Finite State Morphology

Alexander Fraser
fraser@cis.uni-muenchen.de

CIS, Ludwig-Maximilians-Universität München

Computational Morphology and Electronic Dictionaries
SoSe 2017
2017-05-15

Outline

- Today we will cover finite state morphology more formally
  - We'll review basic concepts from the first lecture and from the exercises
  - And define operations in finite state more formally
- We will then show how to convert regular expressions to finite state automata

Credits

- Credits:
  - Slides mostly adapted from:
  - Finite State Morphology
  - Helmut Schmid
  - U. Tübingen - Summer Semester 2015

  - Thanks also to Kemal Oflazer and Lauri Kartunnen

Review: Computational Morphology

- examines word formations processes
- provides analyses of word forms such as *Tarifverhandlungen:* *Tarif<NN>verhandeln<V>ung<SUFF><+NN><Fem><Nom><Pl>*
- splits word forms into roots and affixes
- provides information on
  - part-of-speech such as NN, V
  - canonical forms such as „verhandeln"
  - morphosyntactic properties such as Fem, Nom, Pl

Terminology

- word form
  word as it appears in a running text: weitergehst
- lemma
  citation as listed in a dictionary: weitergehen
- stem
  part of a word to which derivational of inflectional affixes are attached: weitergeh
- root
  stem which cannot be further analysed: geh
- morpheme
  smallest morphological units (stems, affixes): weiter, geh, en

Word Formation Processes

- Inflection
- Derivation
- Compounding

Inflection

- modifies a word in order to express different grammatical categories such as tense, mood, voice, aspect, person, number, gender, case
- verbal inflection: conjugation walks, walked, walking
- nominal inflection: declension computers
- usually realised by
  - prefixation
  - suffixation
  - circumfixation ge+hab+t
  - infixation auf+zu+machen (not a perfect example)
  - reduplication: orang+orang (plural of „man“ in Indonesian)

Derivation

- creates new words
- Examples: un+translat+abil+ity piti+less-ness
- changes the part-of-speech and/or meaning of the word
- adds prefixes, suffixes, circumfixes

- conversion: changes the part-of-speech without modifying the word book (N) → book (V) leid(en) (V) → Leid (N)
- templatic morphology in Arabic
  ktb + CVCCVC + (a,a) -> kattab (write)

Compounding

- creates new words by combining several stems
- example: Donau-dampf-schiff-fahrts-gesellschaft
- very productive in German

- affixoid
  compounding process that turns into a derivation process
  Gas+werk, Stück+werk, Laub+werk
  schul+frei, schulter+frei, schulden+frei

→ no absolute boundary between compounding and derivation

Classification of Languages

- isolating: Chinese, Vietnamese
  little or no derivation and inflection
- analytic: Chinese, English
  little or no inflection
- synthetic
  - agglutinative: Finnish, Turkish, Hungarian, Swahili
    morphemes are concatenated with little modification
    each affix usually encodes a single feature
  - fusional (inflecting): Sanskrit, Latin, Russian, German
    inflectional affixes often encode a feature bundle: les+e (1 sg pres)

Productivity

- productive process
  new word forms can easily be created
  use+less, hope+less, point+less, beard+less

- unproductive process:
  morphological process which is no longer active
  streng+th, warm+th, dep+th

Morphotactics

Which morphemes can be arranged in which order?

translat+abil+ity

*translat+ity+abil

translat+able

*translat+able+ity (Allomorphs able-abil)

Orthographic/Phonological Rules

How is a morpheme realised in a certain context?

city+s → cities

bake+ing → baking (e-elision)

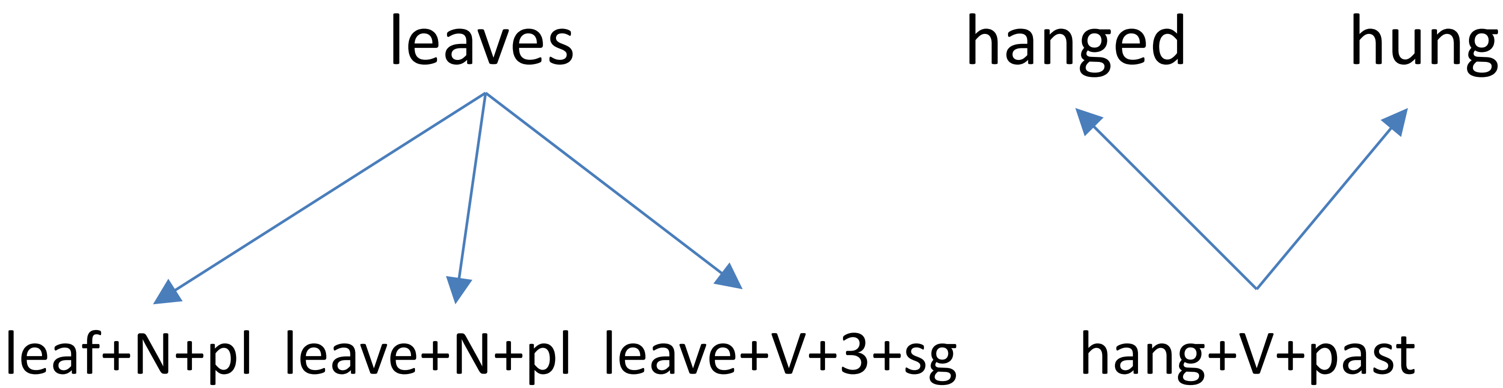
crash+s → crashes (e-epenthesis)

beg+ing → begging (gemination)

ad+simil+ate → assimilate (assimilation)

ip+lEr → ipler kız+lEr → kızlar (vowel harmony)

Morphological Ambiguity



Ingredients of a Morph. Analyser

- List of roots with part-of-speech
- List of derivational affixes
- morphotactic rules
- orthographic (phonological) rules

Computational Morphology

analyses and/or generates word forms

- analysis

  Abteilungen →
  Abteilung<NN><Fem><Nom><Pl>
  Abteilung<NN><Fem><Acc><Pl> …
  ab<VPART>teilen<V>ung<NNSuff><Fem><Acc><Pl> …
  Abtei<NN> Lunge<NN><Fem><Nom><Pl> …
  Abt<NN> Ei<NN> Lunge<NN><Fem><Nom><Pl> …
  Abt<NN> eilen<V> ung<NNSuff><Fem><Nom><Pl> …

- generation

  sichern<+V><1><Sg><Pres><Ind> → sichere, sichre

Implementation

- using a mapping table
  works reasonably well for languages such as English, Chinese
- algorithmic
  more suitable for languages with complex morphology such as Turkish or Czech
  - finite state transducers
    simple, well understood, efficient, bidirectional (analysis & generation)

Short History

1968 Chomsky & Halle propose ordered context-sensitive rewrite rules
$x \rightarrow y$ / w _ z (replace x by y in the context w … z)

1972 C. Douglas Johnson discovers that ordered rewrite rules can be implemented with a cascade of FSTs if the rules are never applied to their own output

1961 Schützenberger proved that 2 sequential transducers (where the output of the first forms the input of the second) can be replaced by a single transducer.

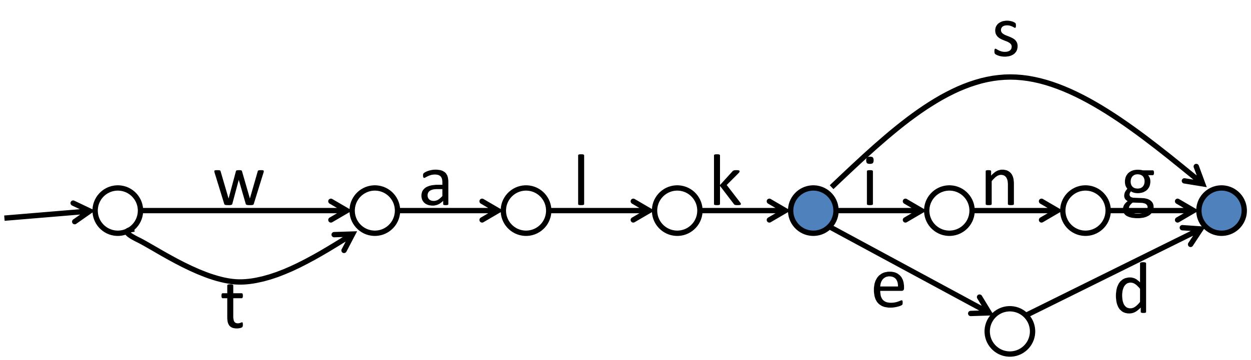
1980 Kaplan & Kay rediscover the findings of Johnson and Schützenberger

1983 Kimmo Koskenniemi invents 2-level-morphology

1987 Karttunen & Koskenniemi implement the first FST compiler based on Kaplan's implementation of the finite-state calculus

Finite State Automaton

directed graph with labelled transitions, a start state and a set of final states



recognises walk, walks, walked, walking, talk, talks, talked, talking

Finite State Automaton

FSAs are isomorphic to regular expressions and regular grammars. All of them define a regular language.

regular expression: (w|t)alk(s|ed|ing)?

regular grammar:

S → w A	B → s	B →
S → t A	B → e d	
A → a l k B	B → i n g	

both equivalent to the automaton on the previous slide

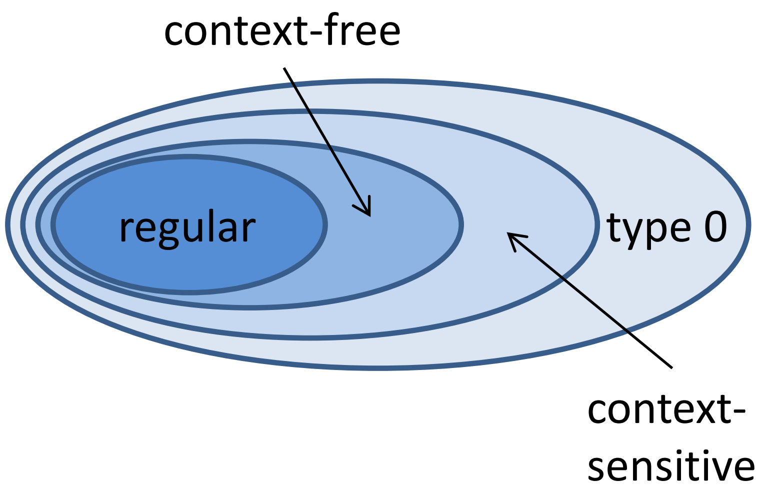
Finite State Automaton

FSAs are isomorphic to regular expressions and regular grammars. All of them define a regular language.

regular expression: (w|t)alk(s|ed|ing)?

regular grammar:

S → w A
S → t A
A → a l k B
B → s
B → e d
B → i n g
B →



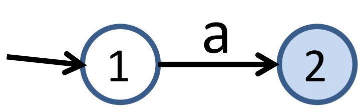
Both are equivalent to the automaton on the previous slide

Operations on FSAs

- Concatenation A B
- Optionality A? = (|A)
- Kleene‘s star A* = (|A|AA|AAA|...)
- Disjunction A | B
- Conjunction A & B
- Complement !A
- Subtraction A – B = A & !B
- Reversal

From Regular Expressions to FSAs

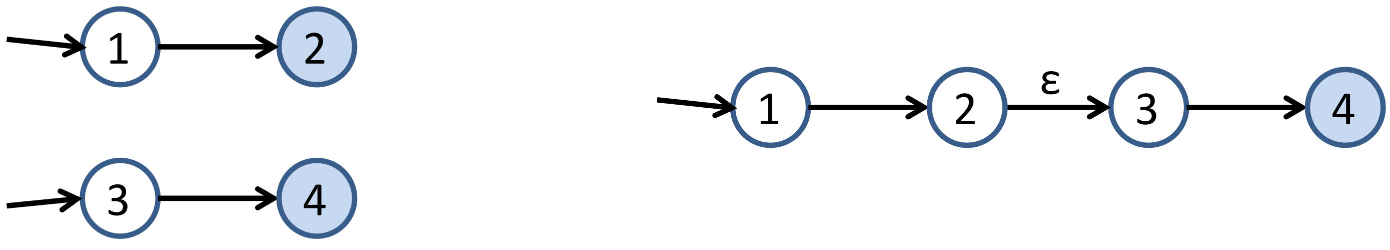
single symbol a



- Create a new start state and a new end state
- Add a transition from the start to the end state labelled „a“

From Regular Expressions to FSAs

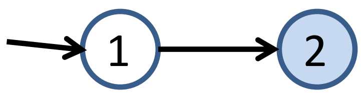
Concatenation A B

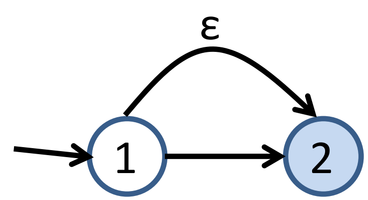


- add epsilon transition from final state of A to start state of B
- make final state of B the new final state

From Regular Expressions to FSAs

Optionality A?

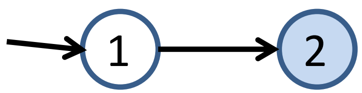


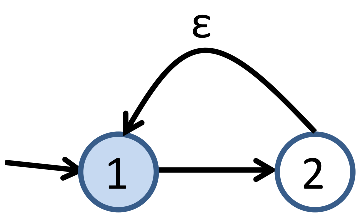


- add an epsilon transition from start to end state

From Regular Expressions to FSAs

Kleene' star A*

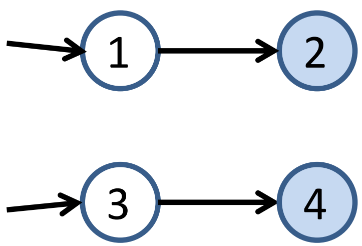


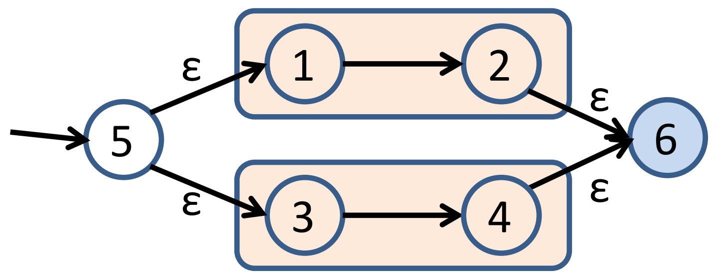


- add an epsilon transition from end to start state
- make start state the new end state

From Regular Expressions to FSAs

Disjunction A B

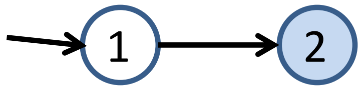


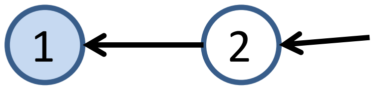


- new start state with epsilon transitions to the old start states
- new final state with epsilon transitions from the old final states

From Regular Expressions to FSAs

Reversal





- reverse all transitions
- swap start and end state

From Regular Expressions to FSAs

Conjunction A & B

- I'm skipping the details of conjunction (see the Appendix for the algorithm)
- Basically, we can automatically create a new FSA that essentially runs both acceptors in parallel
- Our new FSA only accepts if both FSAs are in the accept state
- Clearly the FSA A&B then only accepts strings that are in the regular languages accepted by both FSAs (FSA A and FSA B)

Properties of FSAs

- epsilon-free
  no transition is labelled with the empty string epsilon
- deterministic
  epsilon-free and no two transitions originating in the same state have the same label
- minimal
  no other automaton has a smaller number of states

Properties of FSAs II

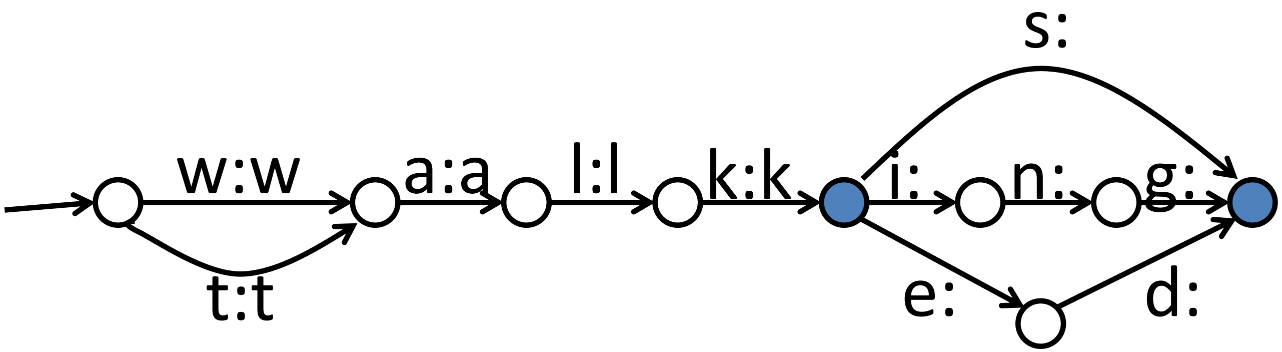
- We can algorithmically construct a new FSA from the old FSA such that it is:
  - epsilon-free
  - deterministic
  - minimal
- See the Appendix for the algorithms

Conclusion: Finite State Acceptors

- Any regular expression can be mapped to a finite state acceptor
  - However, "regexes" in Python are misnamed!
    - "Regexes" contain more powerful constructs than mathematical regular expressions
      - For instance /(.+)\1/
      - However, these constructs are not used much
    - See EN Wikipedia page on regular expressions, subsection "Regular expressions in programming languages" for details
- We will now move on to finite state transducers

Finite State Transducers

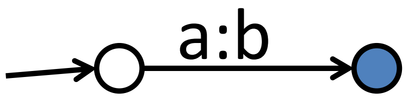
- FSTs are FSAs whose transitions are labelled with symbol pairs
- They map strings to (sets of) other strings



- maps walk, walks, walked, walking to walk
- and talk, talks, talked, talking to talk (in generation mode)
- can also map walk to walk, walks, walked, walking in analysis mode

FSTs and Regular Expressions

Single symbol mapping a:b



Operations on FSTs

- Concatenation, Kleene's star, disjunction, conjunction, complement (from FSAs)
- composition A || B
  The output of transducer A is the input of transducer B.
- projection
  - upper language replaces transition label a:b by b:b
  - lower language replaces transition label a:b by a:a

  The result corresponds to an automaton

Relations and Transducers

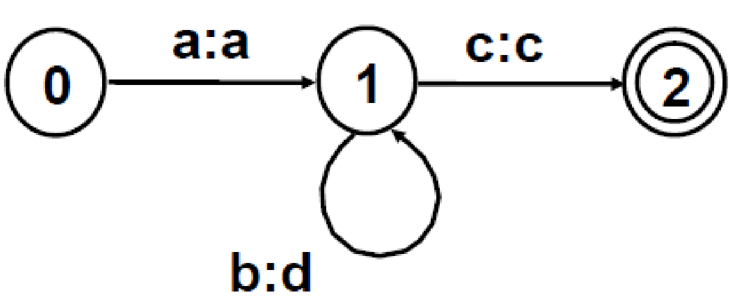
Regular relation

{ <ac,ac>, <abc,adc>, <abbc,addc>, <abbbc,adddc>... }

between [a b* c] and [a d* c].

“upper language” “lower language”

Finite-state transducer



Regular expression

a:a [b:d]* c:c

Slide courtesy of Lauri Karttunen

Relations and Transducers

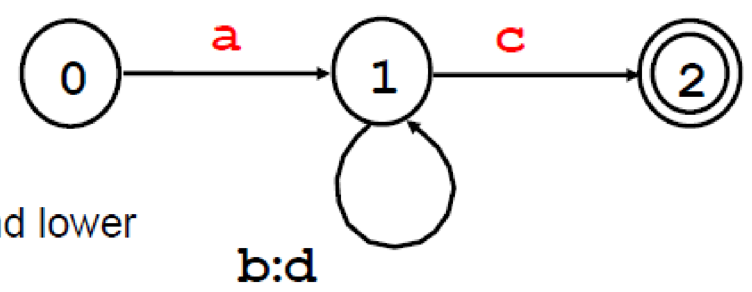
Regular relation

{ <ac,ac>, <abc,adc>, <abbc,addc>, <abbbc,adddc>... }

between [a b* c] and [a d* c].

"upper language" "lower language"

Finite-state transducer



Regular expression

`a [b:d]* c`

Convention: when both upper and lower symbols are same

Slide courtesy of Lauri Karttunen

Weighted Transducers

- A weighted FST assigns a numerical weight to each transition
- The total weight of a string-to-string mapping is the sum of the weights on the corresponding path from start to end state.
- Weighted FSTs allow disambiguation between different analyses by choosing the one with the smallest (or largest) weight

Working with FSTs

- FSTs can be specified by means of regular expressions (like FSAs). The translation is performed by a compiler.
- Using the same algorithms as for FSA
  - FSTs can be made epsilon-free in the sense that no transition is labelled with ε:ε (a pair of empty string symbols)
  - FSTs can be made deterministic in the sense that no two transitions originating in the same state have the same label pair
  - FSTs can be minimised in the sense that no other FST which produces the same regular relation with the same input-output alignment is smaller. (There might be a smaller transducer producing the same relation with a different alignment.)
- FSTs can be used in both directions (generation and analysis)

FST Toolkits

Some FST toolkits

- Xerox finite-state tools xfst and lexc
  well-suited for building morphological analysers
- foma (Mans Hulden)
  open-source alternative to xfst/lexc
- AT&T tools
  weighted transducers for tasks such as speech recognition
  little support for building morphological analysers
- openFST (Google, NYU)
  open-source alternative to the AT&T tools
- SFST
  open-source alternative to xfst/lexc but using a more general and flexible programming language

SFST

- programming language for developing finite-state transducers
- compiler which translates programs to transducers
- tools for
  - applying transducers
  - printing transducers
  - comparing transducers

SFST Example Session

> echo "Hello\ World\!" > test.fst	*storing a small test program*
> fst-compiler test.fst test.a	*calling the compiler*
test.fst: 2	
> fst-mor test.a	*interactive transducer usage*
reading transducer...	*transducer is loaded*
finished.	
analyze> Hello World!	*input*
Hello World!	*recognised*
analyze> Hello World	*another input*
no result for Hello World	*not recognised*
analyze> q	*terminate program*

SFST Programming Language

Colon operator a:b

empty string symbol <>

Example: m:m o:i u:<> s:c e:e

identity mapping a (an abbreviation for a:a)

Example: m o:i u:<> s:c e

{abc}:{AB} is expanded to a:A b:B c:<>

Example: {mouse}:{mice}

Disjunction

John | Mary | James

accepts these three strings and maps them onto themselves

mouse | {mouse}:{mice}

analyses mouse and mice as mouse

note that analysis here maps lower language (mice) to upper language (mouse), i.e., implements lemmatization

Generation goes in the opposite direction

Multi-Character Symbols

strings enclosed in <…> are treated as a single unit.

{mouse<N><pl>}:{mice}

analyzes mice as mouse<N><pl>

Multi-Character Symbols

A more complex example:

```
schreib {<V><pres>}:{} (\
   {<1><sg>}:{e} |\
   {<2><sg>}:{st} |\
   {<3><sg>}:{t} |\
   {<1><pl>}:{en} |\
   {<2><pl>}:{t} |\
   {<3><pl>}:{en})
```

The backslashes (\) indicate that the expression continues in the next line

What is the analysis of schreibst and schreiben?

Conclusion: Finite State Morphology

- Talked about finite state morphology in a more formal way
- Showed how to convert regular expressions to finite state automata
- Talked about finite state transducers for computational morphology
  - Morphological analysis and generation

- Thank you for your attention

Appendix

- Details of Conjunction of FSAs
- Algorithms for Determinisation, Composition and Minimisation of FSAs

From Regular Expressions to FSAs

Conjunction A & B

- The new state space Q is the Kartesian product of the old state spaces Q_1 and Q_2, i.e. $Q = \{(a,b) \mid a \in Q_1 \ \& b \in Q_2\}$
- The new start state is the pair of the old start states.
- The new final state is the pair of the old final states
- A transition labelled a exists from new state (a,b) to new state (c,d) iff a transition labelled a exists from a to c in A and from b to d in B, i.e. $(a,b) \to (c,d)$ iff $a \to c$ and $b \to d$

Determinisation of FSAs

- The new state set is the powerset of the old state set (set of all subsets).
- The new start state is the epsilon-closure of the old start state (i.e. the start state + all states reachable from it via epsilon transitions)
- There is a transition from state q to r labelled a iff there is a transition labelled a from some old state a in q to some old state b in r.
- The set of final states comprises all states q which contain an old final state a.

Composition of FSAs

- First, make the two FSAs deterministic.
- The new state set is then the Kartesian product of the two old state sets
- The new start state is the pair consisting of the two old start states
- There is a transition from state (a,b) to state (c,d) labelled x:z iff there is some transition labelled x:y from state a to state c and a transition labelled y:z from state b to state d
- The final state set comprises all state pairs (a,b) where both a and b are old final states.

Minimisation of FSAs

Minimisation of A

a simple (but inefficient) minimisation algorithm

1. determinise
2. reverse
3. determinise
4. reverse