# COSC 341
# Theory of Computing
# Lecture 2
# Automata and grammars

Stephen Cranefield
stephen.cranefield@otago.ac.nz

Lecture slides (mostly) by Michael Albert

*Keywords*: deterministic finite state automaton (DFA), regular grammar

# Deterministic finite state automata

A deterministic finite state automaton (DFA), **A**, consists of the following elements:

- A finite set $\Sigma$ called its alphabet,
- A finite set $\mathcal{S}$ called its states,
- A function $T : \mathcal{S} \times \Sigma \to \mathcal{S}$ called its transition function,
- A single element $s \in \mathcal{S}$ called its start state,
- A subset $A \subseteq \mathcal{S}$ called its final states or accepting states.

The language accepted by **A**, $L(\mathbf{A})$ is the set of all strings over $\Sigma$ such that "when you press the buttons defined by the string you wind up in an accepting state".

# What is a string?

Given an alphabet $\Sigma$ what do we mean when we talk about a string with characters from $\Sigma$? (or "over $\Sigma$")

A string of length $n$ over $\Sigma$ is a sequence $\sigma_0\sigma_1 \ldots \sigma_{n-1}$ where each $\sigma_i \in \Sigma$.

What's a sequence?

A sequence of length $n$ over $\Sigma$ is a function:

$$\sigma : \{0, 1, 2, \ldots, n-1\} \to \Sigma.$$

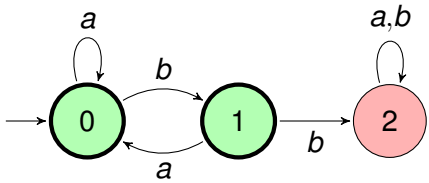
Note the $n = 0$ case defines the *empty string*, which we denote . The set of all strings over $\Sigma$ is denoted .

# Try this

- What other words do we use when we refer to strings?
- How do they relate to the definition above?
- What (if any) are the natural operations on strings?
- How can they be defined in the context above?

We'll look at some of these in the tutorial.

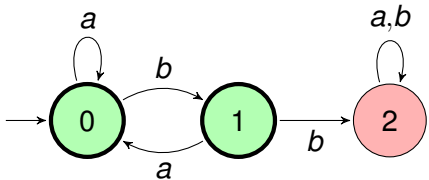
# Our favourite machine




How could we represent the strings accepted by this machine in text form?

$$
\begin{aligned}
S_0 &\rightarrow \epsilon \mid aS_0 \mid bS_1 \\
S_1 &\rightarrow \epsilon \mid aS_0 \mid bS_2 \\
S_2 &\rightarrow aS_2 \mid bS_2
\end{aligned}
$$

# Our favourite machine




How could we represent the strings accepted by this machine in text form?

$$\begin{aligned} S &\rightarrow \epsilon \mid aS \mid bB \\ B &\rightarrow \epsilon \mid aS \end{aligned}$$

# Regular grammars

A right-regular grammar, $G$, consists of:

- An alphabet, $\Sigma$, ($\{a, b\}$).
- A set $N$ of non-terminal symbols, ($\{S, B\}$)
- A designated start symbol, $S \in N$
- A list of production rules ($S \to aS$) each of one of the forms:
  - $X \to \epsilon$ ($X \in N$)
  - $X \to a$ ($X \in N$, $a \in \Sigma$)
  - $X \to aY$ ($X, Y \in N$, $a \in \Sigma$)

# From grammar to language

Each non-terminal, $X$ of a regular grammar, $G$, has an associated language, $L_G(X)$ (or just $L(X)$) which is a subset of $\Sigma^*$ defined recursively as follows:

- If $X \to \epsilon$ (resp. $X \to a$) is in $G$, then $\epsilon \in L(X)$ (resp. $a \in L(X)$). These are the base cases.
- For every production $X \to aY$ in $G$, $L(X)$ contains the set $aL(Y)$.

When we say "defined recursively" we implicitly say "and no other strings except those which must belong to $L(X)$ according to these rules are included"

The language of the grammar itself is the language of its start state, $S$.

# Example

$$\begin{aligned} S &\rightarrow \epsilon \mid aS \mid bB \\ B &\rightarrow \epsilon \mid aS \end{aligned}$$

Now we can build the languages $L(S)$ and $L(B)$ from the bottom up:

# Example

$$
\begin{aligned}
S &\rightarrow \epsilon \mid aS \mid bB \\
B &\rightarrow \epsilon \mid aS
\end{aligned}
$$

Now we can build the languages $L(S)$ and $L(B)$ from the bottom up:

$$
\begin{aligned}
L(S) &= \epsilon, \ldots \\
L(B) &= \epsilon, \ldots
\end{aligned}
$$

# Example

$$
\begin{aligned}
S &\rightarrow \epsilon \mid aS \mid bB \\
B &\rightarrow \epsilon \mid aS
\end{aligned}
$$

Now we can build the languages $L(S)$ and $L(B)$ from the bottom up:

$$
\begin{aligned}
L(S) &= \epsilon, a, b, \ldots \\
L(B) &= \epsilon, a \ldots
\end{aligned}
$$

# Example

$$\begin{aligned} S &\rightarrow \epsilon \mid aS \mid bB \\ B &\rightarrow \epsilon \mid aS \end{aligned}$$

Now we can build the languages $L(S)$ and $L(B)$ from the bottom up:

$$L(S) = \epsilon, a, b, aa, ab, ba, \ldots$$
$$L(B) = \epsilon, a, aa, ab, \ldots$$

# Example

$$\begin{aligned} S &\rightarrow \epsilon \mid aS \mid bB \\ B &\rightarrow \epsilon \mid aS \end{aligned}$$

Now we can build the languages $L(S)$ and $L(B)$ from the bottom up:

$$L(S) = \epsilon, a, b, aa, ab, ba, aaa, aab, aba, baa, bab, \ldots$$
$$L(B) = \epsilon, a, aa, ab, aaa, aab, aba, \ldots$$

Note that the order in which the elements of the language are generated is in some sense backwards from the way we'd think about it in the automaton (i.e., we add characters on the left, rather than the right).

# Things that don't match with DFAs

In a regular grammar we might have multiple productions with the same initial symbol, e.g.,:

$$A \to aB \mid aC \mid bD,$$

or (as above) we might have no productions with some initial symbol:

$$B \to \epsilon \mid aS.$$

Neither of these is "allowed" in a DFA since each state is supposed to have exactly one outgoing arrow corresponding to each letter of the alphabet.

# Things that don't match with DFAs

In a regular grammar we might have multiple productions with the same initial symbol, e.g.,:

$$A \to aB \mid aC \mid bD,$$

or (as above) we might have no productions with some initial symbol:

$$B \to \epsilon \mid aS.$$

Neither of these is "allowed" in a DFA since each state is supposed to have exactly one outgoing arrow corresponding to each letter of the alphabet.

So, regular grammars potentially allow for *more languages* than are represented by DFAs. But do they really?

# What's the big deal about languages anyhow?

In what sense is generating or recognising languages computing?