Finite-State Machines

Robert Keller
November 2010

From Sequential Logic

- State-transition diagrams and tables
- Acceptor examples

Acceptance

- A string (sequence of symbols) is accepted by a DFA iff there is a path from *the one* initial state to *some* accepting state with labels corresponding to that path.

Language

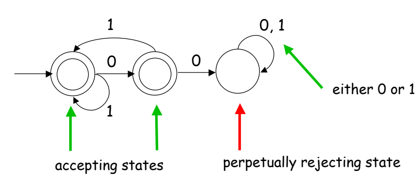
- The language accepted by a DFA is the set of all strings accepted by it.
- A language is a subset of Σ^*, the set of all finite strings of symbols in Σ.
- A language could be finite or infinite, depending on the DFA.

Language Examples

- The set of all strings over {0, 1} such that every 0, if followed by any symbol, is followed by a 1.
- The set of all strings over {0, 1} such that the number of symbols is a multiple of 4.
- The set of all strings that contain the same number of 0's and 1's.

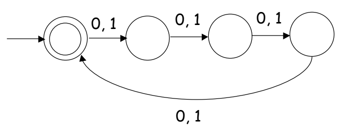
Language Examples

- The set of all strings over {0, 1} such that every 0, if followed by any symbol, is followed by a 1.



Language Examples

- The set of all strings over {0, 1} such that the number of symbols is a multiple of 4.



Language Examples

- The set of all strings with the same number of 0's and 1's.

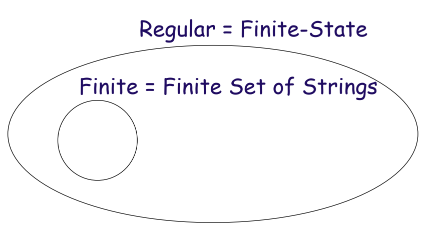
Regularity

- Some languages are accepted by a DFA.
- For some, there is no DFA.
- If there is a DFA, the language is called "regular".

Regular vs. Non-Regular Examples

- Regular: The set of all strings over {0, 1} such that the number of symbols is a multiple of 4.
- Non-Regular: The set of all strings with the same number of 0's and 1's.

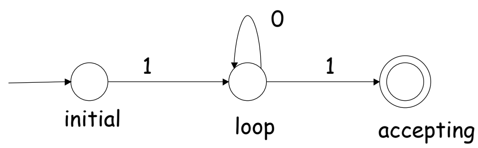
Regularity and Finiteness



When is a Regular Language Not-Finite?

- There must be a loop in the acceptor.
- There must be a state in the loop reachable from the initial state.
- There must be an accepting state reachable from a state in the loop.

When is a Regular Language Not-Finite?

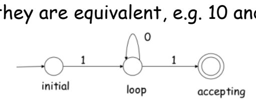


Distinguishability of Strings

- Two strings x, y are called **distinguishable** iff there a string z such that xz is accepted, but yz is not.
- If two strings are not distinguishable, they are called **equivalent**.

Example

- If two strings lead from the initial state to the same state, they are equivalent, e.g. 10 and 10000 below.

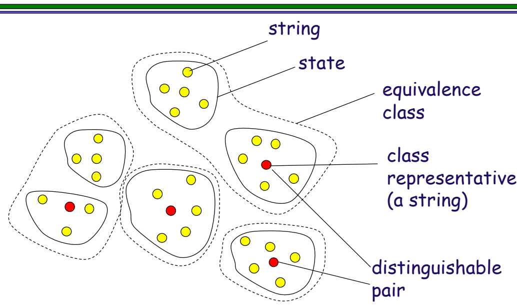


- Contrapositively, if two strings are distinguishable, they cannot possibly lead to the same state from the initial state.

States and Equivalence

- All strings that are mutually equivalent can be lumped together in a single **equivalence class**.
- If two strings x, y lead from the initial state to some common state, the strings must be equivalent.
- Thus, there are at least as many states as there are equivalence classes.

States and Equivalence states $\geq$ classes



When is a Language not Regular?

- The language must have an infinite set of equivalence classes.
- The classes correspond to an infinite set of strings, no two of which are equivalent (i.e. every pair of which is distinguishable).

Example of Non-Regular

- The set of all strings with the same number of 0's and 1's.
- 0 is distinguishable from 00, since 01 is in, but 001 is out.
- Similarly, 0^m is distinguishable from 0^n for any pair $m \neq n$.
- There is an infinite set of distinguishable pairs.

Characterization of Finite-State Machines by "Regular Expressions"

- **Regular expressions** are a *machine-independent* way of specifying a language.
- They are often used in textual **pattern-matching** applications.
- They are closely related to **grammars**, but the form of recursion is limited to "iterative" forms only.

Regular Expressions

- Discovered by the mathematical-logician S.C. Kleene (1909-1994, Prof. at U. of Wisconsin) in studying "nerve nets" in 1956.
- Kleene was also a principal developer of the field of recursion (computability) theory

About Mr. Kleene

Kleene pronounced his last name /klay'nee/.

/klee'nee/ and /kleen/ are extremely common mispronunciations. His first name is /steev'n/, not /stef'n/.

His son, Ken Kleene <kenneth.kleene@umb.edu>, wrote: "As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father."

Regular Expressions Defined

- A regular expression (RE) is always defined with respect to a finite **alphabet** of symbols, Σ. The definition is inductive:
- Basis:
  - Any symbol in Σ is an RE.
  - The special symbol λ is an RE (often ε is used instead of λ).
  - The special symbol ^ is an RE.
- Induction step: If R and S are RE's, then so are:
  - RS
  - R | S
  - R*

Regular Expression Examples

- Take $\Sigma = \{0, 1\}$.
- Basis:
  - Any symbol in Σ is an RE: 0 1
  - The special symbol λ is an RE: λ
  - The special symbol ^ is an RE: ^
- Induction step: If R and S are RE's, then so are:
  - RS: 00 01 0001 1010 1(00 | 11)*0
  - R | S 00 | 11 0 | 1 | λ
  - R* 0* 01*0 (00 | 11)*

Meaning of Regular Expressions(1)

- Each regular expression R denotes a **language** (set of strings) L(R) over its alphabet:
- Basis:
  - A symbol σ in Σ denotes the language of one string of one letter: $L(\sigma) = \{\sigma\}$.
  - The special symbol λ denotes the empty string (no letters): $L(\lambda) = \{\lambda\}$.
  - The special symbol ˆ denotes the empty set (no strings): L(ˆ) = ˆ .

Meaning of Regular Expressions (2)

- Induction step: Suppose R and S are regular expressions and L(R) and L(S) have been defined. Then
  (concatenation of two strings)
  - $L(RS) = \{xy \mid x \in L(R) \text{ and } y \in L(S)\}$
  - $L(R \mid S) = L(R) \cup L(S)$
  - $L(R^*) = \{\lambda\} \cup L(R) \cup L^2(R) \cup L^3(R)$ ...

  where $L^k(R)$ means the language formed by concatenating k strings, each one from L(R).

Similarity to Grammar Rules

Suppose that we have a grammar in which
auxiliary symbol r derives the strings in L(R) and
auxiliary symbol s derives the strings in L(S).
Then:

- Adding t → r s would make t derive the strings in L(RS).
- Adding t → r | s would make t derive the strings in L(R | S).
- Adding t → {r} would make t derive the strings L(R*).

Note on Precedence in Regular Expressions

- It is common to omit parentheses.
- The binding order is:
  - * binds most tightly
  - juxtaposition is next
  - | binds most weakly

Examples of RE's, with Meanings

- 0101
  The set of one string "0101".
- 0101 | 1010
  The set of two strings, "0101" and "1010".
- 1(0101 | 1010)0
  The set of two strings, "101010" and "110100".
- 01*0
  The set of strings that begin and end with 0 and contain a continuous run of 1's (of length 0 or more).

Examples of RE's, with Meanings

- 0*1*
  The set of strings in which no 1 is followed by a 0.
- 0*1*0*1*
  The set of strings in which at most one 1 is immediately followed by a 0.
- 0*(100*)*
  The set of strings in which every one is followed by a 0.

Try These

- (0*10*1)*0*
- ((0 | 1)(0 | 1))*
- 0*10* | 1*01*
- (0*1*)*

Give Regular Expressions (over alphabet {0, 1}) for

- The set of strings with at most two 0's
- The set of strings with more than two 0's
- The set of strings in which 0's and 1's strictly alternate

Kleene's Remarkable Result

- The languages accepted by finite-state acceptors and the languages denoted by regular expressions are the same thing.

In other words:

- Part I: The language accepted by any finite-state acceptor can be expressed as a regular expression.
- Part II: For every regular expression, there is a finite state acceptor that accepts the language denoted by the expression.

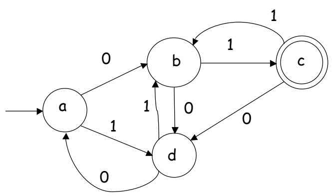
Proof of Part I

- The language accepted by any finite-state acceptor can be expressed as a regular expression.

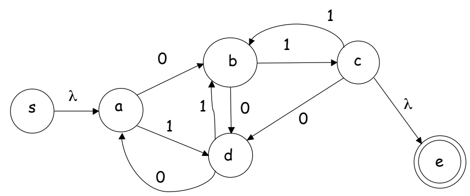
Proof of Part I:

- Given a finite-state acceptor, how to derive a regular expression?

DFA → RE Example

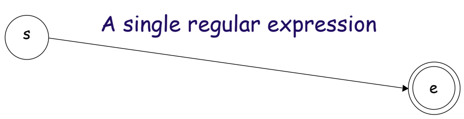


Step 1: Add Isolated Start and End States



View each arc as having a regular expression label, not just a single symbol.

Ultimate goal

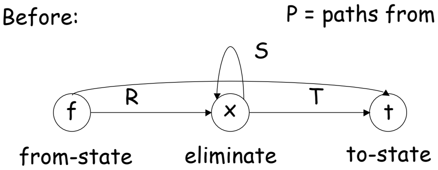


Elimination Step

- Pick a node for elimination.
- Add to the regular expression of each pair of nodes having a path through that node
  an additional expression component representing those paths.

Elimination Step Illustrated

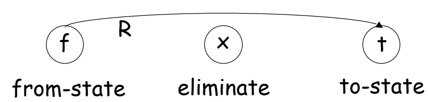
Before: P = paths from f to t



added paths from f to t:

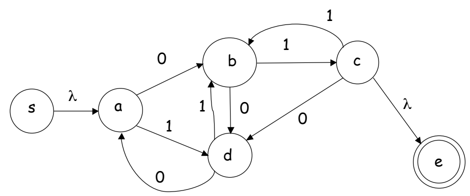
Elimination Step Illustrated

After: P | RS*T

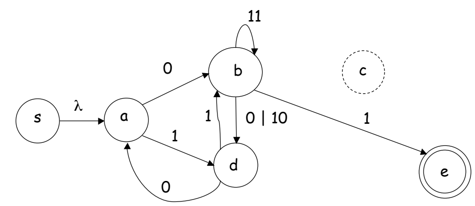


This has to be done for **all** pairs f, t **including** the case where f = t.

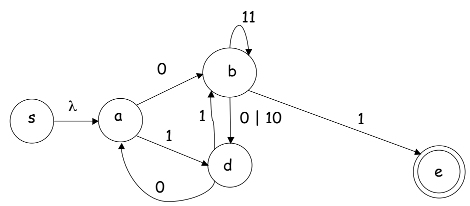
Eliminate c



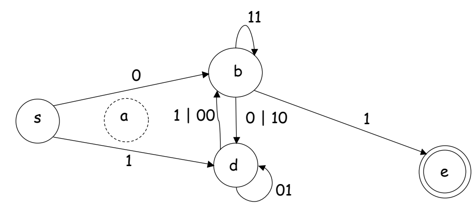
c Eliminated



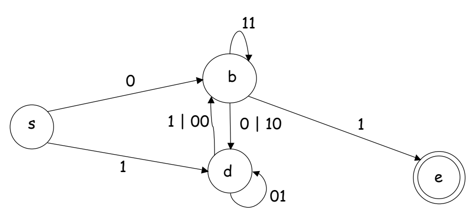
Eliminate a



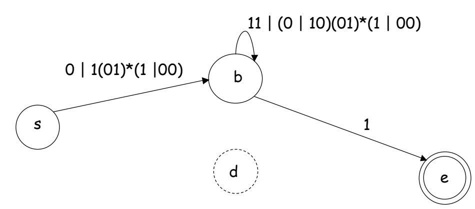
a Eliminated



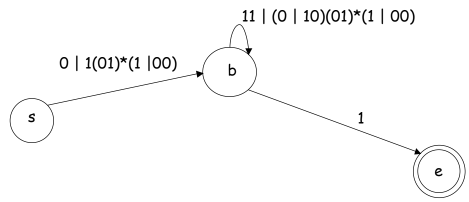
Eliminate d



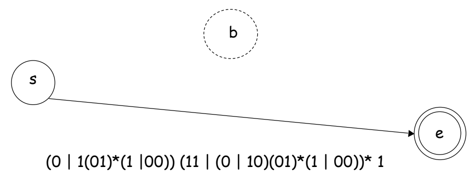
d Eliminated



Eliminate b



b Eliminated (= done)



Proof of Part II

- For every regular expression R, there is an FSA that accepts L(R), the language denoted by R.

Non-Deterministic FSAs

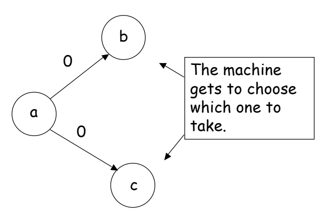
- An easy way to prove part II is to appeal to the idea of a non-deterministic finite-state acceptor (NFA):
  - Part IIa: For every regular expression R, there is an **NFA** that accepts L(R).
  - Part IIb: For every NFA N there is a (deterministic) finite-state acceptor that accepts L(N).

NFAs

- A non-deterministic finite-state acceptor (NFA) is a finite-state acceptor with free-choice of transitions:
  - A given state may have more than one transition leaving with the same symbol, or
  - A state may be left spontaneously via a λ transition.

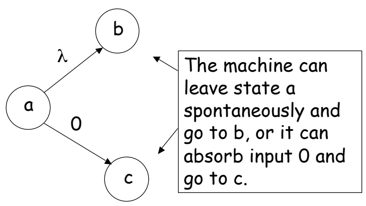
NFAs

- A given state may have more than one (or even no) transition leaving with a given symbol.



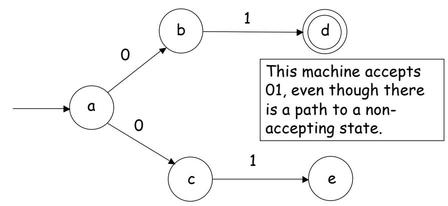
NFAs

- A state may be left spontaneously via a λ transition.



Acceptance Notion for NFAs

- An NFA **accepts** an input sequence iff there is ***some*** path from ***some*** initial state (an NFA can have more than one) to ***some*** accepting state.



Proof of Part IIa: Structural Induction

- Part IIa: For every regular expression R, there is an NFA that accepts L(R).
- This proof is by **structural induction** on the formation of regular expressions.
  - Basis:
    - Any symbol in Σ is an RE.
    - The special symbol λ is an RE.
    - The special symbol ^ is an RE.
  - Induction step: If R and S are RE's, then so are:
    - RS
    - R | S
    - R*

Proof of Part IIa (1)

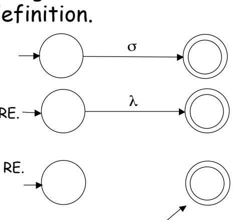
- We construct an accepting NFA for each RE introduced in the definition.
  - Basis:
    - Any symbol in Σ is an RE.

This is a string, not an alphabet symbol.

...ol λ is an RE.

This is neither a string nor an alphabet symbol.

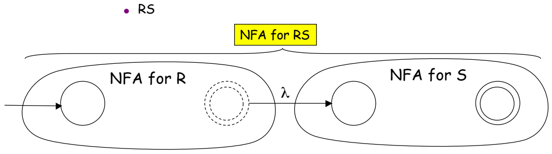
...an RE.



You can't get here from there.

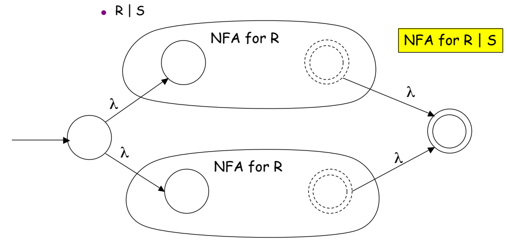
Proof of Part IIa (2)

- We construct an accepting NFA for each RE introduced in the definition.
  - Induction step: If R and S are RE's, then so are:
    - RS
    - R | S
    - R*
  - We assume that NFA's exist for R and S, and construct them for these three cases:
    - RS



Proof of Part IIa (3)

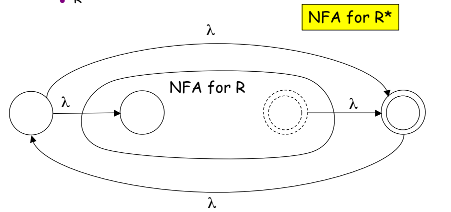
- We assume that NFA's exist for R and S, and construct them for these three cases:
  - R | S



Proof of Part IIa (4)

- We assume that NFA's exist for R and S, and construct them for these three cases:
  - R*

NFA for R*

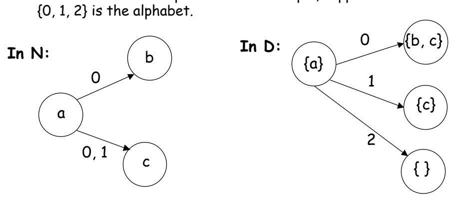


Proof of Part IIb (1)

- For every NFA N there is a (deterministic) FSA that accepts L(N).
- The idea is that for an NFA N we can construct a FSA D accepting L(N) by "simulating in parallel" all the **choices** the NFA could make. An input sequence is accepted iff **any** of those choices led to acceptance in N.

Proof of Part IIb (2)

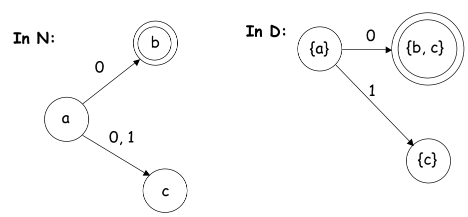
- To simulate an NFA, we construct D to have as its states **subsets** of the states of N. The transitions of D emulate all transitions for N "in parallel". For example, suppose that {0, 1, 2} is the alphabet.



Definition of f, the state transition for D:
$f(S, \sigma) = \{q' \mid (\exists q \in S)\ q \xrightarrow{\sigma} q' \text{ in N}\}$

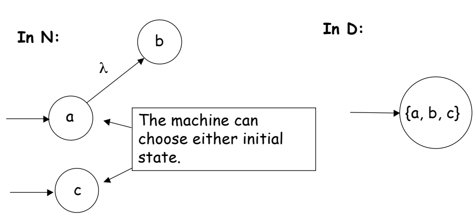
Proof of Part IIb (3)

- An **accepting** state in D is any that has an accepting state of N as a member.

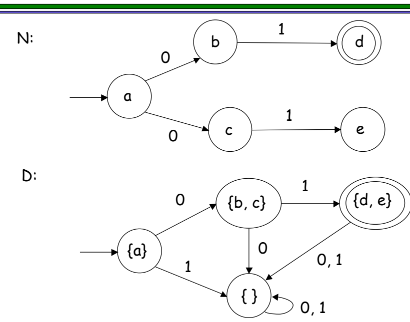


Proof of Part IIb (4)

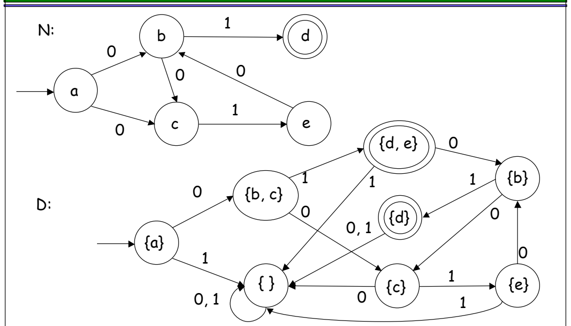
- The **initial** state in D is the set of all states reachable from ***some*** initial state in N by the empty sequence (i. e. including λ transitions)



The Complete Construction for a Simple Example



A More Complex Example with a Loop

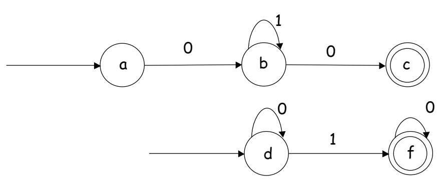


This Completes the Proof of Kleene's Theorem

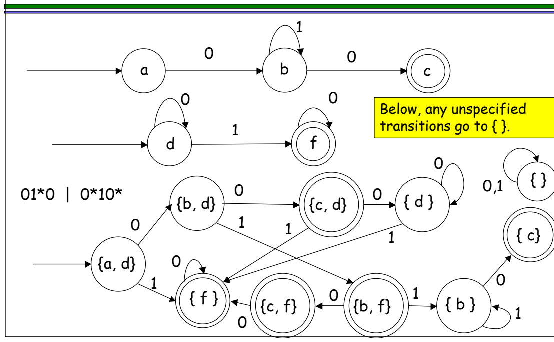
- We now know that the following are equivalent:
  - L is a language denoted by some regular expression.
  - L is a language accepted by an NFA.
  - L is a language accepted by an FSA.

Example: Regular Expression to FSA (1)

- Construct an FSA for the RE
  01*0 | 0*10*
- By inspection we can do NFA's for 01*0 and 0*10*:



Example: Regular Expression to FSA (2)



"Combination Lock" Type Problems

- Consider a sequential combination lock without an explicit reset. Regardless of what came before, if the the last digits entered correspond to the combination, the lock opens.
- Example combination:
  0 1 0 1 01 1
- Design a DFA for this lock

Regular Expressions in Everday Practice:

e.g. Unix **egrep**

used for searching for ***lines containing*** matching strings in files

- Do, e.g. **`man egrep`** to get this information on a Unix box:
  - Most single characters match themselves (exceptions: `. * [ ] \ ^ $`)
  - . matches any character, except new-line
  - ^ matches beginning of line (must occur first)
  - $ matches end of line (must occur last)
- Examples:
  - **`egrep 'elle'`** *filename*
  - **`egrep 'll.*ll'`** *filename* .* is like Σ*
  - **`egrep 'll$'`** *filename*
  - **`egrep '^Ll'`** *filename*
  - **`egrep 'aa|bb|cc'`** *filename*
  - **`egrep '(aa|bb)c'`** *filename*

Regular Expressions in Everday Practice:
Iteration constructs

- * The preceding item will be matched zero or more times.
- \+ The preceding item will be matched one or more times.
- ? The preceding item is optional and matched at most once.
- {n} The preceding item is matched exactly n times.
- {n,} The preceding item is matched n or more times.
- {n,m} The preceding item is matched at least n times, but not more than m times.
- etc.

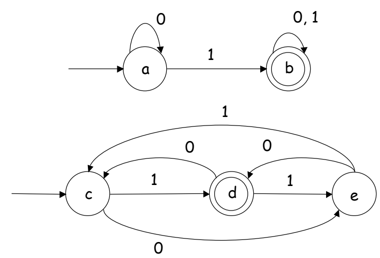
Closure Properties

- Based on their connection to regular expressions, regular languages are closed under ∪, concatenation, *.
- They are also closed under:
  - complementation Σ^*-R
  - intersection ∩
  - substitution (of arbitrary strings for letters)

Product Construction

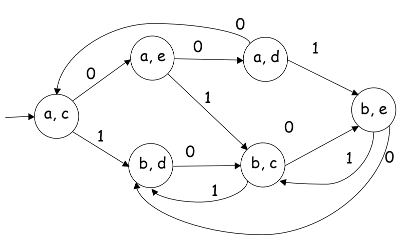
- Similar to subset construction.
- Can be used to show closure under ∩, ∪, -, and any other Boolean combination.
- The product of two DFA's is a DFA that simulates both in tandem.
- Its accepting states are some combination of accepting states of the two DFA's.

Product Construction



State set will be {a, b} x {c, d, e}.

Product Construction



Accepting states depending on which operation is desired.