Deterministic Finite Automata

COMP1600 / COMP6260

Victor Rivera Dirk Pattinson
Australian National University

Semester 2, 2021

The Story So Far ...

Logic.

- language and proofs to speak about systems *precisely*
- useful to *express properties* and *do proofs*

Functional Programs

- establish *properties* of functional programs
- main tool: (structural) induction

Imperative Programs.

- again: focus on *properties* of programs
- main tool: Hoare Logic

Q. Is there a *general* notion of computation? That encompasses both?

First Shot: Your Laptop

Abstract Characteristics.

- can do computation
- has memory – a finite amount
- has (lots of) internal states

From Laptops to Formal Models

Concrete (your laptop)

- realistic (it exists!)
- complex
- hard to analyse

Abstract (mathematical model)

- exists only as a model
- simple
- easy to analyse

Q. What is a "*good*" simple model of computation?

- should match what really exists (possibly by a long shot)
- should be *conceptually simple*

First Answer: Finite State Automata

Basic Components.

- internal states – finitely many
- state transitions – triggered by reading input
- simplifying assumption: just *one* output: yes/no

Data.

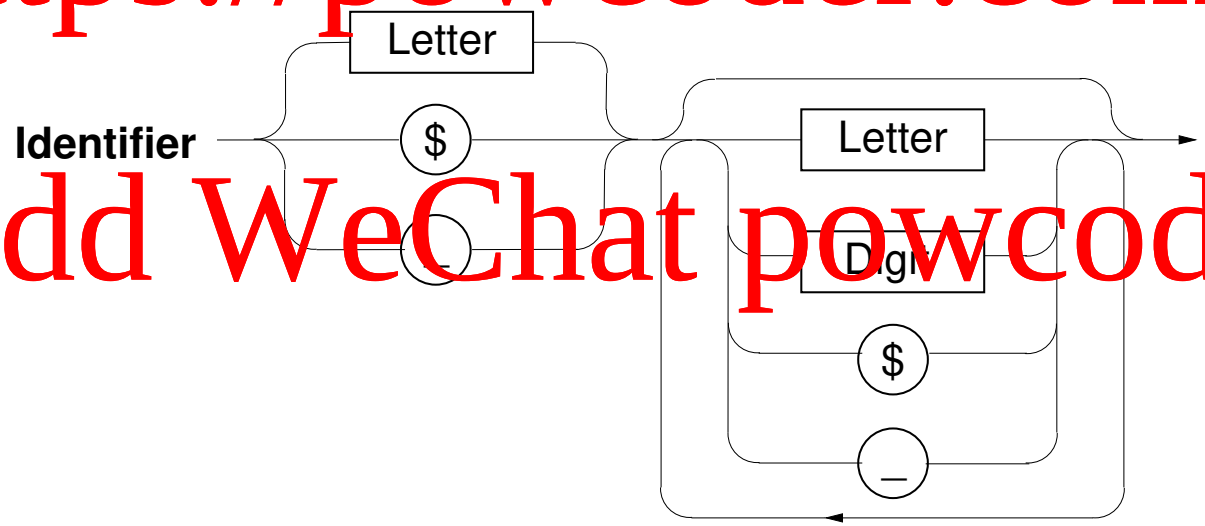
- basic input: strings (what you type in, text/xml file)
- characters: drawn from *finite* set (alphabet)

Example: Java Identifiers

From Oracle's Java Language Specification.

> *An identifier is a sequence of one or more characters. The first character must be a valid first character (letter, $, _) in an identifier of the Java programming language, hereafter in this chapter called simply "Java". Each subsequent character in the sequence must be a valid nonfirst character (letter, digit, $, _) in a Java identifier.*

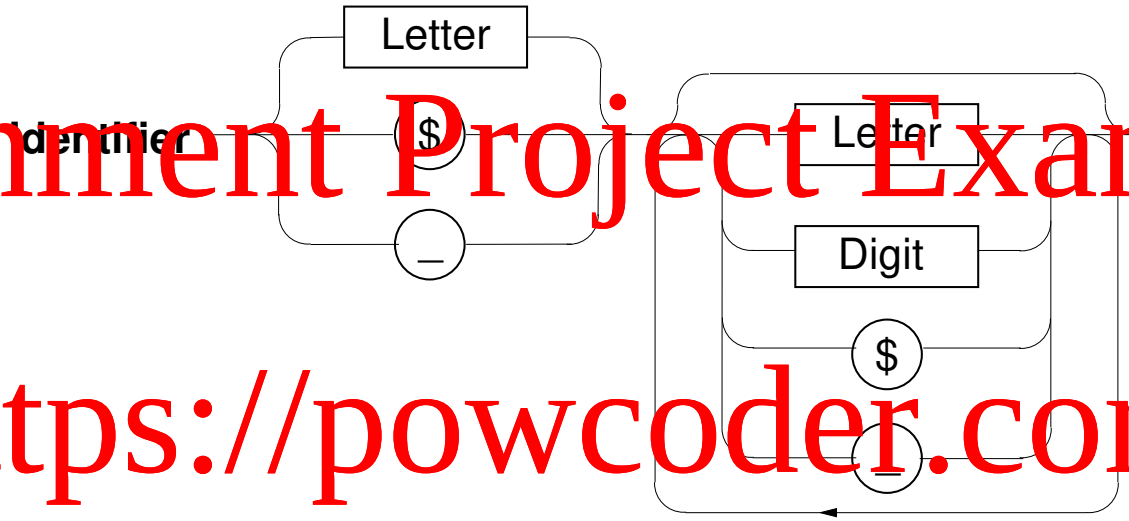
Graphical Specification



Q. Can you "see" a machine that *recognises* Java identifiers?

Java Identifiers

Example: Main Components



Data.

- drawn form a finite alphabet (unicode, or ASCII)

Control.

- "yes" if I can get from the left to the right, "no" otherwise
- have *states* after taking a transition (implicit in diagram)

Computational Problem with yes/no answer:

- it a given sequence of characters a valid Java identifier?

Preview.

Next two weeks. Finite Automata

- start with simplest model: finite automata
- relate to regular languages, non-determinism
- conclusion: finite automata "too simple"

The week after. Pushdown automata

- like finite automata, but some more memory
- useful for e.g. specifying syntax of programming languages
- still "too simple" for general computation

Then. Turing machines

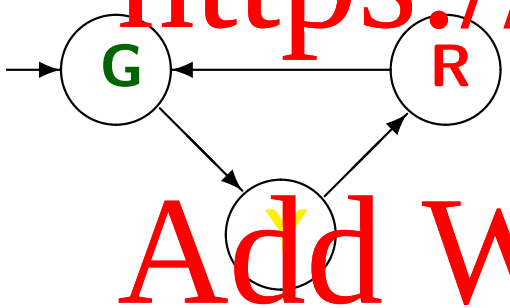
- *The* most widely accepted model of computation
- *infinite* memory
- idea: buy another hard disk whenever your computation runs out of memory
- *limits* of what can be computed

Finite State Automata: First Example

The simplest useful abstraction of a **"computing machine"** consists of:

- A fixed, finite set of **states**
- A **transition relation** over the states

Example a traffic light FSA has 3 states:



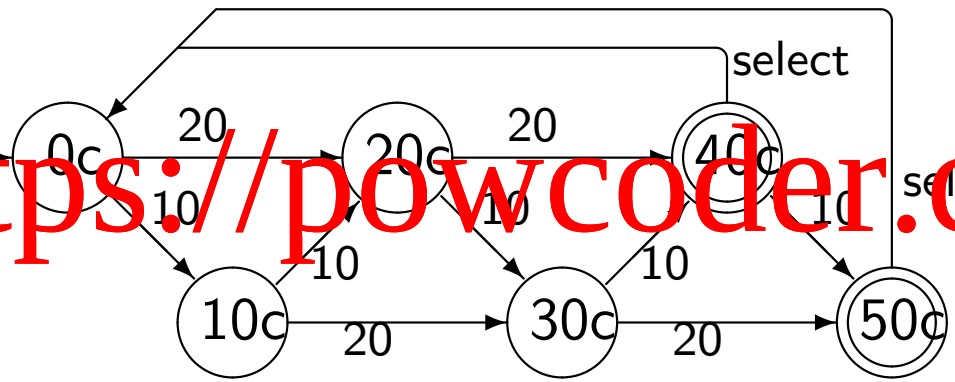
G names state in which light is green.
Y names state in which light is yellow.
R names state in which light is red.

System designs are often in terms of state machines.

Second Example: Vending Machine

Operation

- accept 10c and 20c coins
- delivers if it has received at least 40c and selection is made



Note.

- transitions are *labelled*
- new ingredient: *final states* (doubly circled)

Computation. Sequences of actions (labels) from initial to final state.

Language Examples

Main Idea.

- input: a string over a fixed character set
- operation: transitions labelled with characters
- output: yes if in final state after reading the input

More Generally.

- Setup: Fix a finite set of characters (an alphabet)
- Problem: A set of strings (called *language*) that are "valid" or "good"
- Task: decide computationally which strings are "good"

Example Languages.

1. A finite set.

$$\{a, aa, ab, aaa, aab, aba, abb\}$$

2. Palindromes consisting of bits (0,1):

$$\{0, 1, 00, 11, 010, 101, 000, 111, 0110, ...\}$$

Languages in this sense are called *formal* languages.

Terminology

Alphabet.

A finite set (of symbols). Usually denoted by Σ.

Strings over an alphabet Σ

finite sequence of characters (elements of Σ), can be the empty sequence. E.g. for $\Sigma = \{a, b, c\}$, $ababc$ is a string over Σ.

Languages over alphabet Σ

are just sets of strings over Σ.

Sentences of the language

just another name for the elements (strings) of the language.

Notation:

- Σ^* is the set of all strings over Σ.
- Therefore, every language with alphabet Σ is some **subset** of Σ^*.

Automata

First Model of Computation. Deterministic Finite Automata

- solve computational problem: given string s, is s accepted?

Basic Ingredients. (see e.g. traffic light and vending machine example)

- The **alphabet** of a DFA is a finite set of **input tokens** that an automaton acts on.
- a DFA consists of a finite set of **states** (a primitive notion)
- One of the states is the **initial** state — where the automaton starts
- At least one of the states is a **final** state

- A **transition function** *(next state function):*

$$State \times Token \rightarrow State$$

Recurring Theme

Diagrammatic Notation.

- useful for humans
  - e.g. the transition diagram of the vending machine

Mathematical Notation.

- useful for formal manipulation (e.g. proving theorems)
- useful for computer implementation

Glue between Diagrams and Maths

- both notions convey *precisely* the same information
- crucial: being able to switch back and forth!

Formal Definition of DFA

A Deterministic Finite State Automaton (DFA) consists of five parts:

$$A = (\Sigma, S, s_0, F, N)$$

- an input **alphabet** Σ, the set of tokens
- a set of **states** S
- an **"initial"** state $s_0 \in S$ (we start here)
- a set of **"final"** states $F \subseteq S$ (we hope to finish in one of these)
- a **transition function** $N : S \times \Sigma \to S$

Aside. Having a transition *function* is what makes the automaton deterministic.

Finite State Automata as String Acceptors

Idea. A finite state automaton

- works on *strings* over an alphabet Σ
- determines which strings in Σ are "good" (accepted) and which strings are "bad" (rejected)

Acceptance Informally. Let $A = (\Sigma, S, s_0, F, N)$ be a DFA. Then A *accepts* the string $w = a_1 a_2 \dots a_n$ if there is a sequence of states

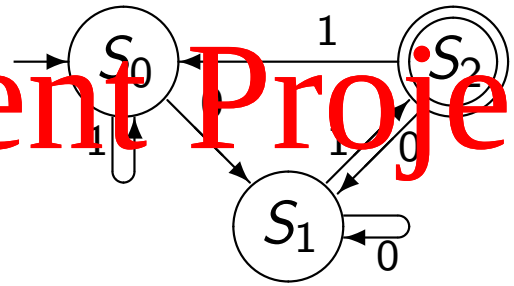
$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n$$

where s_0 is the starting state, $s_n \in F$ is an accepting state, and $s \xrightarrow{a} t$ if $\delta(s, a) = t$.

Informally. Run the automaton from the starting state, move states according to the individual letters of the word, and accept if you end up in a final state.

Example 1

As a diagram.



In Mathematical Notation.

- Alphabet - $\{0, 1\}$
- States - $\{S_0, S_1, S_2\}$
- Initial state - S_0
- Final states - $\{S_2\}$
- Transition function (as a table) -

	0	1
S_0	S_1	S_0
S_1	S_1	S_2
S_2	S_1	S_0

Q1. Which strings are accepted by this automaton?
Q2. What changes if we re-name the states?

Example 1, ctd

Recall. $N : S \times \Sigma \to S$ is the transition function.

	0	1
S_0	S_1	S_0
S_1	S_1	S_2
S_2	S_1	S_0

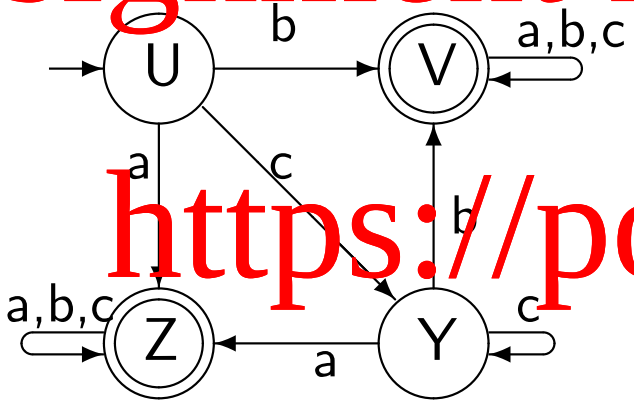
Single Steps of the automaton

- $N(S_0, 0)$ is the state that the automation transitions to from state S_0 reading letter 0.
- Here: $N(S_0, 0) = S_1$.

Multiple Steps of the automaton

- $N(N(S_0, 0),\ 1)$ is the state of the automation when starting in S_0 and reading first 0, then 1.
- Here: $N(N(S_0, 0),\ 1) = S_2$.

Example 2



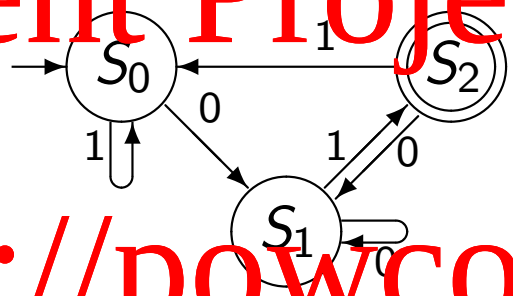
	a	b	c
→U	Z	V	Y
⊙V	V	V	V
Y	Z	V	Y
⊙Z	Z	Z	Z

(the table carries the same information as the diagram)

Q. What is the language of this automaton?

Eventual State Function

Revisit example 1:



- Input 0101 takes the DFA from S_0 to S_2,
  Input 1011 takes the DFA from S_1 to S_0, etc
- A complete list of such possibilities is a function from a given state and a string to an 'eventual state.'

This is the idea of **Eventual State Function.**

Eventual State Function — Definition

Definition. Let A be a DFA with states S, alphabet Σ, and transition function N.

The *eventual state function* for A is of type

$$N^* : S \times \Sigma^* \to S$$

and is defined inductively by:

$$N^*(s, \epsilon) = s \quad \text{(N1)}$$

$$N^*(s, x\alpha) = N^*(N(s, x), \alpha) \quad \text{(N2)}$$

Or in Haskell, where strings are lists of elements of type `Sigma`

```
%   n :: State -> Sigma -> State          -- given
  nstar :: State -> [Sigma] -> State
  nstar s [] = s
  nstar s (a:as) = nstar (n s a) as
```

Informally. $N^*(s, w)$ is the state A reached by starting in state s and reading string w.

An Important (but Unsurprising) Theorem about N^*

Theorem. For all states $s \in S$ and for all strings $\alpha, \beta \in \Sigma^*$:

$$N^*(s, \alpha\beta) \;=\; N^*(N^*(s, \alpha), \beta)$$

Proof by induction on the length of α.

Base case: $\alpha = \epsilon$

$$\text{LHS} = N^*(s, \epsilon\beta) = N^*(s, \beta)$$

$$\text{RHS} = N^*(N^*(s, \epsilon), \beta)$$
$$= N^*(s, \beta) = \text{LHS} \qquad \text{(by (N1))}$$

Proof ctd: Step case:

Step Case. Show that $N^*(s,(x\alpha)\beta) = N^*(N^*(s,x\alpha),\beta)$

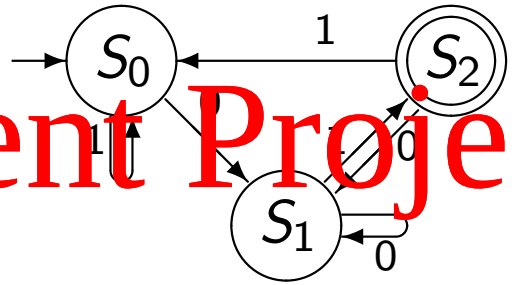
$$\begin{aligned}
\text{LHS} &= N^*(s,(x\alpha)\beta) \\
&= N^*(s,x(\alpha\beta)) \\
&= N^*(N(s,x),\alpha\beta) && \text{(by (N2))} \\
&= N^*(N^*(N(s,x),\alpha),\beta) && \text{(by IH)}
\end{aligned}$$

$$\begin{aligned}
\text{RHS} &= N^*(N^*(s,x\alpha),\beta) \\
&= N^*(N^*(N(s,x),\alpha),\beta) && \text{(by (N2))}
\end{aligned}$$

Corollary — when β is a single token

$$N^*(s,\alpha y) \;=\; N(N^*(s,\alpha),y)$$

Example



$$\begin{aligned} N^*(S_1, 1011) &= N^*(N(S_1, 1), 011) \\ &= N^*(S_2, 011) \\ &= N^*(S_1, 11) \\ &= N^*(S_2, 1) \\ &= N^*(S_0, \epsilon) \\ &= S_0 \end{aligned}$$

Language of an Automaton, Revisited

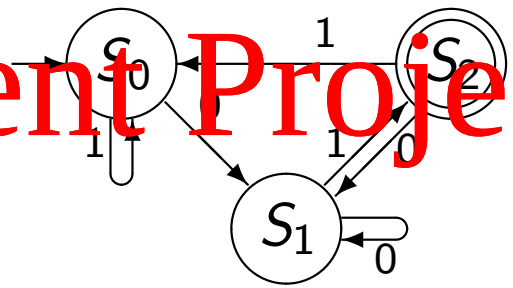
Acceptance, with eventual states. Let $A = (\Sigma, S, s_0, F, N)$ be an DFA and w be a string in Σ^*.
Then w is *accepted* by A if

$$N^*(s_0, w) \in F$$

Q1. How does this compare with the earlier notion of acceptance?
Q2. How can we prove that both are equivalent?

Example 1 again

A_1:

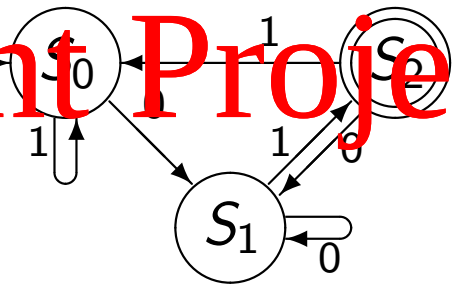


Q. Which strings are accepted?

- e.g. 0011101 takes the machine from state S_0 through states S_1, S_1, S_2, S_0, S_0, S_1 to S_2 (a final state).
- $N^*(S_0, 0011101) = N^*(S_1, 011101) = N^*(S_1, 11101) = \ldots N^*(S_1, 1) = S_2$
- others: 01, 001, 101, 0001, 0101, 00101101 ...

Example 1 (ctd.)

A_1:



Accepted Strings.

01, 001, 101, 0001, 0101, 00101101 . . .

Strings that are not accepted.

ϵ, 0, 1, 00, 10, 11, 100 . . .

Q. What do the accepted strings have in common? How do we justify this?

Proving an Acceptance Predicate — in General

Our Claim. The automaton A accepts precisely the strings that are elements of the language $L = \{w \in \Sigma^* \mid P(w)\}$.

(P is sometimes called an *acceptance predicate*.)

Proof Obligations.

1. Show that any string satisfying P is accepted by A.
2. Show any string accepted by A satisfies P.

Proving an Acceptance Predicate for A_1

Proof obligation 1:

If a string ends in 01, then it is accepted by A_1. That is:

$$\text{For all } \alpha \in \Sigma^*,\ N^*(S_0, \alpha 01) \in F$$

Proof obligation 2:

If a string is accepted by A_1, then it ends in 01. That is:

$$\text{For all } w \in \Sigma^*,\ \text{if } N^*(S_0, w) \in F \text{ then } \exists \alpha \in \Sigma^*.\ w = \alpha 01$$

Part 1: $\forall \alpha \in \Sigma^*,\ N^*(S_0, \alpha 01) \in F$

Lemma:

$\forall s \in S,\ N(s, 01) \in S_2$

Proof by cases:

$$N^*(S_0, 01) = N^*(S_1, 1) = S_2$$
$$N^*(S_1, 01) = N^*(S_1, 1) = S_2$$
$$N^*(S_2, 01) = N^*(S_1, 1) = S_2$$

So, by the "append" theorem above,

$$N^*(S_0, \alpha 01) = N^*(N^*(S_0, \alpha), 01) = S_2 \square$$

Part 2: $N^*(S_0, w) = S_2 \implies \exists \alpha.\ w = \alpha 01$

Proof. Suppose $N^*(S_0, \alpha xy) = S_2$.

By corollary to append-theorem (case of single token):

$$N(N^*(S_0, \alpha x), y) = S_2$$

By the definition of N, y must be 1 and $N^*(S_0, \alpha x)$ must be S_1.

Similarly,

$$N(N^*(S_0, \alpha), x) = S_1$$

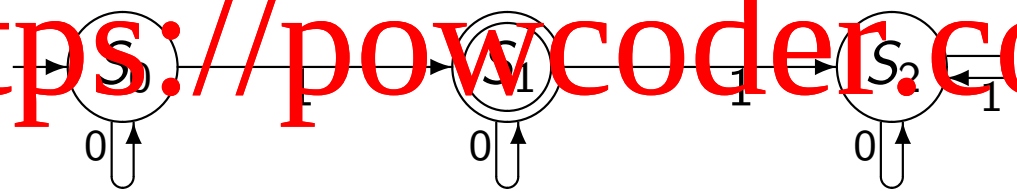
and x is 0, again by the definition of N.

□

Another Example

What language does this DFA accept?

SOB:



Answer for *SOB*

SOB accepts the language of bitstrings containing **exactly one 1-bit.**

Proof obligations:

- Show that if a bitstring contains exactly one 1-bit then it is accepted by *SOB*.
- Show that if a string is accepted by *SOB* it contains exactly one 1-bit.

SOB:

Mapping to Mathematics

Expressed mathematically, the main conclusion is

$$L(SOB) = \{w \in \Sigma^* \mid w = 0^n 1 0^m\}$$

The two subgoals are:

1. If $w = 0^n 1 0^m$ then $N^*(S_0, w) = S_1$
2. If $N^*(S_0, w) = S_1$ then $w = 0^n 1 0^m$.

For this DFA the phrase "w is accepted by SOB" is captured by the expression $N^*(S_0, w) = S_1$.

Proving these subgoals

The first subgoal follows immediately from the following two lemmas, which are easily proved by induction:

Lemma 1: $\forall n \geq 0.\ N^*(S_0, 0^n) = S_0$

Lemma 2: $\forall n \geq 0.\ N^*(S_1, 0^n) = S_1$

Therefore

$$
\begin{aligned}
& N^*(S_0, 0^n 10^m) \\
=& N^*(N^*(S_0, 0^n), 10^m) && \text{(by apppend Theorem)} \\
=& N^*(S_0, 10^m) && \text{(by Lemma 1)} \\
=& N^*(N^*(S_0, 1), 0^m) && \text{(by def. } N^*\text{)} \\
=& N^*(S_1, 0^m) && \text{(by def. } N\text{)} \\
=& S_1 && \text{(by Lemma 2)}
\end{aligned}
$$

The second subgoal, stated more formally as

$$\forall w:\ N^*(S_0, w) = S_1 \implies \exists n, m \geq 0.\ w = 0^n 10^m$$

can be proved in a similar fashion to Example 1 on earlier slides.

Limitations of FSAs

Q. Is an FSA a "good" model of computation?

- Suppose we have a program P that always terminates and outputs "yes" or "no" for every input string
- Is there an FSA that accepts precisely the strings for which P says "yes"?

Technical Analysis. Properties of languages accepted by a DFA.

A very important example: $L = \{ a^n b^n \mid n \in \mathbb{N} \}$

- $L = \{\epsilon, ab, aabb, aaabbb, a^4b^4, a^5b^5, \ldots\}$
- **Claim.** There is no FSA that recognises this language.

(because an FSA's memory is limited.)

Q. Given the claim above, are FSA's *realistic* models of computation?

Proof of Claim

Proof by contradiction.

Suppose A is an FSA that accepts L. That is $L = L(A)$.

Then each of the following are states of A:

$$N^*(S_0, a), N^*(S_0, a^2), N^*(S_0, a^3) \ldots$$

But A only has finitely many states, so some state must repeat:

There are distinct i and j such that $N^*(S_0, a^i) = N^*(S_0, a^j)$.

- that is, the automaton *cannot* tell a^i and a^j apart.

Proof by contradiction (ctd)

Since $a^i b^i$ is accepted, we know

$$N^*(S_0, a^i b^i) \in F$$

By the *append* theorem

$$N^*(N^*(S_0, a^i), b^i) = N^*(S_0, a^i b^i) \in F$$

Now, since $N^*(S_0, a^i) = N^*(S_0, a^j)$

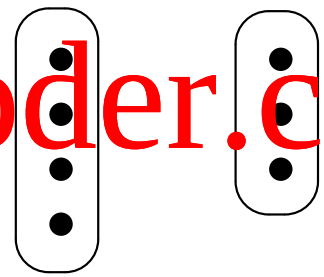
$$N^*(N^*(S_0, a^j), b^i) = N^*(S_0, a^j b^i) \in F$$

So $a^j b^i$ is accepted by A but $a^j b^i$ is not in L, contradicting the initial assumption.

Pigeon-Hole Principle

The proof used the **pigeon-hole principle**:

No function from one set to a smaller finite set can be one-to-one.



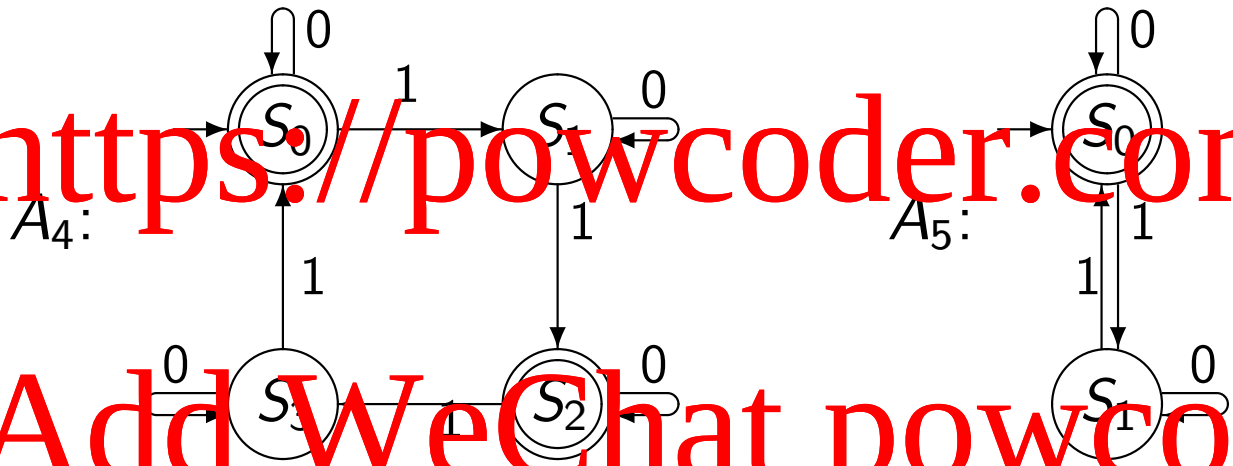
(Finiteness is not really necessary — no function from one set to another with smaller *cardinality* can be one-to-one.)

"You cannot fit $n + 1$ pigeons into n holes"

Equivalence of Automata

Two automata are said to be **equivalent** if they accept the same language.

Example:



Q. Can FSAs be simplified? is there an equivalent FSA with *fewer states?*

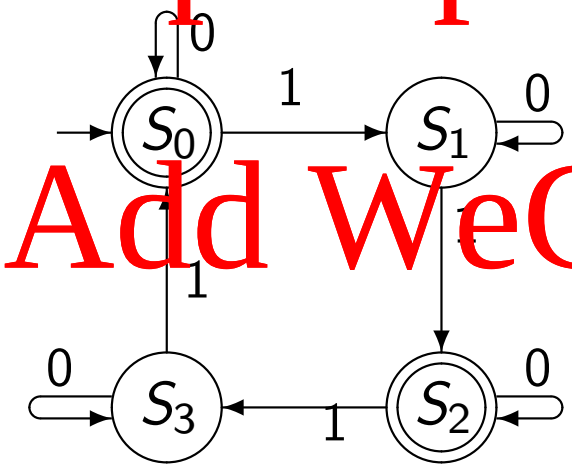
Equivalence of States

Two states S_j and S_k of an FSA are equivalent if, for all input strings w

$N^*(S_j, w) \in F$ if and only if $N^*(S_k, w) \in F$

Example In A_4, S_2 is equivalent to S_0 and S_1 is equivalent to S_3.

A_4:



Elimination of Equivalent States

Assumptions

- $A = (\Sigma, S, S_0, F, N)$ is an FSA
- S_k and S_j be equivalent
- $S_k \neq S_0$ (don't eliminate the initial state!)

Elimination of S_k from A: new automaton $A' = (\Sigma, S', S_0, F', N')$

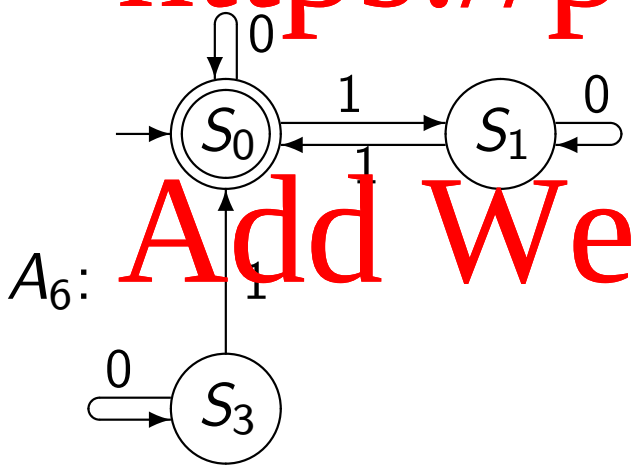
- S' is S without S_k
- F' is F without S_k

- $N'(s, w) =$ (if $N(s, w) = S_k$ then S_j else $N(s, w)$)

Example

Since $S_2 \equiv S_0$ in A_4, let's eliminate S_2

- New set of states is $\{S_0, S_1, S_3\}$
- New set of final states is $\{S_0\}$
- New transition function is:



	0	1
S_0	S_0	S_1
S_1	S_1	S_0
S_3	S_3	S_0