FAdo: tools for finite automata and regular expressions manipulation

Rogério Reis　　　　Nelma Moreira

Technical Report Series: DCC-2002-02

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

`http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html`

FAdo: tools for finite automata and regular expressions manipulation

Rogério Reis and Nelma Moreira

DCC-FC& LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150 Porto, Portugal
email: {rvr,nam}@ncc.up.pt

August 2002

Abstract

FAdo is an ongoing project which goal is the development of a Python environment for manipulation of finite automata and regular expressions. Currently it provides most standard automata operations including conversion from deterministic to non-deterministic, minimisation, boolean operations, concatenation, conversion between automata and regular expressions, and word recognition. It includes, also, an innovative method for testing non-equivalence of two automata (or regular expressions) using a `DFA` canonical form and a witness generator of the difference of two automata.

Our main motivation is the development of interactive tools for teaching concepts in automata theory and formal languages. Towards this goal we are also developing graphical tools for creating and manipulating automata and a graphical user interface.

Key-words: Automata theory, Interactive visual tools, e-learning

1 Introduction

Regular languages are fundamental computer science structures and most efficient software tools are available for their representation and manipulation. But for experimenting, studying and teaching their formal and computational models it is useful to have tools for manipulating them as first-class objects. Automata theory and formal languages courses are mathematical in essence, and traditionally are taught without computers. Well known advantages of the use of computers in education are: interactive manipulation, concepts visualisation and feedback to the students. We believe that an automata theory course can benefit from this advantages, because:

- most of the mathematical concepts can be visualised graphically. Interactivity can help in the consolidation of the concepts and an easier grasp of the formal notation.
- most of the theorem proofs are algorithmic and can be interactively constructed
- automatic correction of exercises provides immediate feedback to the students allowing for a quicker understanding of the concepts.

In this paper, we describe a collection of tools implemented in Python [Lut96] that are a first step towards an interactive environment to teach and experiment with regular and other formal languages. The use of Python, a high-level object-oriented language with high-level data types and dynamic typing, allows us to have a system which is modular, extensible, clearly and easily implemented, and portable. The Python interface to the `Tk` graphical toolkit [Ous94], gives a good platform to build the graphical environment.

In the next section, we present the implementation details of the package and we describe which functionalities are currently available. For the more technical aspects, we assume some familiarity with the Python language. In Section 3 we show how **FAdo** can be used to solve and correct some automata theory problems. Section 4 introduces the graphical environment and its functionalities. Some related work is discussed in Section 5. Planned future work is presented in Section 6.

2 Representing Regular Languages

We assume basic knowledge of formal languages and automata theory [HU79, HMU00, Koz97]. The set of regular languages over an alphabet Σ contains $\emptyset$, $\{\epsilon\}$, $\{a\}$ for all $a \in \Sigma$, and is closed under union, concatenation and Kleene closure. Regular languages can be represented by regular expressions (`regexp`) or finite automata (`FA`), among other formalisms. Finite automata can be deterministic (`DFA`) or non-deterministic (`NDFA`). All three notations can represent the same set of languages. In **FAdo**, we can manipulate each of these representations and convert between them, as shown in Figure 1.

$$\begin{tikzcd}
\textbf{NDFA} \arrow[r] & \textbf{DFA} \arrow[dl] \arrow[d] \\
\textbf{regexp} \arrow[u] & \textbf{MinDFA} \arrow[l, dashed]
\end{tikzcd}$$

Figure 1: Conversions between regular language representations

2.1 Finite Automata

A finite automaton has a finite set of states and each transition from a state to another one is parametrised by an input symbol. It is deterministic if given an input symbol there is only one state to which the automaton can transition from the any state. It is non-deterministic, otherwise.

Formally a deterministic finite automaton (`DFA`) is specified by a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where S is the set of states, Σ is the input alphabet, δ is the transition function $\delta : S \times \Sigma \to S$, s_0 the start symbol, and F is a set of final states. In a nondeterministic automata δ is a function from $S \times \Sigma$ to the set of subsets of S ($\mathcal{P}(S)$), $\delta : S \times \Sigma \to \mathcal{P}(S)$.

The class `FA` implements the basic structure of finite automata shared by deterministic and non-deterministic ones. This class defines

`Sigma` the input alphabet

`States` the set of states

`Initial` the initial state

`Final` the set of final states.

This class also provides methods for manipulating this structures: add, set, delete, test, etc. A list of its main attributes and methods can be found in Table 1, Appendix A. As we suppose that a finite automata is either a `DFA` or `NDFA`, instances of this class are not to be defined by themselves.

2.1.1 Nondeterministic Automata

The class `NDFA` inherits from the class `FA`, and provides methods to manipulate a `NDFA`. In the literature, there is a distinction between `NDFA` with and without ϵ-transitions (`NDFA` and ϵ-`NDFA`). An ϵ-transition is a transition that does not consume an input symbol (ϵ represents the empty string). Normally, ϵ is added to the alphabet and a special treatment is given to that type of transitions. In **FAdo**, we allowed all `NDFA`'s to be ϵ-`NDFA`. But it is easy to construct methods to test for ϵ-transitions and to convert an ϵ-`NDFA` to a `NDFA`. This class defines the method `addTransition()` for constructing the transition function and the method `evalWord()` for evaluating whether a word is recognised. See Table 2, Appendix A, for more details.

2.1.2 Deterministic Automata

The class `DFA` inherits from the class `FA`, and provides methods to manipulate a `DFA`. Mathematically `DFA`'s are richer than `NDFA`'s[1]. In particular, evaluating whether a word is recognised by a `DFA`, is the most efficient way to test membership in a regular language (it is a `DLOGSPACE`-complete problem[JBR91]). But there are other more important features which we analyse in the next paragraphs.

Minimisation, Equivalence and a canonical form for `DFA`'s
It is possible to test if two `DFA`'s are equivalent (i.e, if they define the same language), and given a `DFA` to find an equivalent `DFA` that has a minimum number of states. The key point is to find states that are equivalent or *indistinguishable*. Two states are equivalent if for all inputs, from both of them either a final state is reachable or not. Equivalent states can be merged in only one state. The method `Minimal()` implements `DFA` minimisation using the *table-filling* algorithm [HMU00] to find equivalent states.

For testing equivalence of two `DFA`'s, we can minimise the two automata and verify if the two minimised `DFA`'s are *isomorphic* (i.e are the same up to renaming of states). The best way to verify isomorphism is to have a canonical form for `DFA`'s. Considering a (complete) `DFA (Sigma,States,delta,Initial,Final)` we can obtain a unique string that represents it. Let `Sigma` be ordered (p.e, lexicographically), the set `States` is reordered in the following manner:

1. `Initial` is the first state
2. Following `Sigma` order, visit the states reachable from `Initial` in one transition, and if a state was not yet visited, give him the next number.
3. Repeat step 2 for the second state, third state,... until the number of the current state is the number of states (in the new order).

For each state a list of the states visited from it is made and a list of these lists is constructed. The list of final states is appended to that list. The result is a *canonical form*. An example is given in Figure 2 and the algorithm is presented in Figure 3.

If a `DFA` is minimal, the alphabet and its canonical form uniquely represent a regular language. For test equivalence we only need to check whether the alphabet and the canonical form are the same.

It is interesting to note that this approach does not induce a canonical form for `NDFA`'s. It is known that minimal automata for a `NDFA` are not unique (and very hard to obtain, actually it is a `PSPACE`-complete problem [MS72]). But given a `NDFA` we can obtain a equivalent minimal `DFA` in canonical form. The example in Figure 4 shows that we can not induce the state's order of that `DFA` to the states of the `NDFA`. As far as we know, it is an open question if there can be a canonical form for `NDFA`'s...but if it exists it must be very large...

[1]In fact `NDFA`, as other nondeterministic machines, more than computational devices, they are convenient and succinct representations...

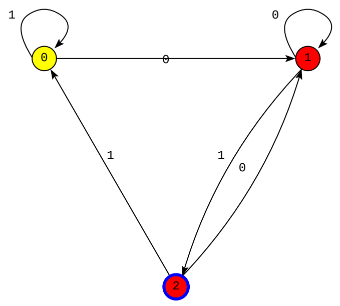


Figure 2: Let `Sigma={1,0}`, `Initial=0` and `Final={2}`, the canonical form of this `DFA` is `[0,1],[2,1],[0,1],[2]`

```
def uniqueStr(self):
    tf, tr = {}, {}
    string = []
    i , j = 0, 0
    tf[self.Initial], tr[0] = 0, self.Initial
    while i <= j:
       list = []
       for c in self.Sigma:
           foo = self.delta[tr[i]][c]
           if foo not in tf.keys():
              j = j + 1
              tf[foo] , tr[j] = j, foo
           list.append(tf[foo])
       string.append(list)
       i = i + 1
 list = []
 for s in self.Final:
     list.append(tf[s])
 list.sort()
 string.append(list)
 return string
```

Figure 3: Determining a unique representation for a `DFA`

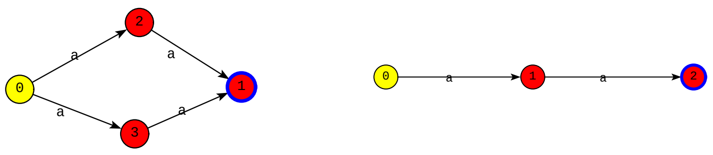


Figure 4: A state order in the minimal `DFA` at the right, can not induce a unique state order in the (equivalent) `NDFA` at the left.

Other DFA operations

Besides union, concatenation and Kleene closure, regular languages are closed under other operations, as intersection, complement, difference of two languages and reverse. Some of the above operations are trivial using regular expressions. In the current implementation we choose to define, in the class `DFA`, those which are "closed" for `DFA`'s, that is, are usually performed without the construction of a `NDFA` or other regular language representation. From the above set we excluded, concatenation and Kleene closure. See Table 3, Appendix A, for more details. Note that we used the special overloading methods (preceded and followed by double underscores) for the standard operations `|`, `&` , `~`, `=`, etc.

Witnessing the difference of two automata

Although finite automata are recognition devices, not generation devices (as grammars are), sometimes it is useful to generate a word recognisable by an automaton (*witness*). This is the case, in correcting exercises where we have the solution and an answer from a student. Instead of only reporting that an answer is wrong, we can exhibit a word that belongs to the language of the solution, but not to the language of the answer (or vice-versa).

A *witness* of a `DFA`, can be obtained by finding a path from the initial state to some final state. If no *witness* is found, the `DFA` accepts the empty language.

Given `A` and `B` two `DFA`'s, if $\neg$`A` $\cap$ `B` or `A` $\cap$ $\neg$`B` have a witness then `A` and `B` are not equivalent. If both `DFA`'s accept the empty language, `A` and `B` are equivalent. This test is implemented by the method `witnessDiff()`.

2.1.3 Converting NDFA's to DFA's

The equivalence of nondeterministic and deterministic automata is one of the most important facts about regular languages. Trivially a `DFA` can be seen as a `NDFA`. The conversion of a `NDFA` to a `DFA` that describes the same language, can be achieved by *subset construction* [HMU00]. This method is usually teached in automata theory courses, so it is important to be able to illustrate and animate it. It is is implemented by the module function `NDFA2DFA()`, that given a `NDFA` as argument, returns a `DFA`.

2.1.4 File Format for I/O

In the current version, we have a very simple format to read and write finite automata definitions. Each file can contain several definitions and must obey the following specifications:

- an `#` begins a comment
- `@DFA` or `@NDFA` begins a new automata (and determines its type). It must be followed by the list of the final states separated by blanks
- each following line represents a transition. It is a triple that consists of the source state (name), an input symbol, and the target state (name). Each of the fields are separated by a blank. The name of a state can be any string (except `#`).
- the source state of the first transition is the name of the initial state.

The automaton represented in Figure 2 can be specified by the following instructions:

```
@DFA 2
0 1 0
0 0 1
1 1 2
1 0 1
```

```
2 1 0
2 0 1
```

The method `readFromFile()` reads finite automata definitions from a file and returns a list of `DFA`'s and/or `NDFA`'s. The method `saveToFile()` saves a finite automaton definition to a file.

2.2 Regular Expressions

A regular expression can be a symbol of the alphabet, the empty set ($\emptyset$), the empty string (ϵ) or the concatenation or the union ($+$) or the Kleene star ($\star$) of a regular expression. Examples of regular expressions are `a+b`, $(\mathtt{a} + \mathtt{ba})\star$ and $(\epsilon + \mathtt{a})(\mathtt{ba} + \mathtt{ab} + \emptyset)$.

The class `regexp` implements the three base cases and the complex cases are the subclasses `concat`, `disj` and `star`, respectively. For base cases, the attribute `val` stores the value of the regular expression. The constant `Epsilon` represents the empty string and the constant `Emptyset` represents the empty set. For complex cases, the arguments are stored in the attributes `arg`, for unary, and `arg1` and `arg2`, for binaries. See Table 6, Appendix A, for more details.

2.2.1 Regular Expressions I/O

The input/output of a `regexp` instance is the usual one, with ϵ represented by `@` and {} representing $\emptyset$. Output is implemented by the `__str__` (overloading) standard method of each class. For the input, we used a parser generator, `pyParser`, that was also developed for this project. Specifically for regular expressions we had to provide a lexical analyser, a grammar and the associated semantic rules. In the lexical analyser, we restricted the alphabet to letters and digits. The grammar is a classical non-ambiguous context-free grammar for regular expressions, where the operators precedence order is union, concatenation and star. The semantic rules compositionally build a `regexp` instance from their arguments. For example, the semantic function for the union rule $\mathtt{r} \to \mathtt{r} + \mathtt{c}$ is:

```
def OrSemRule(args,context=None):
   return disj(args[2],args[0])
```

The class `ParseReg` implements the parser for regular expressions and the method `str2regexp()` returns a `regexp` object, given a string as argument.

2.3 Converting Finite Automata to Regular Expressions

The conversion from `DFA`'s to regular expressions, is based on successively constructing regular expressions $r_{ij}^{(k)}$, that represent the language recognised between state i and state j, without going through a state number higher than k [HMU00]. It is implemented by the method `regexp()` of the class `DFA`. This algorithm is mathematically very instructive, but it is highly inefficient: it can build a regular expression with 4^n symbols from an automata with n states.

We plan to implement other less redundant (but less general) methods, such as the method by elimination of states.

2.4 Converting Regular Expressions to Finite Automata

The conversion is from regular expressions to *epsilon*-`NDFA`'s using the *Thompson's* construction [HMU00, Tho68]. The idea is to recursively build an ϵ-`NDFA` for each type of `regexp`. Each `regexp` subclass has a method `ndfa()` that allows to construct an `NDFA` for its type.

3 Using FAdo

In this section we illustrate how **FAdo** can be used for solving (and correcting) some typical exercises for a first course in automata theory. In Python interactive mode, we must first import the package:

```
>>> from dfa import *
>>>
```

Example 1 *Convert to a regular expression the following* `NDFA`*:*

```
@NDFA 2
0 1 1
1 1 2
2 0 1
2 0 2
2 0 0
```

```
>>> n=readFromFile("examples/e1.fa")[0]
>>> d=NFDA2DFA(n)
>>> print d.regexp().simplify()
((1 1) + (1 1)) + (1 1 0 0* 1 ((0 + (1 0)) 0* 1)* (1 + 1))
```

◇

Example 2 *Convert the regular expression* (0+1)∗(012) *to a* `NDFA`*. Obtain an equivalent minimal* `DFA`*(and the* `DFA` *canonical form).*

```
>>> a=str2regexp("(0+1)*(012)")
>>> d=NDFA2DFA(a.ndfa())
>>> d.Minimal()
>>> saveToFile("e2.fa",d)
>>> d.uniqueStr()
[[1, 0, 2], [1, 3, 2], [2, 2, 2], [1, 0, 4], [2, 2, 2], [4]]
```

The minimal automata definition that was saved is the following:

```
@DFA 3
0 1 0
0 0 1
0 2 4
1 1 2
1 0 1
1 2 4
2 1 0
2 0 1
2 2 3
3 1 4
3 0 4
3 2 4
4 1 4
4 0 4
4 2 4
```

◇

Example 3 *Check whether the following two regular expressions are equivalent* (01+0)∗ *and* 0(10 + 0)∗*.*

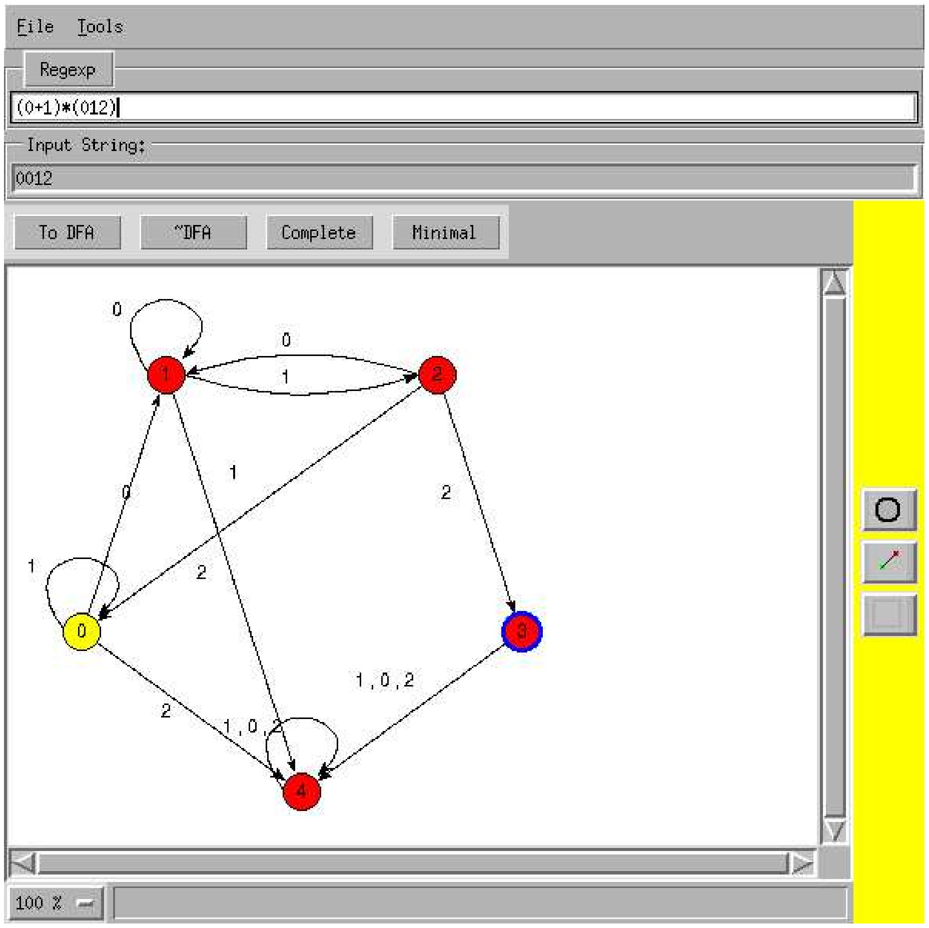
```
>>> a=str2regexp("(01+0)*")
>>> b=str2regexp("0(10+0)*")
>>> da=NDFA2DFA(a.ndfa())
>>> db=NDFA2DFA(b.ndfa())
>>> da.witnessDiff(db)
@
```

The two regular expressions are not equivalent: the first one describes a language that contains the empty string but not the second one. ⋄

In the last example we could have defined a method that given two (or more) regular expressions would determine if they were equivalent. The possibility of constructing new methods from those defined in the package is one of the advantages of **FAdo**.

4 Graphical Interface

Currently, the great advantage of the **FAdo** graphical environment is to allow the visualisation and the edition of the diagrams representing finite automata. Figure 5 shows a diagram for the minimal DFA of Example 2.



Regexp
(0+1)*(012)

Input String:
0012

Figure 5: **FAdo** graphical interface

A diagram can be constructed from a finite automata definition, or created/transformed using the edit toolbox (at the right side of the interface). The editing operations available are:

- add/move a state
- add a transition between two states; a label is prompted to the user.
- delete a state or a transition

Clicking `Button3` in a state we can make it the initial (yellow background) and/or a final state (blue border).

In the first implementation, we used a basic graph drawing library from the `Gato` project whose aim is the visualisation and the animation of algorithms on graphs [gat02]. But, that library was not easily adaptable to our needs and we decided to implement an independent library for graph visualisation.

In the current version, the graphical user interface provides access to some conversions whose results are visualised:

`NDFA`→ `DFA` pressing the `To DFA` button

`DFA`→ `regexp` pressing the `Regexp` button

`regexp`→ `MinDFA` pressing `return-key` after giving a regular expression

`DFA`→ `MinDFA` pressing the `Minimal` button

`DFA`→ **complete** `DFA` pressing the `Complete` button

`DFA`→ ¬`DFA` pressing the `~DFA` button

Given a `NDFA` or a `DFA`, an input string can be entered and evaluated.

5 Related Work

There are several projects whose goal is the manipulation of automata theory objects. In late 1999, when this project begun, not many packages provided graphical and interactive manipulation. That was the case of the `Grail+` project, a `C++` library for finite automata and regular expressions [RW95]. `Grail+` operations are accessible either as individual programs (used as shell filters) or through a `C++` class library. In the `Grail`'s homepage [fas02],it can be found a set of links to other automata theory software which includes `AUTOMATE` [CH91], `AMoRE` [JPTW90], `Fire Lite` [Wat], etc. In the last few years several `Java applets` for finite automata processing became available on-line, but normally their functionalities are very limited. `JFLAP` is one of the projects that shares most of our goals. Its main motivation is teaching and the current emphasis is in animation and interactive visualisation of automata theory concepts [BLP+97, GR99, HR00]. `JFLAP` supports drawing and execution of finite-state machines, conversions between deterministic and non-deterministic, and animated conversions from and to regular expressions. `JFLAP` also provides tools for manipulation with grammars, pushdown automata and Turing machines. `Gaminal` is a project for the development of learning software for compiler design [DK00]. One of its components is an environment for processing finite automata. In particular, an electronic book that illustrates many of the regular languages manipulations, is available online [gan02]. Finally we refer `FSA` a set of tools for manipulating several types of finite-automata and regular expressions implemented in Prolog [fsa02].

Although our project was suspended for a couple of years and in spite of some new software being available, we think its goals are worthwhile. Obtaining a good interaction with the students and fine grain animation tools are two areas where much more research is needed.

6 Future Work

There are several different lines of future work.

The graphical environment should provide:

- interactive animation of the several conversion algorithms and word evaluations.
- simultaneous access to several automata and regular expressions, in order to compare and compose their languages.
- more information about the objects and the languages they represent.

In order to develop interactive tools for an automata theory course, much more must be done. Many more operations and algorithms can be implemented for the manipulation of finite automata and regular expressions. In particular, constructions of `NDFA`'s without ϵ-transitions from regular expressions [Glu60, MY60, BK92, Wat93], constructions of regular expressions from `DFA`'s by state elimination and simplification techniques, `DFA`'s constructions based on Myhill-Nerode theorem and Brzozowski derivatives [HU79, Brz62, Wat93], several minimisation algorithms [Wat95], etc. We will also extend the functionalities to other kinds of automata, specially to pushdown automata and transducers. In the near future we plan to implement tools for manipulating several concepts associated with regular and context-free grammars, namely, conversions between automata models, construction of derivation trees, transformations to normal forms, etc. An integrated module for the simulation of Turing machines (and unrestricted grammars) should also be part of the package.

The integration of the package for correcting exercises in a Web system or in an electronic book must be designed. In particular, the functionalities available must be parameterisable and the communication between the several components (wordings of exercises, edition of students answers, formatting of solutions, grading systems, etc) must be specified.

Finally we are also planning to use standard formats for exchanging information. We already have a proposal for extending GraphML, a new extensible XML language for graphs [gra02, UBM02], to describe automata. We also would like to export automata diagrams to the `xypic` format (for `LaTeX` documents).

Acknowledgement

We thank Miguel Filgueiras for helpful comments.

References

[BK92] Anne Brügermann-Klein. Regular expressions into finite automata. In *Proceedings of Latin '92*, 1992.

[BLP+97] Anna O. Bilska, Kenneth H. Leider, Magdalena Procopiuc, Octavian Procopiuc, Susan H. Rodger, Jason R. Salemme, and Edwin Tsang. A collection of tools for making automata theory and formal languages come alive. *SIGCSEB: SIGCSE Bulletin*, 29, 1997.

[Brz62] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962. Volume 12 of MRI Symposia Series.

[CH91] Jean M. Champarnaud and G. Hanset. AUTOMATE, a computing package for automata and finite semigroups. *Journal of Symbolic Computation*, 12:197–220, 1991.

[DK00] Stephan Diehl and Thomas Kunze. Visualizing principles of abstract machines by generating interactive animations. *Future Generation Computer Systems*, 16(7):831–839, 2000.

[fas02] Links to finite-state machines software.
`http://www.csd.uwo.ca/research/grail/links.html`, 08/2002.

[fsa02] Fsa6.2xx: Finite state automata utilities.
`http://odur.let.rug.nl/~vannoord/Fsa/fsa.html`, 08/2002.

[gan02] Ganifa, generating finite automata.
`http://rw4.cs.uni-sb.de/~ganimal/GANIFA/`, 08/2002.

[gat02] Gato project.
`http://www.zpr.uni-koeln.de/~gato/`, 08/2002.

[Glu60] V. M. Glushkov. On synthesis algorithm for abstract automata. *Ukr. Mathem. Zhurnal*, 12(2):147–156, 1960. In Russian.

[GR99] Eric Gramond and Susan H. Rodger. Using JFLAP to interact with theorems in automata theory. *SIGCSEB: SIGCSE Bulletin*, 31, 1999.

[gra02] GraphML file format.
http://graphml.graphdrawing.org/, 2002.

[HMU00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 2nd edition, 2000.

[HR00] Ted Hung and Susan H. Rodger. Increasing visualization and interaction in the automata theory course. *SIGCSEB: SIGCSE Bulletin*, 32, 2000.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[JBR91] T. Jiang and ar B. Raviku. A note on the space complexity of some decision problems. *Information Processing Letters*, 40:25–31, 1991.

[JPTW90] V. Jansen, A. Potthoff, W. Thomas, and U. Wermuth. A short guide to the AMoRE system. Aachener informatik-berichte (90) 02, Lehrstuhl fur Informatik II, Universitat Aachen, January 1990.

[Koz97] Dexter C. Kozen. *Automata and Computability.* Undergraduate texts in computer science. Springer, 1997.

[Lut96] M. Lutz. *Programming Python.* O'Reilly & Associates, 1996.

[MS72] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proceedings of the 13th Annual Symp. on switching and Automata Theory*, pages 125–129. IEEE Computer Society, 1972.

[MY60] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960.

[Ous94] John Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[RW95] Darrell Raymond and Derick Wood. Grail: A C++ library for automata and expressions. *J.Symbolic Computation*, 11, 1995.

[Tho68] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[UBM02] I. Herman M. Himsolt U. Brandes, M. Eiglsperger and M.S. Marshall. GraphML progress report: Structural layer proposal. In *Proc. 9th Intl. Symp. Graph Drawing (GD '01)*, number 2265 in LNCS, pages 501–512. Springer Verlag, 2002.

[Wat] Bruce W. Watson. The FIRE Lite: FAs and REs in C++. In *Proceedings of the First Workshop on Implementing Automata*, pages 167–188.

[Wat93] Bruce W. Watson. A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Eindhoven University of Technology, September 1995.

A Package Description

Class `FA`

Method/Attribute	**Description**
`States`	a list of states, their index represents the state and the value is an optional name (string)
`Sigma`	a set of symbols
`Initial`	the initial state
`Final`	a set of final states
`delta`	a nested dictionary that associates a state to a transition
`addState()`	adds a final state
`validateState()`	checks if a state pertains to a `FA`
`setInitial()`	sets the initial state
`setFinal()`	sets a list of final states
`setSigma()`	defines the alphabet
`addSigma()`	adds a new symbol to the alphabet
`stateName()`	given a state name returns its index
`renameStates()`	renames all states using a new list of names
`noBlankNames()`	substitutes blank state names by their index
`completeP()`	checks if it is a complete `FA`
`complete()`	transforms a `FA` in a complete `FA`
`compact()`	eliminates unused states
`__len__()`	returns the number of states

Table 1: Class for finite automata

Class `NDFA`, inherits from `FA`

Method/Attribute	**Description**
`addTransition()`	adds a new transition
`EpsilonList()`	ϵ-closure of a state
`evalWord()`	tests if the `NDFA` recognises a word
`evalSymbol()`	determines the next set of possible states consuming a symbol

Table 2: Class for nondeterministic finite automata

Class `DFA`, inherits from `FA`

Method/Attribute	**Description**
`addTransition()`	adds a new transition
`evalWord()`	tests if the `DFA` recognises a word
`evalSymbol()`	determines the next possible state consuming a symbol
`Minimal()`	minimises the `DFA`
`compat()`	tests compatibility between two states
`__cmp__()`	verifies if the two automata are equivalent
`uniqueStr()`	returns a unique string that gives us a `DFA` canonical form
`__invert__()`	returns a `DFA` that recognises the complementary language
`__or__()`	returns a `DFA` that recognises the union of two languages
`__and__()`	returns a `DFA` that recognises the intersection of two languages
`reverse()`	returns a `DFA` that recognises the reversal language
`witness()`	generates a word recognisable by the automata; if no word is found, it accepts the empty language and an exception is raised
`witnessDiff`	returns a witness for the difference of two `DFA`'s
`regexp()`	returns a regexp for the current `DFA`

Table 3: Class for deterministic finite automata

Class `regexp`

Class `disj`, inherits from `regexp`

Class `star`, inherits from `regexp`

Class `concat`, inherits from `regexp`

Method/Attribute	**Description**
`ndfa()`	returns a `NDFA` that accepts the `regexp`
`type()`	returns the `regexp` type or value
`empty()`	tests if it is *emptyset*
`epsilon()`	tests if it is ϵ
`simplify()`	applies some (naive) simplification rules

Table 4: Classes for regular expressions

Method/Attribute	**Description**
`NDFA2DFA()`	returns a `DFA` equivalent to a `NDFA`, given as argument. The method proceeds by subset construction.
`isFA()`	tests if the argument is a `FA`
`isDFA()`	tests if the argument is a `DFA`
`isNDFA()`	tests if the argument is a `NDFA`
`readFromFile()`	reads finite automata definitions from a file and returns a list of `DFA`'s and/or `NDFA`'s
`saveToFile()`	saves a finite automaton definition to a file
`str2regexp()`	parses a string and returns a `regexp`

Table 5: Module methods

Constants	**Value**	**Printable**	**Description**
`Epsilon`	`@epsilon`	`@`	empty string and regular expression (ϵ)
`EmptySet`	`@empty_set`	$\{\}$	empty set and regular expression ($\emptyset$)

Table 6: Constants