

Week4 wednesday

Regular sets are not the end of the story

- Many nice / simple / important sets are not regular
- Limitation of the finite-state automaton model: Can't "count", Can only remember finitely far into the past, Can't backtrack, Must make decisions in "real-time"
- We know actual computers are more powerful than this model...

The **next** model of computation. Idea: allow some memory of unbounded size. How?

- To generalize regular expressions: **context-free grammars**
- To generalize NFA: **Pushdown automata**, which is like an NFA with access to a stack: Number of states is fixed, number of entries in stack is unbounded. At each step (1) Transition to new state based on current state, letter read, and top letter of stack, then (2) (Possibly) push or pop a letter to (or from) top of stack. Accept a string iff there is some sequence of states and some sequence of stack contents which helps the PDA process the entire input string and ends in an accepting state.

Is there a PDA that recognizes the nonregular language $\{0^n 1^n \mid n \geq 0\}$?



The PDA with state diagram above can be informally described as:

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If the stack becomes empty and we are at the end of the input string, accept the input. If the stack becomes empty and there are 1s left to read, or if 1s are finished while the stack still contains 0s, or if any 0s appear in the string following 1s, reject the input.

Trace the computation of this PDA on the input string 01.

Trace the computation of this PDA on the input string 011.

A PDA recognizing the set { $0^n 1^n$ } can be informally described as:

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If the stack becomes empty and there is exactly one 1 left to read, read that 1 and accept the input. If the stack becomes empty and there are either zero or more than one 1s left to read, or if the 1s are finished while the stack still contains 0s, or if any 0s appear in the input following 1s, reject the input.

Modify the state diagram below to get a PDA that implements this description:



Definition A **pushdown automaton** (PDA) is specified by a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q is the finite set of states, Σ is the input alphabet, Γ is the stack alphabet,

$$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$$

is the transition function, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of accept states.

Draw the state diagram and give the formal definition of a PDA with $\Sigma = \Gamma$.

Draw the state diagram and give the formal definition of a PDA with $\Sigma \cap \Gamma = \emptyset$.

Extra practice: Consider the state diagram of a PDA with input alphabet Σ and stack alphabet Γ .

Label	means
$a, b; c$ when $a \in \Sigma, b \in \Gamma, c \in \Gamma$	
$a, \varepsilon; c$ when $a \in \Sigma, c \in \Gamma$	
$a, b; \varepsilon$ when $a \in \Sigma, b \in \Gamma$	
$a, \varepsilon; \varepsilon$ when $a \in \Sigma$	

How does the meaning change if a is replaced by ε ?

Note: alternate notation is to replace ; with \rightarrow

Week4 friday

For the PDA state diagrams below, $\Sigma = \{0, 1\}$.

Mathematical description of language

State diagram of PDA recognizing language

$\Gamma = \{\$, \#\}$



$\Gamma = \{ @, 1 \}$



$$\{0^i 1^j 0^k \mid i, j, k \geq 0\}$$

Big picture: PDAs were motivated by wanting to add some memory of unbounded size to NFA. How do we accomplish a similar enhancement of regular expressions to get a syntactic model that is more expressive?

DFA, NFA, PDA: Machines process one input string at a time; the computation of a machine on its input string reads the input from left to right.

Regular expressions: Syntactic descriptions of all strings that match a particular pattern; the language described by a regular expression is built up recursively according to the expression's syntax

Context-free grammars: Rules to produce one string at a time, adding characters from the middle, beginning, or end of the final string as the derivation proceeds.

Term	Typical symbol	Definition
Context-free grammar (CFG)	G	$G = (V, \Sigma, R, S)$
Variables	V	Finite set of symbols that represent phases in production pattern
Terminals	Σ	Alphabet of symbols of strings generated by CFG $V \cap \Sigma = \emptyset$
Rules	R	Each rule is $A \rightarrow u$ with $A \in V$ and $u \in (V \cup \Sigma)^*$
Start variable	S	Usually on LHS of first / topmost rule
Derivation	$S \Rightarrow \dots \Rightarrow w$	Sequence of substitutions in a CFG Start with start variable, apply one rule to one occurrence of a variable at a time
Language generated by the CFG G	$L(G)$	$\{w \in \Sigma^* \mid \text{there is derivation in } G \text{ that ends in } w\} = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$
Context-free language		A language that is the language generated by some CFG
Sipser pages 102-103		

Examples of context-free grammars, derivations in those grammars, and the languages generated by those grammars

$G_1 = (\{S\}, \{0\}, R, S)$ with rules

$$S \rightarrow 0S$$

$$S \rightarrow 0$$

In $L(G_1)$...

Not in $L(G_1)$...

$$G_2 = (\{S\}, \{0, 1\}, R, S)$$

$$S \rightarrow 0S \mid 1S \mid \varepsilon$$

In $L(G_2) \dots$

Not in $L(G_2) \dots$

$(\{S, T\}, \{0, 1\}, R, S)$ with rules

$$\begin{aligned} S &\rightarrow T1T1T1T \\ T &\rightarrow 0T \mid 1T \mid \varepsilon \end{aligned}$$

In $L(G_3) \dots$

Not in $L(G_3) \dots$

$G_4 = (\{A, B\}, \{0, 1\}, R, A)$ with rules

$$A \rightarrow 0A0 \mid 0A1 \mid 1A0 \mid 1A1 \mid 1$$

In $L(G_4) \dots$

Not in $L(G_4) \dots$

Extra practice: Is there a CFG G with $L(G) = \emptyset$?

Week6 friday

Nondeterministic Turing machine

At any point in the computation, the nondeterministic machine may proceed according to several possibilities: $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ where

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

The computation of a nondeterministic Turing machine is a tree with branching when the next step of the computation has multiple possibilities. A nondeterministic Turing machine accepts a string exactly when some branch of the computation tree enters the accept state.

Given a nondeterministic machine, we can use a 3-tape Turing machine to simulate it by doing a breadth-first search of computation tree: one tape is “read-only” input tape, one tape simulates the tape of the nondeterministic computation, and one tape tracks nondeterministic branching. Sipser page 178

Two models of computation are called **equally expressive** when every language recognizable with the first model is recognizable with the second, and vice versa.

Church-Turing Thesis (Sipser p. 183): The informal notion of algorithm is formalized completely and correctly by the formal definition of a Turing machine. In other words: all reasonably expressive models of computation are equally expressive with the standard Turing machine.

A language L is **recognized by** a Turing machine M means




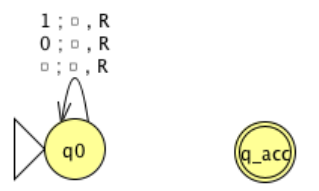
A Turing machine M **recognizes** a language L if means

A Turing machine M is a **decider** means

A language L is **decided by** a Turing machine M means

A Turing machine M **decides** a language L means

Fix $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \sqcup\}$ for the Turing machines with the following state diagrams:

 <p>Decider? Yes / No</p>	 <p>Decider? Yes / No</p>
 <p>Decider? Yes / No</p>	 <p>Decider? Yes / No</p>

Claim: If two languages (over a fixed alphabet Σ) are Turing-recognizable, then their union is as well.

Proof using Turing machines:

Proof using nondeterministic Turing machines:

Proof using enumerators:

Describing Turing machines (Sipser p. 185)

To define a Turing machine, we could give a

- **Formal definition:** the 7-tuple of parameters including set of states, input alphabet, tape alphabet, transition function, start state, accept state, and reject state; or,
- **Implementation-level definition:** English prose that describes the Turing machine head movements relative to contents of tape, and conditions for accepting / rejecting based on those contents.
- **High-level description:** description of algorithm (precise sequence of instructions), without implementation details of machine. As part of this description, can “call” and run another TM as a subroutine.

The Church-Turing thesis posits that each algorithm can be implemented by some Turing machine

High-level descriptions of Turing machine algorithms are written as indented text within quotation marks.

Stages of the algorithm are typically numbered consecutively.

The first line specifies the input to the machine, which must be a string. This string may be the encoding of some object or list of objects.

Notation: $\langle O \rangle$ is the string that encodes the object O . $\langle O_1, \dots, O_n \rangle$ is the string that encodes the list of objects O_1, \dots, O_n .

Assumption: There are Turing machines that can be called as subroutines to decode the string representations of common objects and interact with these objects as intended (data structures).

For example, since there are algorithms to answer each of the following questions, by Church-Turing thesis, there is a Turing machine that accepts exactly those strings for which the answer to the question is “yes”

- Does a string over $\{0, 1\}$ have even length?
- Does a string over $\{0, 1\}$ encode a string of ASCII characters?¹
- Does a DFA have a specific number of states?
- Do two NFAs have any state names in common?
- Do two CFGs have the same start variable?

¹An introduction to ASCII is available on the w3 tutorial here.

Week3 monday

The state diagram of an NFA over $\{a, b\}$ is below. The formal definition of this NFA is:



The language recognized by this NFA is:

Suppose A_1, A_2 are languages over an alphabet Σ . **Claim:** if there is a NFA N_1 such that $L(N_1) = A_1$ and NFA N_2 such that $L(N_2) = A_2$, then there is another NFA, let's call it N , such that $L(N) = A_1 \cup A_2$.

Proof idea: Use nondeterminism to choose which of N_1, N_2 to run.

Formal construction: Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ and assume $Q_1 \cap Q_2 = \emptyset$ and that $q_0 \notin Q_1 \cup Q_2$. Construct $N = (Q, \Sigma, \delta, q_0, F_1 \cup F_2)$ where

- $Q =$
- $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ is defined by, for $q \in Q$ and $a \in \Sigma_\epsilon$:

Proof of correctness would prove that $L(N) = A_1 \cup A_2$ by considering an arbitrary string accepted by N , tracing an accepting computation of N on it, and using that trace to prove the string is in at least one of A_1, A_2 ; then, taking an arbitrary string in $A_1 \cup A_2$ and proving that it is accepted by N . Details left for extra practice.

Over the alphabet $\{a, b\}$, the language L described by the regular expression $\Sigma^* a \Sigma^* b$

includes the strings

and excludes the strings

The state diagram of a NFA recognizing L is:

Suppose A_1, A_2 are languages over an alphabet Σ . **Claim:** if there is a NFA N_1 such that $L(N_1) = A_1$ and NFA N_2 such that $L(N_2) = A_2$, then there is another NFA, let's call it N , such that $L(N) = A_1 \circ A_2$.

Proof idea: Allow computation to move between N_1 and N_2 “spontaneously” when reach an accepting state of N_1 , guessing that we’ve reached the point where the two parts of the string in the set-wise concatenation are glued together.

Formal construction: Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ and assume $Q_1 \cap Q_2 = \emptyset$. Construct $N = (Q, \Sigma, \delta, q_0, F)$ where

- $Q =$
- $q_0 =$
- $F =$
- $\delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q)$ is defined by, for $q \in Q$ and $a \in \Sigma_\varepsilon$:

$$\delta((q, a)) = \begin{cases} \delta_1((q, a)) & \text{if } q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1((q, a)) & \text{if } q \in F_1 \text{ and } a \in \Sigma \\ \delta_1((q, a)) \cup \{q_2\} & \text{if } q \in F_1 \text{ and } a = \varepsilon \\ \delta_2((q, a)) & \text{if } q \in Q_2 \end{cases}$$

Proof of correctness would prove that $L(N) = A_1 \circ A_2$ by considering an arbitrary string accepted by N , tracing an accepting computation of N on it, and using that trace to prove the string can be written as the result of concatenating two strings, the first in A_1 and the second in A_2 ; then, taking an arbitrary string in $A_1 \circ A_2$ and proving that it is accepted by N . Details left for extra practice.

Suppose A is a language over an alphabet Σ . **Claim:** if there is a NFA N such that $L(N) = A$, then there is another NFA, let's call it N' , such that $L(N') = A^*$.

Proof idea: Add a fresh start state, which is an accept state. Add spontaneous moves from each (old) accept state to the old start state.

Formal construction: Let $N = (Q, \Sigma, \delta, q_1, F)$ and assume $q_0 \notin Q$. Construct $N' = (Q', \Sigma, \delta', q_0, F')$ where

- $Q' = Q \cup \{q_0\}$
- $F' = F \cup \{q_0\}$
- $\delta' : Q' \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q')$ is defined by, for $q \in Q'$ and $a \in \Sigma_\varepsilon$:

$$\delta'((q, a)) = \begin{cases} \delta((q, a)) & \text{if } q \in Q \text{ and } q \notin F \\ \delta((q, a)) & \text{if } q \in F \text{ and } a \in \Sigma \\ \delta((q, a)) \cup \{q_1\} & \text{if } q \in F \text{ and } a = \varepsilon \\ \{q_1\} & \text{if } q = q_0 \text{ and } a = \varepsilon \\ \emptyset & \text{if } q = q_0 \text{ and } a \in \Sigma \end{cases}$$

Proof of correctness would prove that $L(N') = A^$ by considering an arbitrary string accepted by N' , tracing an accepting computation of N' on it, and using that trace to prove the string can be written as the result of concatenating some number of strings, each of which is in A ; then, taking an arbitrary string in A^* and proving that it is accepted by N' . Details left for extra practice.*

Application: A state diagram for a NFA over $\Sigma = \{a, b\}$ that recognizes $L((\Sigma^*b)^*)$:

True or False: The state diagram of any DFA is also the state diagram of a NFA.

True or False: The state diagram of any NFA is also the state diagram of a DFA.

True or False: The formal definition $(Q, \Sigma, \delta, q_0, F)$ of any DFA is also the formal definition of a NFA.

True or False: The formal definition $(Q, \Sigma, \delta, q_0, F)$ of any NFA is also the formal definition of a DFA.

Week3 wednesday

Consider the state diagram of an NFA over $\{a, b\}$:



The language recognized by this NFA is

The state diagram of a DFA recognizing this same language is:

Suppose A is a language over an alphabet Σ . **Claim:** if there is a NFA N such that $L(N) = A$ then there is a DFA M such that $L(M) = A$.

Proof idea: States in M are “macro-states” – collections of states from N – that represent the set of possible states a computation of N might be in.

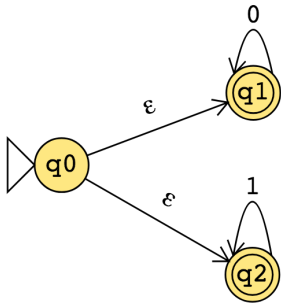
Formal construction: Let $N = (Q, \Sigma, \delta, q_0, F)$. Define

$$M = (\mathcal{P}(Q), \Sigma, \delta', q', \{X \subseteq Q \mid X \cap F \neq \emptyset\})$$

where $q' = \{q \in Q \mid q = q_0 \text{ or is accessible from } q_0 \text{ by spontaneous moves in } N\}$ and

$\delta'((X, x)) = \{q \in Q \mid q \in \delta(r, x) \text{ for some } r \in X \text{ or is accessible from such an } r \text{ by spontaneous moves in } N\}$

Consider the state diagram of an NFA over $\{0, 1\}$. Use the “macro-state” construction to find an equivalent DFA.



Prune this diagram to get an equivalent DFA with only the “macro-states” reachable from the start state.

Suppose A is a language over an alphabet Σ . **Claim:** if there is a regular expression R such that $L(R) = A$, then there is a NFA, let's call it N , such that $L(N) = A$.

Structural induction: Regular expression is built from basis regular expressions using inductive steps (union, concatenation, Kleene star symbols). Use constructions to mirror these in NFAs.

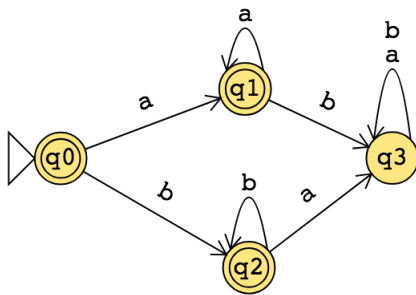
Application: A state diagram for a NFA over $\{a, b\}$ that recognizes $L(a^*(ab)^*)$:

Suppose A is a language over an alphabet Σ . **Claim:** if there is a DFA M such that $L(M) = A$, then there is a regular expression, let's call it R , such that $L(R) = A$.

Proof idea: Trace all possible paths from start state to accept state. Express labels of these paths as regular expressions, and union them all.

1. Add new start state with ε arrow to old start state.
2. Add new accept state with ε arrow from old accept states. Make old accept states non-accept.
3. Remove one (of the old) states at a time: modify regular expressions on arrows that went through removed state to restore language recognized by machine.

Application: Find a regular expression describing the language recognized by the DFA with state diagram



Conclusion: For each language L ,

There is a DFA that recognizes L $\exists M$ (M is a DFA and $L(M) = A$)
if and only if

There is a NFA that recognizes L $\exists N$ (N is a NFA and $L(N) = A$)
if and only if

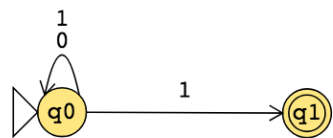
There is a regular expression that describes L $\exists R$ (R is a regular expression and $L(R) = A$)

A language is called **regular** when any (hence all) of the above three conditions are met.

Week2 friday

Nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$	
Finite set of states Q	Can be labelled by any collection of distinct names. Default: q_0, q_1, \dots
Alphabet Σ	Each input to the automaton is a string over Σ .
Arrow labels Σ_ϵ	$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.
Transition function δ	Arrows in the state diagram are labelled either by symbols from Σ or by ϵ $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ gives the set of possible next states for a transition from the current state upon reading a symbol or spontaneously moving.
Start state q_0	Element of Q . Each computation of the machine starts at the start state.
Accept (final) states F	$F \subseteq Q$.
M accepts the input string	if and only if there is a computation of M on the input string that processes the whole string and ends in an accept state.
Page 53	

The formal definition of the NFA over $\{0, 1\}$ given by this state diagram is:



The language over $\{0, 1\}$ recognized by this NFA is:

Change the transition function to get a different NFA which accepts the empty string.

The state diagram of an NFA over $\{a, b\}$ is below. The formal definition of this NFA is:



The language recognized by this NFA is: