class in fa), 15
dfaAuPoint() (reex.regexp method), 47
dfaBrzozowski() (reex.regexp method), 48
DFAtoADFA() (in module fl), 86
dfaYMG() (reex.regexp method), 48
DFCA (class in fl), 80
DFS() (fa.GFA method), 37
dfs_visit() (fa.GFA method), 37
diff() (fl.FL method), 79
DiGraph (class in graphs), 87
DiGraphVm (class in graphs), 88
directRank() (fl.AFA method), 81
disj (class in reex), 58
disj() (fa.FA method), 6
disjunction() (fa.FA method), 6
disjWStar() (in module comboperations), 102
diss() (fl.ADFA method), 83
dissMin() (fl.ADFA method), 83
dist() (fa.DFA method), 17
distDerivative() (reex.specialConstant method), 51
distMin() (fa.DFA method), 17
distR() (fa.DFA method), 18
distRMin() (fa.DFA method), 18
distTS() (fa.DFA method), 18
do() (grail.Grail method), 113
dotDrawEdge() (graphs.DiGraph static method), 88
dotDrawState() (fa.FA method), 6
dotDrawState() (fa.SemiDFA method), 10
dotDrawTransition() (fa.FA method), 6
dotDrawTransition() (fa.OFA static method), 12
dotDrawTransition() (fa.SemiDFA static method), 10
dotDrawVertex() (graphs.DiGraph method), 88
dotFormat() (fa.FA method), 7
dotFormat() (fa.NFA method), 29
dotFormat() (fa.SemiDFA method), 11
dotFormat() (graphs.DiGraph method), 88
dump() (fa.OFA method), 12
dup() (fa.DFA method), 18
dup() (fa.GFA method), 38
dup() (fa.NFA method), 30
dup() (fa.OFA method), 12
dup() (fl.ADFA method), 83
dup() (transducers.SFT method), 75

E

editDistanceW() (in module codes), 110
elim_unitary() (cfg.CNF method), 91
elimEpsilon() (fa.NFA method), 30
elimEpsilonO() (fa.NFAr method), 35
eliminate() (fa.GFA method), 38
eliminateAll() (fa.GFA method), 38
eliminateDeadName() (fa.FA method), 7
eliminateEpsilonTransitions() (fa.NFA method), 30
eliminateSingles() (fa.OFA method), 12
eliminateState() (fa.GFA method), 38
eliminateStout() (fa.OFA method), 12
eliminateTSymbol() (fa.NFA method), 30
emptyP() (fa.OFA method), 12
emptyP() (transducers.SFT method), 75
emptyset (class in reex), 53
emptysetP() (reex.emptyset static method), 54
emptysetP() (reex.regexp static method), 48
ensureDead() (fl.AFA method), 81
enum() (fa.EnumL method), 40
enumCrossSection() (fa.EnumL method), 40
enumDFA() (fa.DFA method), 18
EnumL (class in fa), 39
enumNFA() (fa.NFA method), 30
epsilon (class in reex), 52
epsilonClosure() (fa.NFA method), 30
epsilonLength() (reex.atom static method), 62
epsilonLength() (reex.emptyset method), 54
epsilonLength() (reex.epsilon static method), 53
epsilonLength() (reex.regexp method), 48
epsilonLength() (reex.sconnective method), 66
epsilonLength() (reex.snot method), 70
epsilonOutP() (transducers.SFT method), 75
epsilonP() (fa.NFA method), 30
epsilonP() (reex.emptyset method), 54
epsilonP() (reex.epsilon static method), 53

epsilonP() (reex.regexp method), 48
epsilonP() (transducers.SFT method), 75
epsilonPaths() (fa.NFA method), 30
equal() (fa.DFA method), 18
equivalentP() (fa.FA method), 7
equivalentP() (reex.regexp method), 48
equivP() (reex.regexp method), 48
equivReduced() (fa.NFA method), 31
ErrCorrectProp (class in codes), 108
ErrDetectProp (in module codes), 108
evalNumberOfStateCycles() (fa.OFA method), 12
evalRank() (fl.AFA method), 81
evalSymbol() (fa.DFA method), 18
evalSymbol() (fa.FA method), 7
evalSymbol() (fa.NFA method), 31
evalSymbol() (fa.OFA method), 13
evalSymbolI() (fa.DFA method), 18
evalSymbolL() (fa.DFA method), 19
evalSymbolLI() (fa.DFA method), 19
evalWord() (fa.DFA method), 19
evalWordP() (fa.DFA method), 19
evalWordP() (fa.NFA method), 31
evalWordP() (reex.regexp method), 48
evalWordP() (transducers.SFT method), 75
evalWordSlowP() (transducers.SFT method), 75
ewp() (reex.emptyset method), 54
ewp() (reex.epsilon static method), 53
ewp() (reex.regexp method), 48
ewp() (reex.sconcat method), 67
ewp() (reex.sconj method), 70
ewp() (reex.sdisj method), 69
ewp() (reex.sigmaP method), 54
ewp() (reex.sigmaS method), 55
ewp() (reex.snot method), 70
exponentialDensityP() (in module codes), 111
exportToGrail() (in module grail), 114

F

FA (class in fa), 5
fa (module), 5
FAFromGrail() (in module grail), 114
FAToGrail() (in module grail), 114
fillStack() (fa.EnumL method), 40
filter() (fl.FL method), 79
finalCompP() (fa.DFA method), 19
finalCompP() (fa.NFA method), 31
finalCompP() (fa.OFA method), 13
finalP() (fa.FA method), 7
finalsP() (fa.FA method), 7
fio (module), 45
first() (reex.atom method), 62
first() (reex.regexp method), 48
first() (reex.sdisj method), 69
first() (reex.specialConstant static method), 51
FL (class in fl), 79
fl (module), 79
followFromPosition() (fa.NFA method), 31
followLists() (reex.atom method), 62
followLists() (reex.sdisj method), 69
followLists() (reex.specialConstant method), 51
followListsD() (reex.atom method), 62
followListsD() (reex.specialConstant method), 52
followListsStar() (reex.atom method), 63
followListsStar() (reex.sdisj method), 69
followListsStar() (reex.specialConstant static method), 52
forceToDFA() (fl.ADFA method), 83
forceToDFCA() (fl.ADFA method), 83
functionalP() (transducers.SFT method), 75

G

gamma() (rndadfa.ADFArnd method), 98
generate() (cfg.cfgGenerator method), 92
genRandomTrie() (in module fl), 86
genRndTrieBalanced() (in module fl), 86
genRndTriePrefix() (in module fl), 86
genRndTrieUnbalanced() (in module fl), 86
getLeaves() (fl.AFA method), 81
GFA (class in fa), 37
Grail (class in grail), 113
grail (module), 113
Graph (class in graphs), 87
graphs (module), 87
gRules() (in module cfg), 92

H

half() (fa.NFA method), 31
hasStateIndexP() (fa.FA method), 7
hasTransitionP() (fa.NFA method), 31
hasTrapStateP() (fa.DFA method), 20
head() (reex.sconcat method), 67
head_rev() (reex.sconcat method), 67
homogeneousFinalityP() (fa.NFA method), 31
homogenousP() (fa.NFA method), 31
homogenousP() (fa.NFAr method), 36
hyperMinimal() (fa.DFA method), 20

I

IATProp (class in codes), 107
ICDFArgen (class in rndfa), 95
ICDFArnd (class in rndfa), 95
ICDFArndIncomplete (class in rndfa), 96
iCompleteP() (fa.EnumL method), 40
importFromGrailFile() (in module grail), 114
inDegree() (fa.DFA method), 20
indexList() (fa.FA method), 7
infix() (fa.DFA method), 20
inIntersection() (transducers.SFT method), 76
inIntersectionSlow() (transducers.SFT method), 76
initialComp() (fa.DFA method), 20
initialComp() (fa.NFA method), 32
initialComp() (fa.OFA method), 13
initialP() (fa.DFA method), 20
initialP() (fa.FA method), 8
initialSet() (fa.DFA method), 20
initialSet() (fa.FA method), 8

initStack() (fa.EnumL method), 40
inputS() (fa.FA method), 8
intersection() (fl.FL method), 80
interWStar() (in module comboperations), 102
inverse() (graphs.DiGraph method), 88
inverse() (transducers.SFT method), 76
IPTProp (class in codes), 106

J

joinStates() (fa.DFA method), 20

L

last() (reex.atom method), 63
last() (reex.regexp method), 48
last() (reex.sdisj method), 69
last() (reex.specialConstant method), 52
length (fl.DFCA attribute), 81
lEquivNFA() (fa.NFA method), 32
level() (fl.ADFA method), 83
linearForm() (reex.atom method), 63
linearForm() (reex.regexp method), 49
linearForm() (reex.sconcat method), 67
linearForm() (reex.sconj method), 70
linearForm() (reex.sdisj method), 69
linearForm() (reex.sigmaP method), 54
linearForm() (reex.sigmaS method), 55
linearForm() (reex.snot method), 70
linearForm() (reex.specialConstant method), 52
linearForm() (reex.sstar method), 68
linearFormC() (reex.atom method), 63
linearFormC() (reex.sconcat method), 67
linearFormC() (reex.sdisj method), 69
linearFormC() (reex.sigmaP method), 54
linearFormC() (reex.sigmaS method), 55
linearFormC() (reex.snot method), 70
linearP() (reex.atom method), 63
lrEquivNFA() (fa.NFA method), 32

M

MADFA() (fl.FL method), 79
makeCode() (codes.IPTProp method), 106
makeCodeO() (codes.IPTProp method), 106
makenonterminals() (cfg.CFGrammar method), 91
makeReversible() (fa.DFA method), 20
maketerminals() (cfg.CFGrammar method), 91
mark() (reex.atom method), 63
mark() (reex.disj method), 58
mark() (reex.regexp method), 49
marked() (reex.regexp method), 49
markNonEquivalent() (fa.DFA method), 21
markVertex() (graphs.DiGraphVm method), 89
maximalP() (codes.IPTProp method), 107
measure() (reex.atom static method), 63
measure() (reex.emptyset static method), 54
measure() (reex.epsilon static method), 53
mergeInitial() (fl.ANFA method), 85
mergeLeaves() (fl.ANFA method), 85
mergeStates() (fa.DFA method), 21
mergeStates() (fa.NFAr method), 36
mergeStates() (fl.ANFA method), 85
mergeStatesSet() (fa.NFAr method), 36
minDFCA() (fl.ADFA method), 83
minimal() (fa.DFA method), 21
minimal() (fa.NFA method), 32
minimal() (fl.ADFA method), 83
minimalBrzozowski() (fa.OFA method), 13
minimalBrzozowskiP() (fa.OFA method), 13
minimalDFA() (fa.NFA method), 32
minimalHKP() (fa.DFA method), 21
minimalHopcroft() (fa.DFA method), 21
minimalHopcroftP() (fa.DFA method), 21
minimalIncremental() (fa.DFA method), 21
minimalIncrementalP() (fa.DFA method), 22
minimalMoore() (fa.DFA method), 22
minimalMooreSq() (fa.DFA method), 22
minimalMooreSqP() (fa.DFA method), 22
minimalNCompleteP() (fa.DFA method), 22
minimalNotEquivP() (fa.DFA method), 22
minimalP() (fa.DFA method), 22
minimalP() (fl.ADFA method), 83
minimalWatson() (fa.DFA method), 23
minimalWatsonP() (fa.DFA method), 23
minReversible() (fl.ADFA method), 83
minWord() (fa.EnumL method), 40
minWordT() (fa.EnumL method), 40
moveFinal() (fl.ANFA method), 85
multiLineAutomaton() (fl.FL method), 80

N

next() (fl.RndWGen method), 85
next() (rndadfa.ADFArnd method), 98
next() (rndfa.ICDFArgen method), 95
nextWord() (fa.EnumL method), 40
NFA (class in fa), 27
nfaFollow() (reex.regexp method), 49
nfaFollowEpsilon() (reex.regexp method), 49
nfaGlushkov() (reex.regexp method), 49
nfaNaiveFollow() (reex.regexp method), 49
nfaPD() (reex.regexp method), 49
nfaPDO() (reex.regexp method), 50
nfaPosition() (reex.regexp method), 50
nfaPSNF() (reex.regexp method), 50
NFAr (class in fa), 35
nfaThompson() (reex.atom method), 64
nfaThompson() (reex.disj method), 58
nfaThompson() (reex.epsilon method), 53
nfaThompson() (reex.option method), 60
nfaThompson() (reex.star method), 56
noBlankNames() (fa.FA method), 8
nonEmptyW() (transducers.SFT method), 76
nonFunctionalW() (transducers.SFT method), 76
normalize() (fa.GFA method), 38
notequal() (fa.DFA method), 23
notMaximalW() (codes.ErrCorrectProp method), 109
notMaximalW() (codes.IPTProp method), 107
notMaxStatW() (codes.IPTProp method), 107

notSatisfiesW() (codes.ErrCorrectProp method), 109
notSatisfiesW() (codes.IATProp method), 108
notSatisfiesW() (codes.IPTProp method), 107
NULLABLE() (cfg.CFGrammar method), 91

O

OFA (class in fa), 11
option (class in reex), 60
ordered() (fl.AFA method), 81
orderedStrConnComponents() (fa.DFA method), 23
outIntersection() (transducers.SFT method), 76
outIntersectionDerived() (transducers.SFT method), 76
outputS() (transducers.SFT method), 77

P

pairGraph() (fa.DFA method), 23
ParseReg (class in reex), 66
ParserFAdo (class in fio), 45
ParserGrail (class in grail), 113
ParserVerso (class in verso), 115
partialDerivatives() (reex.atom method), 64
partialDerivatives() (reex.epsilon static method), 53
partialDerivatives() (reex.sconcat method), 67
partialDerivatives() (reex.sconj method), 70
partialDerivatives() (reex.sdisj method), 69
partialDerivatives() (reex.sigmaP method), 55
partialDerivatives() (reex.sigmaS method), 55
partialDerivatives() (reex.snot method), 70
partialDerivatives() (reex.sstar method), 68
partialDerivativesC() (reex.atom method), 64
partialDerivativesC() (reex.emptyset method), 54
partialDerivativesC() (reex.epsilon method), 53
partialDerivativesC() (reex.sconcat method), 67
partialDerivativesC() (reex.sconj method), 70
partialDerivativesC() (reex.sdisj method), 69
partialDerivativesC() (reex.sigmaP static method), 55
partialDerivativesC() (reex.sigmaS method), 55
partialDerivativesC() (reex.snot method), 71
partialDerivativesC() (reex.specialConstant method), 52
partialDerivativesC() (reex.sstar method), 68
PD() (reex.atom method), 61
plus() (fa.FA method), 8
position (class in reex), 65
possibleToReverse() (fa.DFA method), 23
possibleToReverse() (fl.ADFA method), 84
power (class in reex), 59
pref() (fa.DFA method), 23
PrefixProp (class in codes), 108
print_data() (fa.DFA method), 23
product() (fa.DFA method), 23
product() (fa.NFA method), 32
productInput() (transducers.SFT method), 77
productInputSlow() (transducers.SFT method), 77
productSlow() (fa.DFA method), 24

R

re_stateElimination() (fa.OFA method), 14
re_stateElimination_nn() (fa.OFA method), 14
readFromFile() (in module fio), 45
reCG() (fa.OFA method), 13
reCG_nn() (fa.OFA method), 13
reduced() (reex.atom method), 64
reDynamicCycleHeuristic() (fa.OFA method), 13
reex (module), 47
regexp (class in reex), 47
regexp() (fa.DFA method), 24
regexpSE() (fa.OFA method), 14
renameState() (fa.FA method), 8
renameStates() (fa.FA method), 8
renameStatesFromPosition() (fa.NFA method), 33
reorder() (fa.DFA method), 24
reorder() (fa.GFA method), 38
reorder() (fa.NFA method), 33
rEquivNFA() (fa.NFA method), 32
reStaticCycleHeuristic() (fa.OFA method), 13
reStringRGenerator (class in cfg), 92
reversal() (fa.FA method), 9
reversal() (fa.NFA method), 33
reversal() (reex.atom method), 64
reversal() (reex.concat method), 57
reversal() (reex.disj method), 59
reversal() (reex.option method), 60
reversal() (reex.power method), 59
reversal() (reex.specialConstant method), 52
reversal() (reex.star method), 57
reversal() (transducers.SFT method), 77
reverseTransitions() (fa.DFA method), 24
reverseTransitions() (fa.NFA method), 33
rndadfa (module), 97
rndAdfa() (rndadfa.ADFArnd method), 98
rndfa (module), 95
rndNumberSecondSources() (rndadfa.ADFArnd method), 98
rndTransitionsFromSources() (rndadfa.ADFArnd method), 99
RndWGen (class in fl), 85
rpn() (reex.atom method), 64
rpn() (reex.concat method), 58
rpn() (reex.emptyset method), 54
rpn() (reex.epsilon method), 53
rpn() (reex.regexp method), 50
rpn2regexp() (in module reex), 71
runOnNFA() (transducers.SFT method), 77
runOnWord() (transducers.SFT method), 77

S

same_nullability() (fa.FA method), 9
satisfiesP() (codes.ErrCorrectProp method), 109
satisfiesP() (codes.IPTProp method), 107
satisfiesPrefixP() (codes.PrefixProp method), 108
saveToFile() (in module fio), 46
saveToString() (in module fa), 40
sconcat (class in reex), 67
sconj (class in reex), 69
sconnective (class in reex), 66

sdisj (class in reex), 68
SemiDFA (class in fa), 10
setDeadState() (fl.AFA method), 82
setExecPath() (grail.Grail method), 113
setFinal() (fa.FA method), 9
setInitial() (fa.FA method), 9
setInitial() (fa.NFA method), 33
setInitial() (transducers.SFT method), 77
setOfSymbols() (reex.atom method), 64
setOfSymbols() (reex.regexp method), 50
setOfSymbols() (reex.sconnective method), 66
setOfSymbols() (reex.snot method), 71
setOfSymbols() (reex.specialConstant static method), 52
setOutput() (transducers.Transducer method), 73
setSigma() (fa.FA method), 9
setSigma() (fl.FL method), 80
setSigma() (reex.regexp method), 50
SFT (class in transducers), 73
shuffle (class in reex), 61
shuffle() (fa.DFA method), 24
shuffle() (fa.NFA method), 33
sigmaInitialSegment() (in module fl), 86
sigmaP (class in reex), 54
sigmaS (class in reex), 55
simDiff() (fa.DFA method), 25
smallAlphabet() (in module cfg), 93
sMonoid() (fa.DFA method), 24
snf() (reex.atom method), 64
snf() (reex.epsilon method), 53
snot (class in reex), 70
sop() (fa.DFA method), 25
specialConstant (class in reex), 51
SPRegExp() (fa.OFA method), 11
square() (transducers.SFT method), 77
square_fv() (transducers.SFT method), 77
SSemiGroup (class in fa), 39
sSemigroup() (fa.DFA method), 24
sstar (class in reex), 68
star (class in reex), 56
star() (fa.DFA method), 25
star() (fa.NFA method), 33
star() (transducers.SFT method), 78
starConcat() (in module comboperations), 101
starDisj() (in module comboperations), 101
starHeight() (reex.atom static method), 65
starHeight() (reex.regexp method), 50
starI() (fa.DFA method), 25
starInter() (in module comboperations), 102
starInter0() (in module comboperations), 102
starWConcat() (in module comboperations), 101
stateChildren() (fa.DFA method), 25
stateChildren() (fa.GFA method), 38
stateChildren() (fa.NFA method), 33
stateChildren() (fa.OFA method), 14
stateIndex() (fa.FA method), 9
stateName() (fa.FA method), 9
statePairEquiv() (fl.ADFA method), 84
str2regexp() (in module reex), 71
str2sre() (in module reex), 71
stringLength() (reex.atom method), 65
stringToADFA() (in module fl), 86
stringToDFA() (in module fa), 41
stronglyConnectedComponents() (fa.DFA method), 25
stronglyConnectedComponents() (fa.NFA method), 33
subword() (fa.DFA method), 25
subword() (fa.NFA method), 33
succintTransitions() (fa.DFA method), 26
succintTransitions() (fa.FA method), 10
succintTransitions() (fa.NFA method), 34
succintTransitions() (fa.OFA method), 14
succintTransitions() (transducers.Transducer method), 73
suff() (fa.DFA method), 26
suffixClosedP() (fl.FL method), 80
support() (reex.atom method), 65
support() (reex.conj method), 61
support() (reex.sconcat method), 67
support() (reex.sconj method), 70
support() (reex.sdisj method), 69
support() (reex.shuffle method), 61
support() (reex.sigmaP method), 55
support() (reex.sigmaS method), 55
support() (reex.snot method), 71
support() (reex.specialConstant method), 52
support() (reex.sstar method), 68
supportlast() (reex.atom method), 65
supportlast() (reex.shuffle method), 61
supportlast() (reex.specialConstant method), 52
syncPower() (fa.DFA method), 26
syncWords() (fa.DFA method), 26
syntacticLength() (reex.atom static method), 65
syntacticLength() (reex.sconnective method), 66
syntacticLength() (reex.snot method), 71

T

tail() (reex.sconcat method), 67
tail_rev() (reex.sconcat method), 67
toADFA() (fa.DFA method), 26
toANFA() (fl.ADFA method), 84
toDFA() (fa.DFA method), 26
toDFA() (fa.NFA method), 34
toDFA() (fl.FL method), 80
toDFA() (reex.regexp method), 50
toGFA() (fa.DFA method), 26
toGFA() (fa.NFA method), 34
toGFA() (fa.OFA method), 14
toInNFA() (transducers.SFT method), 78
toNFA() (fa.DFA method), 26
toNFA() (fa.NFA method), 34
toNFA() (fa.NFAr method), 36
toNFA() (fl.ADFA method), 84
toNFA() (fl.FL method), 80
toNFA() (reex.regexp method), 50
toNFAr() (fa.NFA method), 34
toNFT() (transducers.SFT method), 78

toOutNFA() (transducers.SFT method), 78
topoSort() (fa.OFA method), 14
toSFT() (transducers.SFT method), 78
TrajProp (class in codes), 105
trajToTransducer() (codes.TrajProp static method), 105
Transducer (class in transducers), 73
transducers (module), 73
treeLength() (reex.atom static method), 65
treeLength() (reex.regexp method), 51
treeLength() (reex.sconnective method), 66
treeLength() (reex.snot method), 71
trieFA() (fl.FL method), 80
trim() (fa.OFA method), 14
trim() (fl.ADFA method), 84
trim() (transducers.SFT method), 78
trimP() (fa.OFA method), 14

U

union() (fa.FA method), 10
union() (fl.FL method), 80
union() (transducers.SFT method), 78
unionSigma() (reex.regexp method), 51
uniqueRepr() (fa.DFA method), 26
uniqueRepr() (fa.NFA method), 34
uniqueRepr() (fa.OFA method), 14
unlinkSoleIncoming() (fa.NFAr method), 36
unlinkSoleOutgoing() (fa.NFAr method), 36
unmark() (fa.DFA method), 27
unmark() (reex.concat method), 58
unmark() (reex.disj method), 59
unmark() (reex.specialConstant method), 52
unmark() (reex.star method), 57
unmarked() (reex.atom method), 65
unmarked() (reex.specialConstant method), 52
usefulStates() (fa.DFA method), 27
usefulStates() (fa.NFA method), 34
usefulStates() (fa.OFA method), 15

V

vDescription() (fa.DFA static method), 27
vDescription() (fa.NFA static method), 34
verso (module), 115
vertexIndex() (graphs.Graph method), 87

W

weight() (fa.GFA method), 38
weightWithCycles() (fa.GFA method), 38
witness() (fa.DFA method), 27
witness() (fa.NFA method), 34
witnessDiff() (fa.DFA method), 27
Word (class in common), 43
wordDerivative() (reex.regexp method), 51
wordDerivative() (reex.specialConstant method), 52
wordGenerator() (fl.ADFA method), 84
WordI() (fa.SSemiGroup method), 39
wordImage() (fa.NFA method), 34
WordPS() (fa.SSemiGroup method), 39
words() (fa.FA method), 10