Finite State Automata

and

Regular Languages

COMP2600 — Formal Methods for Software Engineering

Jeremy Dawson

Australian National University

Semester 2, 2011

Introduction

Topics so far have mostly been about ***semantics*** — what progams *“mean.”*

Formal languages and automata are (mostly) about ***syntax*** — what progams *“look like.”*

There are two main aspects:

- How to **specify** syntax (Chomsky grammars)
- How to **recognise** syntax (Automata)

Two sides of the same coin — there are algorithms for deriving automata that exactly correspond to grammars. There are widely used standard language processing tools based directly on these concepts (YACC, Lex, etc.).

(This material also leads in to Turing machines and the varying power of different notions of computability, so there is also a semantic aspect.)

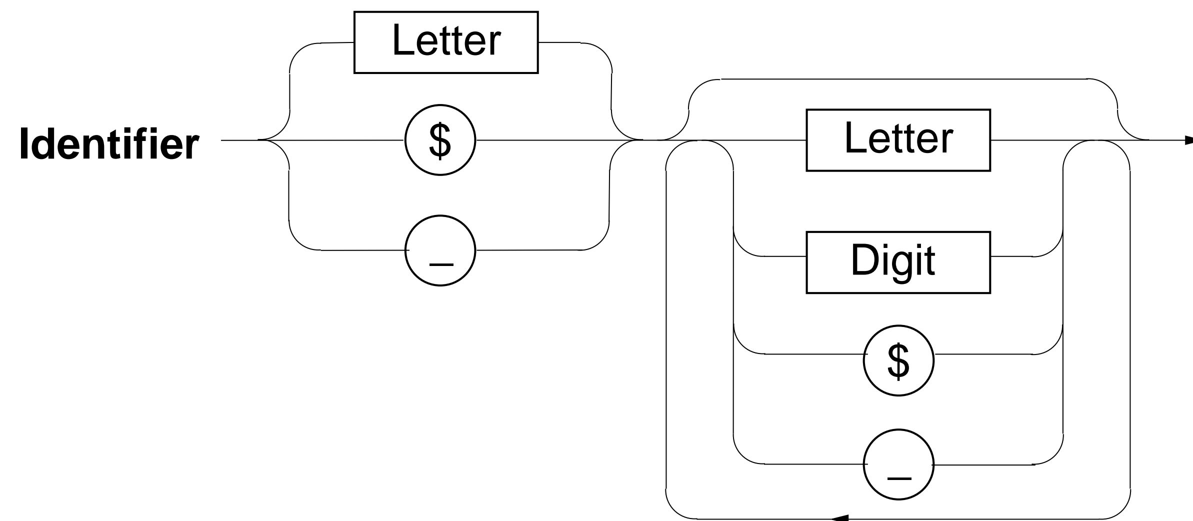
Background

You have probably already encountered:

- Regular expressions
- Abstract machines:
  - Java virtual machine
  - PeANUt
- Notations for specifying the syntax of programming languages
  - Railroad diagrams
  - Backus-Naur Form

Railway Diagrams?

The following diagram clearly specifies the syntax of Java identifiers.



Major Topics Covered

- Finite state machines
- determinism & non-determinism
- Regular expressions and corresponding finite state machines
- Pushdown automata, context free grammars
- Classification of languages and corresponding machines

Formal Language Theory

- Motivation:

  English sentences obey rules;

  Programs do too!

- A decent theory can help:

  terminology

  understanding

  design of tools

- Primitive notions:

  Vocabulary

  Legal sentence

  Language.

Applications of Language Theory

- Linguistics (natural languages):

  words, punctuation

  sentences, questions

  all legal utterances belong to language

  connections with psychology

  dictionaries

- Java, Haskell, etc:

  identifiers & constants, reserved words, punctuation

  syntactically correct programs

  compilers, text-based interactions (e.g. spreadsheet formulas)

Language Examples

Formal Language Theory gets applied to natural and artificial languages and to programming languages, but they are too complicated for beginning studies. More suitable are simple languages like these:

1. A finite set.

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

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

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

In each case the language is the set of strings. The first is a finite language and the second is an infinite language.

Terminology

- The **alphabet** or **vocabulary** of a formal language is a set of **tokens** (or **letters**). It is usually denoted Σ.
- A **string** over Σ is a *sequence* of tokens, or the null-string ε.
  For example, if $\Sigma = \{a, b, c\}$, then $ababc$ is a string over Σ.
- A **language** with alphabet Σ is some set of strings over Σ.
- The strings of a language are called the **sentences** of the languages.

Notation:

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

Practical parsing in two stages

Analysing a string, such as program source code, is often done in stages, eg

- **Scanning, or lexical analysis:** Splitting the input into "words" (identifiers, reserved words of the language, symbols such as ',', '(', ')') : the tokens are single characters
- **Parsing:** Making sense of the resulting sequence of "words": now the tokens are these "words"

```
data Tree a = Nul | Node a ( Tree a ) ( Tree a )
```

The strings of the "scanning" phase, which are the tokens of the "parsing" phase, are underlined.

Specifying Languages

Languages may be defined by various means.

For example,

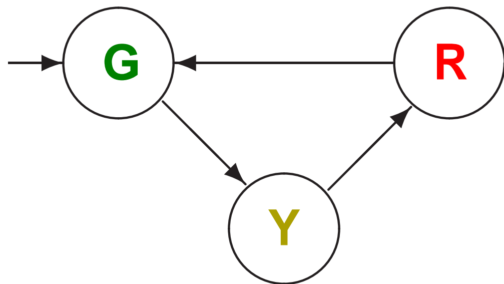
- As an explicit set (members are listed)
- As a set, by giving a predicate
- Algebraically (e.g. regular expression)
- Grammars (e.g. regular or context-free)
- Recognisers (automata)

Finite State Automata

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:

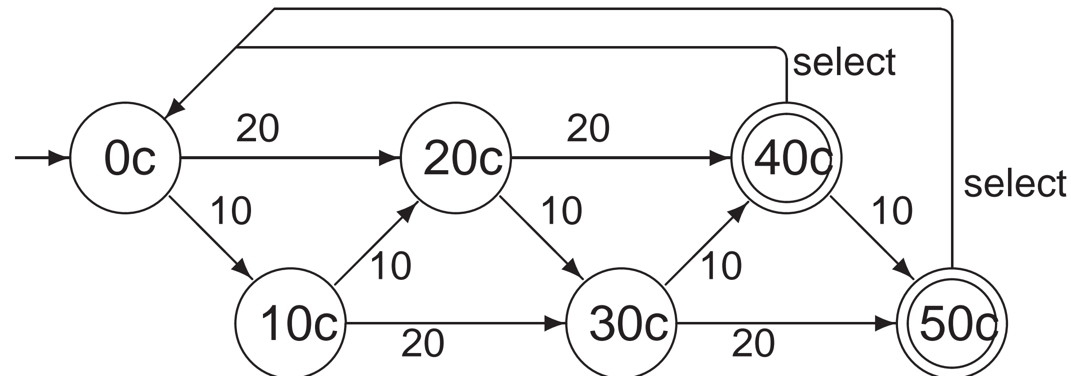


G names state in which light is green.
Y names state in which light is yellow.
R names state in which light is red.

System designs are often in terms of state machines.

Example: Vending Machine

Imagine a vending machine which (1) accepts 10c and 20c coins;
(2) delivers when it gets at least 40c and a selection is made.



Note that the ***transitions*** are labelled with important information.

In COMP2600 we are mainly interested in the relation between automata and languages.

The Vending Machine ctd

- You start at the state “0c”, indicated by the $\rightarrow$ at the left
- the alphabet $\Sigma = \{10, 20, \text{select}\}$
- at the “40c” or “50c” states (circled), you have credit for a purchase
- what strings of tokens (starting at the “0c”) leave you in a circled state?
- these strings are the language defined by the automaton

On Notation . . .

Diagrams are intuitive and useful for (most) humans.

Mathematical (text) descriptions — conveying the same information — are suitable for formal manipulations and for input to programs.

Terminology

We will study **deterministic** finite automata (**DFA**) first.

- 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):*

$$State \times Token \;\rightarrow\; State$$

Formal Definition of DFA

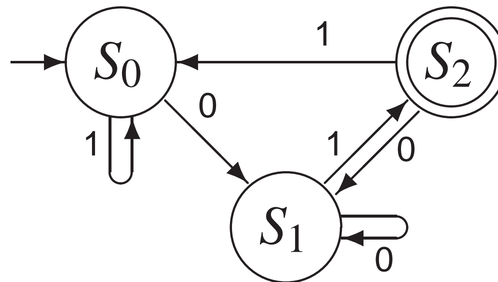
A Deterministic Finite State Automaton (DFA) is completely characterised by five items:

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

- 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 $N : S \times \Sigma \; \rightarrow \; S$

(We will see later that N being a ***function*** is the reason the automaton is deterministic.)

Example 1



- 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

(Note that the actual state names are irrelevant.)

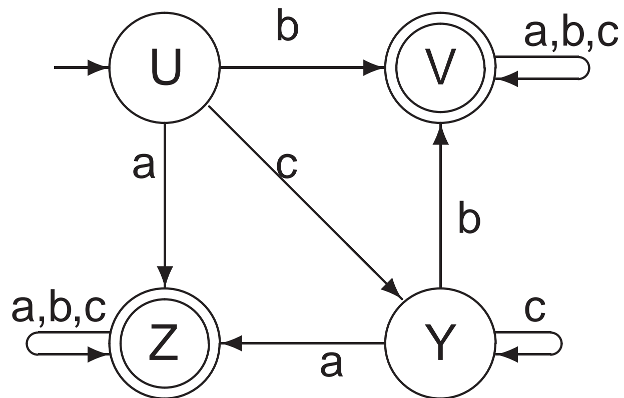
Example 1, ctd

If N is this transition function then we write $N(S_0, 0)$ when we want to refer to the state that the above automaton goes to when it starts in S_0 and absorbs a 0.

From the definition, $N(S_0, 0) \ = \ S_1$.

Similarly, we can talk about the state that the automaton goes to with input 01, namely $N(N(S_0, 0), \ 1)$ which we show is S_2.

Example 2

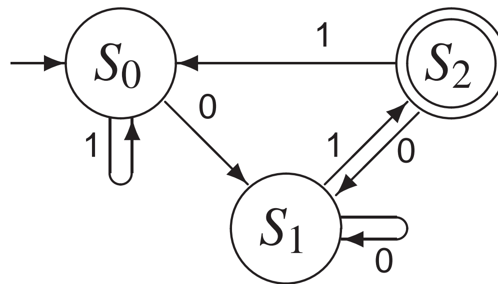


	a	b	c
$\rightarrow$ U	Z	V	Y
$\odot$ V	V	V	V
Y	Z	V	Y
$\odot$ Z	Z	Z	Z

The table on the right carries the same information as the diagram on the left.

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 illustrates the idea of ***Eventual State Function.***

Eventual State Function — Definition

Suppose A is a DFA with states S, alphabet Σ, and transition function N. The eventual state function for A is

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

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

N^* can be defined inductively:

$$N^*(s,\varepsilon) \;=\; s \qquad \text{(N1)}$$

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

For Haskell afficionados:

$$N^* \;=\; \texttt{uncurry}(\texttt{foldl}(\texttt{curry}\; N))$$

An Important (but Unsurprising) Theorem about N^*

The "Append" 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: $\alpha = \varepsilon$

$$\begin{aligned} \mathrm{LHS} &= N^*(s, \varepsilon\beta) = N^*(s, \beta) \\ \mathrm{RHS} &= N^*(N^*(s, \varepsilon), \beta) \\ &= N^*(s, \beta) = \mathrm{LHS} \qquad \text{(by (N1))} \end{aligned}$$

Proof ctd: Step case:

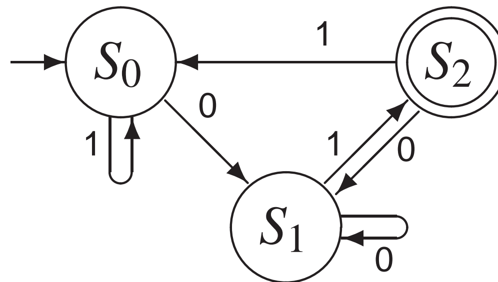
$$
\begin{aligned}
\text{LHS} &= N^*(s,(x\alpha)\beta) \\
&= N^*(s, x(\alpha\beta)) \\
&= N^*(N(s,x), \alpha\beta) && \text{(by (N2))} \\
&= N^*(N^*(N(s,x),\alpha),\beta) && \text{(by IH)}
\end{aligned}
$$

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

Corollary — when β is a single token

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

Example



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

Language of an Automaton

We say a DFA accepts a string if, starting from the start state, it terminates in one of the final states. More precisely, let $A = (\Sigma, S, s_0, F, N)$ be an DFA and w be a string in Σ^*.

We say w is ***accepted*** by A if

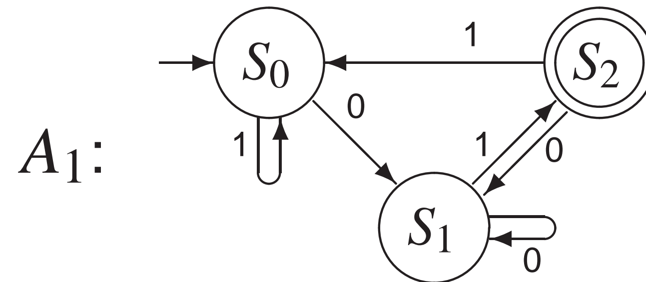
$$N^*(s_0, w) \in F$$

The ***language*** accepted by A is the set of all strings accepted by A:

$$L(A) = \{w \in \Sigma^* \mid N^*(s_0, w) \in F\}$$

That is, $w \in L(A)$ iff $N^*(s_0, w) \in F$.

Example 1 again



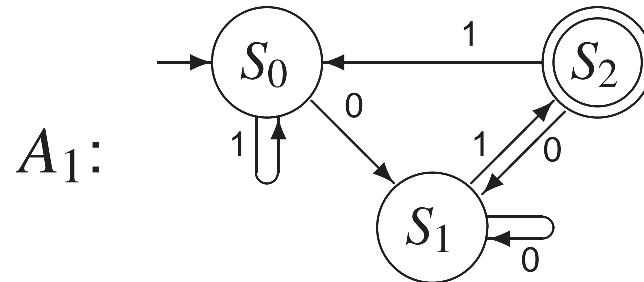
It is easy to see whether a given string is accepted or not.

For example, 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).

$$N^*(S_0, 0011101) = N^*(S_1, 011101) = N^*(S_1, 11101) = \ldots = N^*(S_1, 1) = S_2$$

Other strings that are accepted: 01, 001, 101, 0001, 0101, 00101101 . . .

Example 1 (ctd.)



Accepted strings: 01, 001, 101, 0001, 0101, 00101101 . . .

Not accepted: ε, 0, 1, 00, 10, 11, 100 . . .

What do the accepted strings have in common?

. . . 30 seconds silence . . .

How do we justify the answer you have just given?

Proving an Acceptance Predicate — in General

If we are to claim that a machine M accepts the language that is characterized by a predicate P, then we must prove the following:

Proof obligation 1:

Show that any string satisfying P is accepted by M.

Proof obligation 2:

Show any string accepted by M 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:

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:

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

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

Lemma:

$$\forall s \in S.\ N^*(s, 01) = S_2$$

Proof: Prove that $N^*(S_i, 01) = S_2$ 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, whether $N^*(S_0, \alpha)$ is S_0, S_1 or S_2,

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

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

Proof:

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

Then by the corollary to the “append” theorem,

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

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

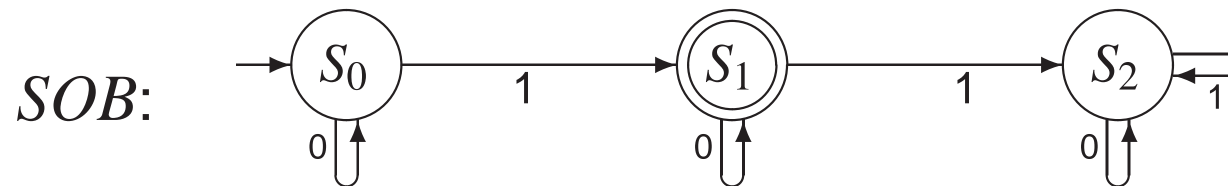
Similarly, $N(N^*(S_0, \alpha), x) = S_1$ and x is 0, again by the definition of N.

But what about strings shorter than any αxy (strings of length 0 or 1)?

- we can show similarly that $N^*(S_0, y) = N(S_0, y) = S_2$ is not possible
- $N^*(S_0, \varepsilon) = S_0 \neq S_2$

Another Example

What language does this DFA accept?

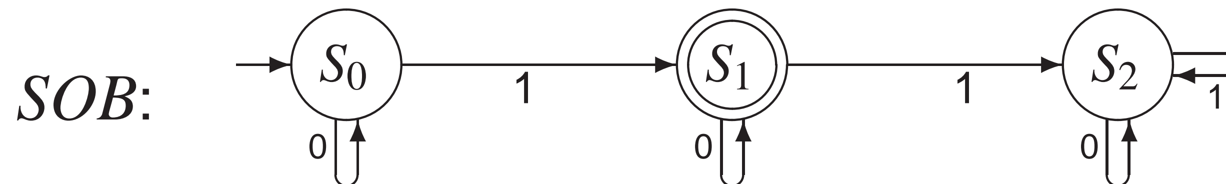


Answer for SOB

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.



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:

$$\forall n \geq 0.\ N^*(S_0, 0^n) \ = \ S_0$$
$$\forall n \geq 0.\ N^*(S_1, 0^n) \ = \ S_1$$

Therefore

$$N^*(S_0, 0^n 10^m) = N^*(N^*(S_0, 0^n), 10^m) = N^*(S_0, 10^m)$$
$$= N^*(N(S_0, 1), 0^m) = N^*(S_1, 0^m) = S_1$$

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 shown in a similar fashion to example 1 on earlier slides.

Limitations of FSAs

General Question: How powerful are FSAs ?

Specifically, what class of languages can be recognised by FSAs ?

A very important example:

Consider this language: $L = \{ a^n b^n \mid n \in \mathbb{N} \}$

That is, $L = \{\varepsilon, ab, aabb, aaabbb, a^4b^4, a^5b^5, \ldots\}$

This language cannot be recognised by any finite state machine

This is because the memory of a FSA is limited, but the machine would have to remember how many ‘a’s it has seen

Proof by contradiction

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

Each of the following expressions denotes a state of A

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

Since this list is infinite and the number of states in A is finite, some of these expressions must denote the same state.

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

(What we have done here is pick on two initial string fragments that the automaton will not be able to distinguish, in terms of what is allowed for the rest of the string)

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

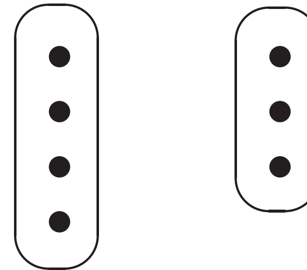
$$N^*(N^*(S_0, a^j), b^i) \;=\; N^*(S_0, a^j b^i) \in F$$

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.



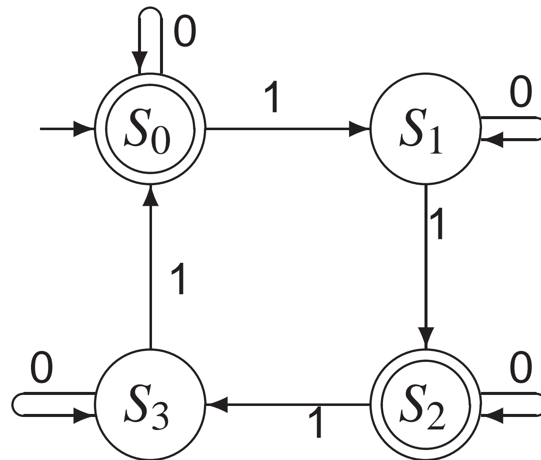
(Finiteness is not really necessary — no function from one set to another with smaller *cardinality* can be one-to-one.)

Equivalence of Automata

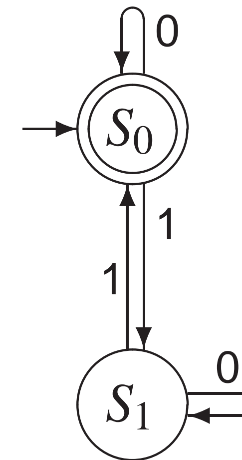
Two automata are said to be **equivalent** if they accept the same language.

Example:

A_4:



A_5:



Interesting question:

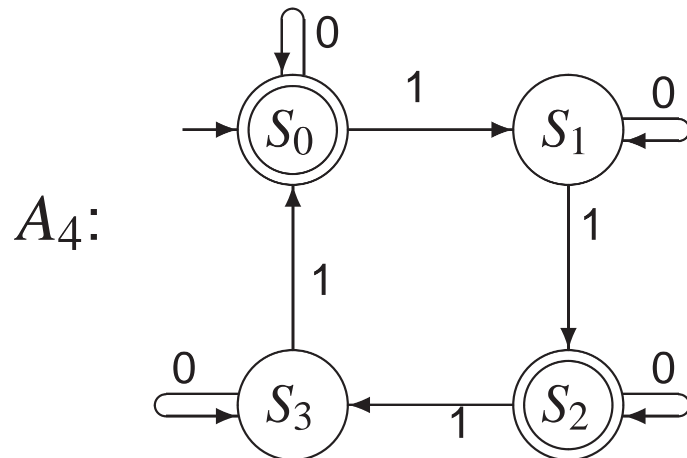
Can we *simplify* a FSA? Is there an equivalent FSA with *fewer states?*

Equivalence of States

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

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

In the following example, S_2 is equivalent to S_0 and S_1 is equivalent to S_3.



Elimination of States

Suppose $A = (\Sigma, S, S_0, F, N)$ is a FSA with state S_k equivalent to S_j. (S_k is not S_0.)

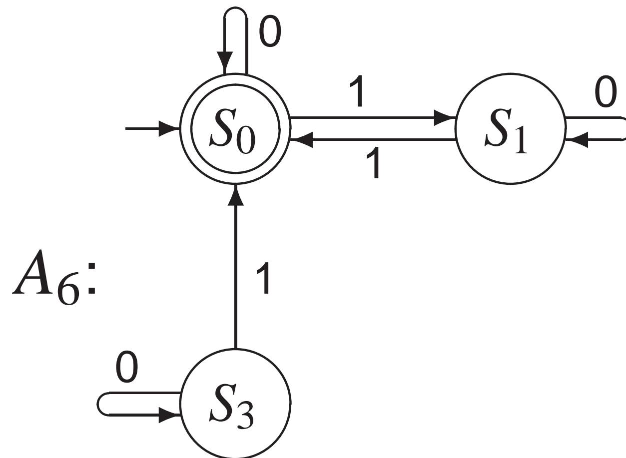
We can eliminate S_k from this automaton by defining a new automaton $A' = (\Sigma, S', S_0, F', N')$ as follows:

- 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

FSA Minimisation

The last example illustrated an automaton being reduced by finding a pair of equivalent states.

If a state cannot be reached from the initial state then it can also be eliminated.

The state S_3 in the FSA A_6 is an example.

(In fact there are deterministic algorithms for FSA minimisation.)

The Standard Minimisation Algorithm

There is an iterative algorithm to compute a list of equivalence classes of states.

- The working data structure for the algorithm is a list of lists ("groups") of states
- Each group contains states that appear to be equivalent, *given the tests we have done so far*
- On each iteration, we test one of the groups with a symbol from the alphabet.
- If we notice differing behaviour, we *split the group.*

The Algorithm Details

- **Input:** A list containing two "groups" (a group is represented as a list of states). One group consists of the Final states and the other consists of the non-final states.

- **Data:** The working data structure, $WDS : [[State]]$, is a list of groups of states. When two states are in different groups, we *know* they are not equivalent.

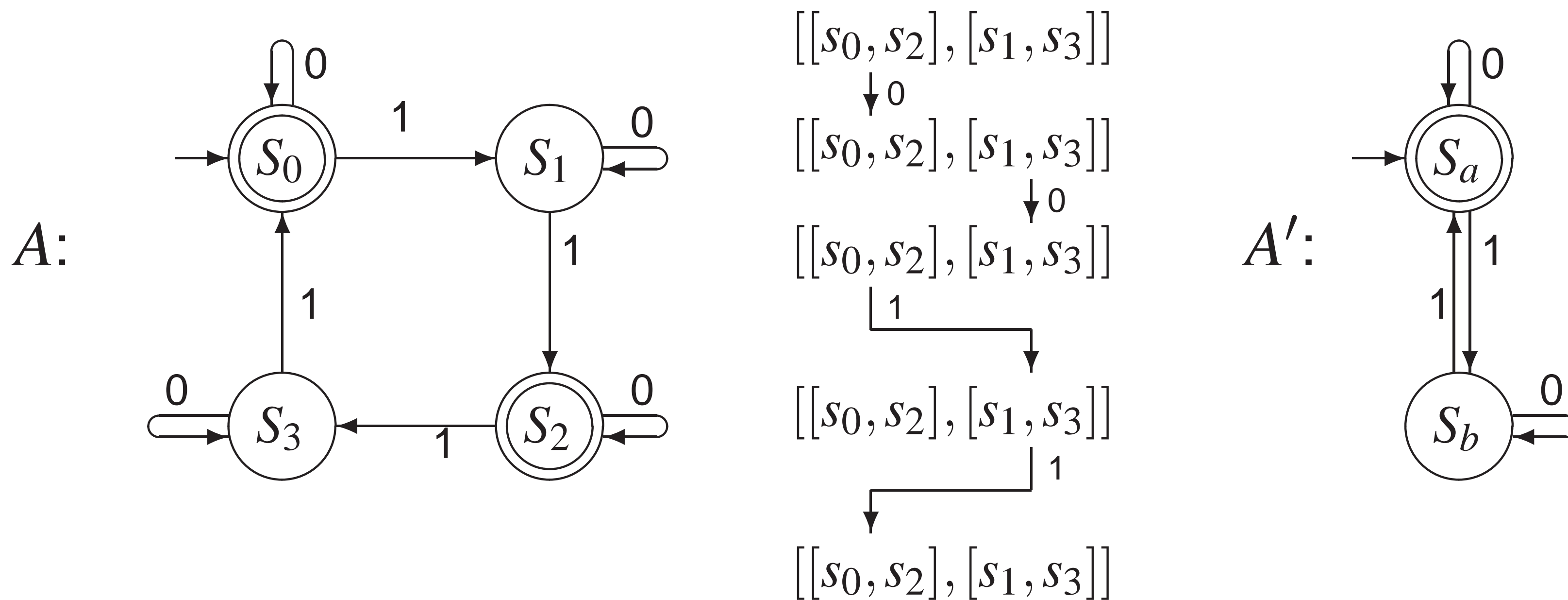
- **Loop:** Pick a group, $\{s_1, \ldots s_j\}$ and a symbol, x. If the states $\{N(s_i, x)\}$ are all in the same group, then the group $\{s_1, \ldots s_j\}$ is not split.

  If the states $\{N(s_i, x)\}$ belong to different groups of WDS, then the group $\{s_1, \ldots s_j\}$ should be split accordingly.

- **Continue until** we cannot, by *any* choice of letter, split *any* group.

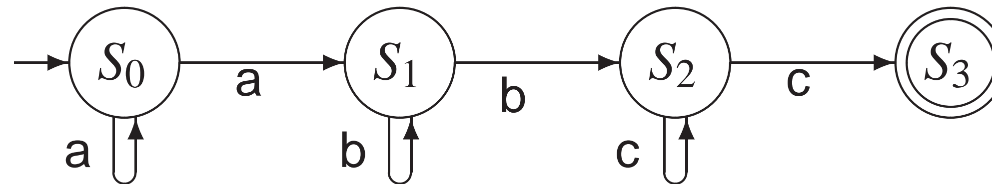
Our Previous Example

Our running example is trivial. The initial split is it.



Non-Deterministic Finite State Automata — NFAs

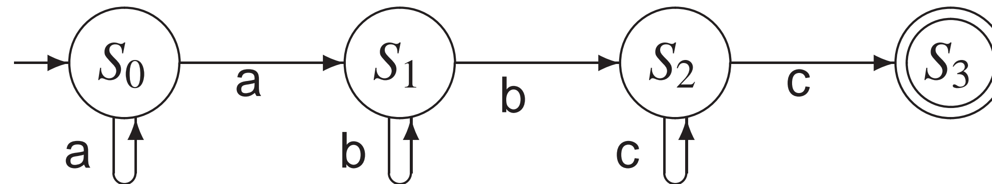
Consider this FSA:



Is it clear what it does?

Is it legal?

Is it legal?



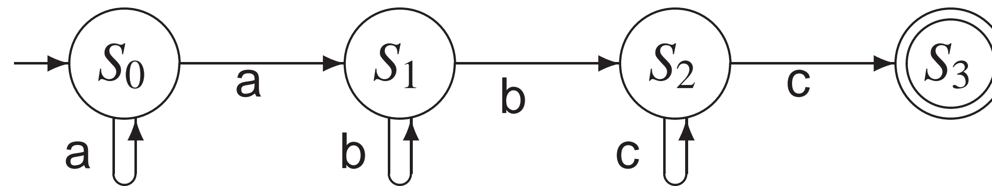
Yes, but it's a *non-deterministic* FSA — a ***NFA***.

How is it different to a DFA?

- Multiple edges with the **same label** come out of states
- For some states, there is **not an edge** for every token

(Therefore, we must have a transition ***relation*** rather than a transition function.)

Is it clear what it does?



For some states, some inputs are not allowed, so the NFA can get “stuck.”

For some states, there are more than one choice of next state.

If the right choices are made, strings of the form $a^i b^j c^k \ (i, j, k \geq 1)$ will take the machine to its final state.

Example: to accept the string $aaabcc$ you need to look ahead to determine the right choice, or backtrack after the wrong choice.

DFAs *vs* NFAs

For each state in a DFA and for each input symbol, there is a unique successor state.

That is, DFAs have a *transition* ***function***.

NFAs allow zero, one or more transitions from a state for the same input symbol.

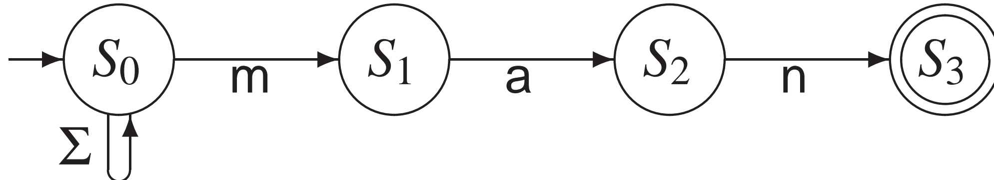
That is, NFAs have a *transition* ***relation***.

An input sequence $a_1, a_2, \ldots, a_n$ is **accepted** by a NFA if *there* ***exists*** *some sequence of transitions that leads from the initial state to a final state.*

Example showing relative simplicity

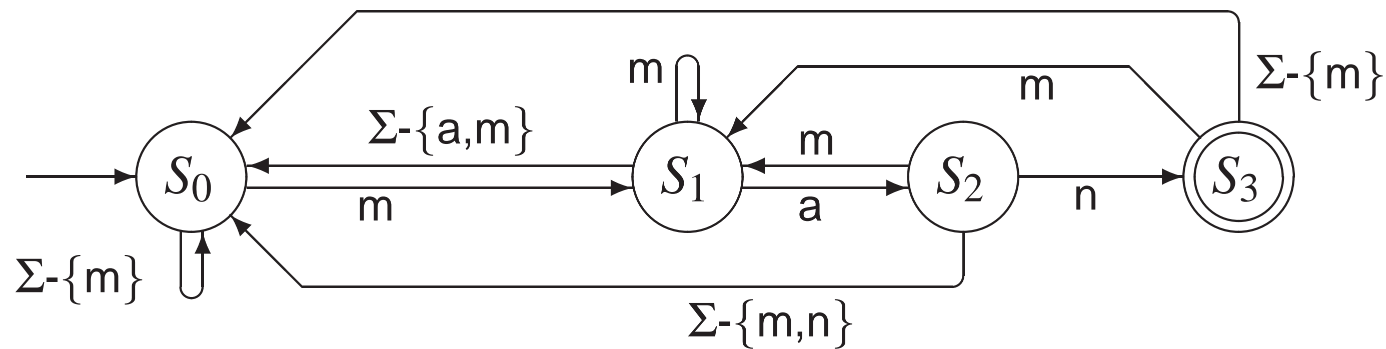
A NFA recognizing strings of letters ending in “man”:

(Σ is the Latin alphabet)



An Equivalent DFA

The following DFA accepts the same language:



Definition of NFA

NFAs differ from DFAs only in that they have a transtion relation rather than a transition function. Normally we also allow an NFA to have *empty* transitions, moving between states without consuming an input symbol.

That is, rather than:

$$N : S \times \Sigma \rightarrow S$$

An NFA has:

$$R \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$$

Or equivalently:

$$R : S \times (\Sigma \cup \{\varepsilon\} \times S \rightarrow Bool$$

Eventual State Relation

In NFAs we have an eventual state *relation*

$$R^* \subseteq S \times \Sigma^* \times S$$

$$\text{or } R^* : S \times \Sigma^* \times S \to Bool$$

$R^*(s, w, s')$ is true if s' is a state the NFA ***can*** reach, starting in state s and reading string w.

If we have an NFA without ε-transitions, here is how we could define R^* inductively:

$$R^*(s, \varepsilon, s)$$

$$R^*(s, x\alpha, s') = \exists s''.R(s, x, s'') \wedge R^*(s'', \alpha, s')$$

For an NFA with ε-transitions, R^* needs to be defined differently.

An Important (but Unsurprising) Theorem about R^*

For all states s, s' and for all strings $\alpha, \beta \in \Sigma^*$

$$R^*(s, \alpha\beta, s') = \exists s''.\ R^*(s, \alpha, s'') \wedge R^*(s'', \beta, s')$$

The proof is similar to the corresponding result for N^* in DFAs.

Language of a NFA

Let $A = (\Sigma, S, s_0, F, R)$ be a NFA.

We say w is ***accepted*** by A if

$$\exists s \in F.\ R^*(s_0, w, s)$$

The ***language*** accepted by A is the set of all strings accepted by A

$$L(A) = \{w \in \Sigma^* \mid \exists s \in F.\ R^*(s_0, w, s)\}$$

That is, $w \in L(A)$ iff ***there exists*** a path through the diagram for A, from s_0 to a final state s ($s \in F$), such that the symbols on the path match the symbols in w

Power of Nondeterminism?

It is easier to come up with a NFA for a given language than a DFA.
(examples earlier)

However, NFAs are no more powerful than DFAs.

Theorem:

If language L is accepted by a NFA, then there is some DFA which accepts the same language.

In fact, there are efficient *algorithms* which transform any NFA to a corresponding DFA.

(The minimisation algorithm in previous slides is part of this)

But the resulting DFA itself may have a relatively huge number of states

Constructing the Equivalent DFA from an NFA

The idea is simple, the details are complicated.

Let $\{q_0, q_1, \ldots, q_n\}$ be the *set* of states of the NFA.

We construct a DFA in which *each* state is labelled by a *set* chosen from the symbols $\{q_0, q_1, \ldots, q_n\}$.

Given an input string $\alpha\beta$: after the NFA has processed α, with β remaining, suppose it may be in any of the states (say) $\{q_1, q_3, q_5\}$.

Then the corresponding DFA, after it has processed α, will be in the state labelled $\{q_1, q_3, q_5\}$.

The state of the DFA labelled $\{q_1, q_3, q_5\}$ will be a final state if *any* of the states q_1, q_3, q_5 are final states of the NFA

Application

There are standard tools (in every decent programming language) which generate *"lexical analysers"* or *"scanners"* that check whether input strings match given **regular expressions**. (Lex, Flex, Alex . . .)

How do they work?

1. Derive a NFA from the regular expressions (see below)

2. Convert the NFA to a DFA

3. A standard driver program takes the DFA as a data structure

Better ones (like Flex) usually produce more efficient scanners than hand-coding.

Application — adapting a DFA

The operation of a DFA is, generally, tweaked as follows:

- The scanner picks part of the input string, and the rest is used next time
- It chooses the *longest* initial substring which is accepted by the DFA
- It also returns an indication of what sort of substring is returned (identifier, reserved word, parenthesis, etc)

Regular Expressions

Regular expressions are used to specify languages by showing a pattern that the strings match.

Widely used in computing — editors, commands (e.g. `grep`), built in to some programming languages (Perl, Ruby . . .).

We will consider only a few basic operators: vertical bar to indicate choice; star to indicate repetition; parentheses for grouping; ε for the empty string; and concatenation for "concatenation" (see later)

For example:

- a^* indicates 0 or more as.
- $\text{yes} \mid \text{no}$ is the language with just the 2 given strings.
- $(0 \mid 1)^*$ indicates the set of binary numerals, and the empty string.

Regular Expressions — More Examples

- $0|(1(0|1)^*)$ is the set of (non-empty) binary numerals with no unnecessary leading zeros.
- $(a \mid b)^* c (a \mid b)^*$ is the set of strings over $\{a, b, c\}$ with just one c.
- $(0^* 10^* 10^*)^*$ is the language of bit-strings that have an even number of ones. (Alternatively $0^*(10^*10^*)^*$)
- $(z^*(x^* \mid y^*)\, z)^*(x^* \mid y^*)$ is the set of strings over $\{x, y, z\}$ with no x and y adjacent.
- $1 \mid (0\ (\ \varepsilon \mid (.(0 \mid 1)^* 1))))$ is binary fractional numerals between 0 and 1 with no trailing zeroes. (e.g. 0.1, 0.110011 but not $.1$ or 0.10)

The Definition of Regular Expressions

The regular expressions on alphabet Σ and the sets that they denote are:

- $\emptyset$ is a regular expression and denotes the empty set $\emptyset$
- ε is a regular expression and denotes the set $\{\varepsilon\}$
- for each $a \in \Sigma$, a is a regular expression and denotes the set $\{a\}$

If α and β are regular expressions denoting languages R and S respectively, then:

- $\alpha \mid \beta$ denotes $R \cup S$
- $\alpha\beta$ denotes RS which is $\{xy \mid x \in R \wedge y \in S\}$
- α^* denotes R^*, ie, the set of *finitely* many $r_i \in R$, concatenated

R^* is (inductively) defined as $\{\varepsilon\} \cup RR^*$

Precedence of Regular Expression Operators

*, then concatenation, then $|$

Thus $ab^*|c^*d$ means $(a(b^*))|((c^*)d)$

Regular Expressions and FSAs

An elegant correspondence…

For every regular expression R there is a NFA which accepts exactly those strings matching R. (next slides)

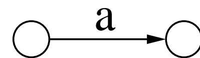
(That means there is also a DFA corresponding to R.)

If language L is accepted by some FSA, there is a regular expression that matches exactly the sentences of L. (hard)

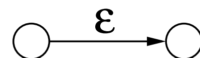
Regular Expressions to NFAs

We inductively define an NFA corresponding to given regular expression:
(NFAs shown with start state on the left, and a *single* final state on the right ;
assume no edges into the start state or out of the final state ;
we can use edges labelled ε to give an NFA with these poperties)

- When the regular expression is a symbol a of the alphabet (language is $\{a\}$) the automaton is



- When the regular expression is ε (language is $\{\varepsilon\}$) the automaton is

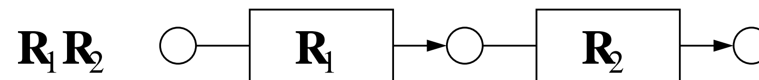


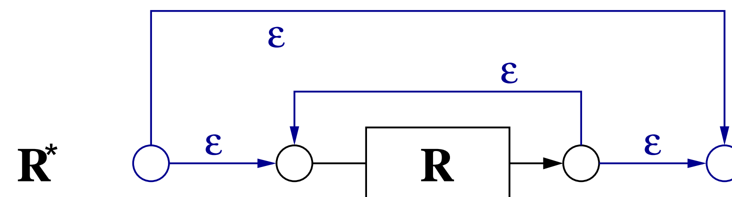
- For the regular expression $\emptyset$ (language is $\emptyset$) the automaton has no edges

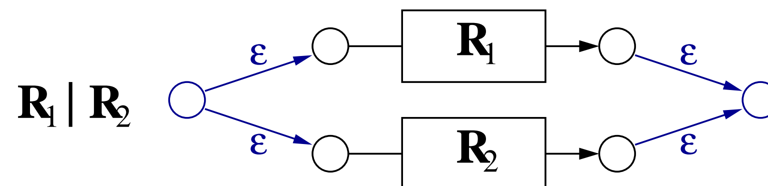
Regular Expressions to NFAs, ctd

Suppose the NFA corresponding to some R is:

Then NFAs corresponding to composite regular expressions are defined as follows:







Example

Given the regular expression for binary numerals without leading zeros, $(0 \mid 1(0|1)^*)$, the above algorithm gives this NFA.

Example — reversing every string in a language — easy with NFAs

Given a language L, consider L', got from L by reversing every string in L.
Given an NFA A accepting L, we want a NFA A' which accepts L'

To do this you ensure A has exactly one final state, then you

- swap the initial and final states
- reverse all the edges

You can then convert the result to a DFA — but it will look quite unlike the original

Comment: reversing a regular expression is also easy, and reverses every string in the language

Example — getting the complement of a language — easy with DFAs

Given a language L, its complement is $\overline{L} = \Sigma^* \setminus L$, the set of strings not in L.

Suppose $A = (\Sigma, S, S_0, F, N)$ is a DFA which accepts L.

Then $A' = (\Sigma, S, S_0, S \setminus F, N)$ is a DFA which accepts $\overline{L}$.

That is, you make the final states non-final, and vice versa.

Then whenever $N^*(S_0, \alpha) \in F$ in A, then $N^*(S_0, \alpha) \notin F$ in A', and vice versa.

That is, strings accepted by A are not accepted by A', and vice versa.