COSC 341
Theory of Computing
Lecture 4
Non-determinism and regular languages

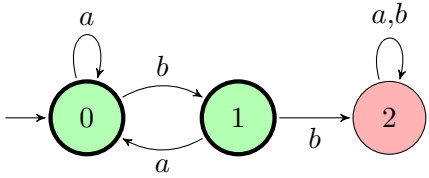
Stephen Cranefield
stephen.cranefield@otago.ac.nz

Lecture slides (mostly) by Michael Albert

Keywords: **non**-deterministic finite state automata (NFA), regular language

Models of computing covered so far

- Deterministic finite state automata (DFAs)



- Right-regular grammars

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

S stands for *start* and B for *seen a* b

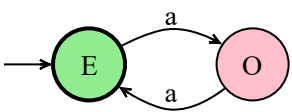
DFAs are deterministic whereas grammars generate languages through non-deterministic application of production rules.

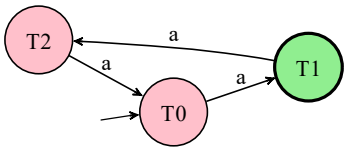
Today we consider finite automata with non-deterministic transitions.

An example language (DFA solution)

Consider the language defined as “words over the alphabet $\{a\}$ such that the number of letters is a multiple of two, or one more than a multiple of three. ”

We can easily write separate DFAs for the “multiple of two” and “one more than a multiple of three” cases.





To get the DFA for the union of the languages accepted by these DFAs we can apply the cross-product construction discussed in Tutorial 2 (Q5).

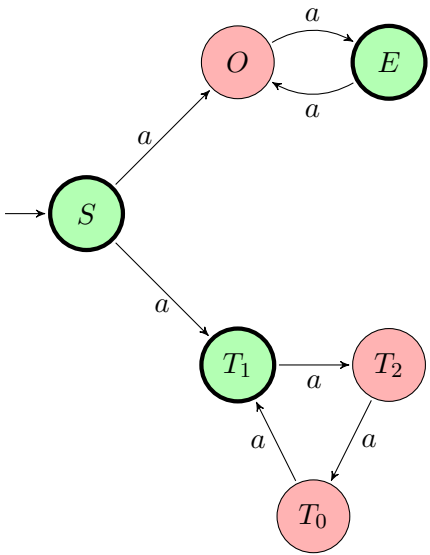
An example language (right-regular grammar solution)

Consider the language defined as “words over the alphabet $\{a\}$ such that the number of letters is a multiple of two, or one more than a multiple of three. ”

$$
\begin{aligned}
S &\to aO \mid aT_1 \\
O &\to aE \\
E &\to \epsilon \mid aO \\
T_1 &\to \epsilon \mid aT_2 \\
T_2 &\to aT_0 \\
T_0 &\to aT_1
\end{aligned}
$$

In effect, the grammar “decides” depending on the production chosen for the first a whether it’s keeping track of the number of letters modulo 2 or modulo 3. Can we do this with a machine?

A non-deterministic finite automaton



Non-deterministic finite state automata

A non-deterministic finite state automaton (NFA), $\mathbf{A}$, is a machine like the above.

- It is associated with a finite alphabet, Σ,
- It has a finite set of states, $\mathcal{S}$,
- One state is designated the start state
- Some states are accepting states
- It has a *set*, T of transitions, *each of which is a triple in* $\mathcal{S} \times (\Sigma \cup \{\epsilon\}) \times \mathcal{S}$

The language accepted by $\mathbf{A}$, $L(\mathbf{A})$ is the set of all strings over Σ such that "when you press the buttons defined by the string you can choose a sequence of valid transitions and finish in an accepting state".

The meaning of a transition like (A, ϵ, B) is that "even if no button is pressed you can choose to go from state A directly to state B". This is known as an ϵ-transition.

Where is the non-determinism?

Recall that a DFA required T to be a function $\mathcal{S} \times \Sigma \to \mathcal{S}$. In contrast, an NFA may have multiple transitions for a given state A and character x, e.g.:

$$(A, x, B)$$

and

$$(A, x, C)$$

or even no transitions for state A and character x.

"In computer science and computer programming, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm." [Wikipedia]

How does an NFA compute?

"Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state q_1 in NFA N_1, and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows all the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an ϵ symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ϵ-labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before."

(Michael Sipser)

A more mechanical explanation

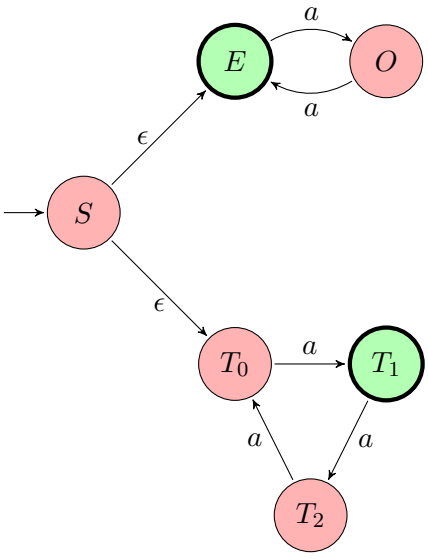
Think of ‘tokens’ that can be placed on the states currently reached by all the possible transitions so far. At most one token can be in each state—adding new ones has no additional effect.

- Initially, the machine has tokens on the ϵ-closure of the start state: the set of all states reachable from the start state by sequences of ϵ transitions.
- When an input character x is received, for every state with a token, ‘slide the token’ (or a copy) along each outgoing x transition to the destination state, leaving no token on the source state.
- Do the same with all ϵ-transitions, but in this case, keep a copy of the token on the transition’s starting state.
- The machine is in an accepting state if any accepting state contains a token

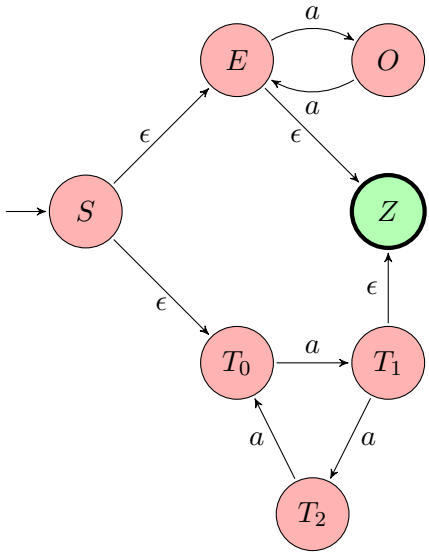
An alternative view: The genie in the NFA

- An alternative view of an NFA is to imagine that it comes with a helpful genie.
- The genie knows in advance which keys will be pressed, and makes appropriate choices to ensure that the string is accepted if there's any possibility that it could be.
- There's no way the genie can trick us into accepting something that shouldn't be accepted since we can observe the state changes and check that they are all allowed as transitions.

The genie can pre-commit



And knows when we've finished



NFAs in standard form

The idea of the construction above is that, given any NFA we can convert (using ϵ-transitions) it into one that accepts the same language and where:

- The start state is non-accepting and has only ϵ-transitions
- There is a unique accepting state Z, which is reached only by ϵ-transitions and which has no outgoing transitions.

The advantage of doing this is that machines of this type are easy to glue together in various ways to prove closure properties of the set of languages accepted by NFAs.

Try these

If L is a language accepted by an NFA then so are all of the following:

- The language consisting of all suffixes of words belonging to L
- The language consisting of all prefixes of words belonging to L
- The language L^* consisting of those words that can be written as the concatenation of zero or more words from L.
- The language L^r consisting of all reversals of words from L

L^*:

- Put the NFA in standard form with start state S and end state Z.
- Create a new start state S^* and end state Z^*.
- Use all existing transitions and add these: $S^* \xrightarrow{\epsilon} Z^*$ (accepts the empty word), $S^* \xrightarrow{\epsilon} S$, $Z \xrightarrow{\epsilon} Z^*$ (connect up the new states), $Z \xrightarrow{\epsilon} S$ (loops the original NFA).

L^r: Put NFA into standard form, reverse all transitions and swap the roles of the start and end states.

Regular languages

The regular languages over an alphabet Σ are defined recursively as follows:

- $\{\}$, $\{\epsilon\}$ and $\{a\}$ are all regular languages (for all $a \in \Sigma$),
- if L is a regular language then so is L^* the language consisting of all the concatenations of zero or more strings in L, and
- if L and K are regular languages then so are LK, the language of all concatenations of a string in L with one in K and $L + K$, the union of L and K.

We'll usually blur the distinction between the language $\{a\}$ and a itself, so we can write regular languages as regular expressions like:

$$(a + ab)^* \left((ba + ab)b(bbb)^*\right)^*$$

Finite languages (those with a finite number of words) are regular. Why?

Try this

Is the language of the machine we looked at that accepts all strings not containing consecutive b's regular?

If so, write down a regular expression for it.

Two possibilities:

$$(a + ba)^*(b + \epsilon)$$
$$a^*(baa^*)^*(ba^* + \epsilon)$$

The landscape of languages

We have defined four ways of generating and/or accepting languages:

- Deterministic finite-state automata
- Regular grammars
- Non-deterministic finite-state automata
- Regular expressions

What relationships do we know about the groups of languages they define?