CS181: Formal Languages and Automata Theory

Professor Campbell

Thilan Tran

Spring 2021

Contents

CS181: Formal Languages and Automata Theory

- what *defines* a computer?
  - input and output
  - storage ie. state information
  - computation or processing ie. is *programmable*
  - deterministic
  - sequential, performs one thing at a time
    - bounded action
  - problems it can solve, and what class of problems it can solve
- these are all attributes that are defined in models of computations
  - a model of computation represents the operation of a computer ie. an idealized computer
- there are different models of computation:
  1. a **deterministic finite automaton (DFA)** or **finite state machine (FSM)** can solve **finite state languages (FSLs)**
  2. a **context-free grammar (CFG)** can solve **context-free languages (CFLs)**
  3. Turing machine
  - there are problems that a computer cannot solve that are not addressed in any of these models
  - going up the ladder of models, we have:
    - increasing expressiveness or computational power
    - decreasing understandibility and predictability
      - some things are impossible to know about what the model does next

Deterministic Finite Automata

- general string definitions:
  - the **alphabet** is any nonempty, finite set of objects called **symbols** except the empty string ε
    - typically denoted by Σ
  - a **string** AKA word over an alphabet is a finite sequence of symbols from the alphabet:
    - ε is the empty string with length 0
    - note that a string of length 1 is *distinct* from the symbol it is
    - $|w|$ is the length or number of symbols in w
    - Σ^+ ie. the Kleene plus is the set of all *nonempty* words over the alphabet Σ
  - the concatenation operation on symbols and strings is defined with $\circ$
    - can be left off for brevity
  - for a symbol a, a^n is the string consisting of a concatenated to itself n times
    - a^0 is ε
  - one string is a **substring** of another if it appears consecutively in it
    - ε is a substring of any string, including itself, in any position
  - one string x is a **prefix** of another string y if $xz = y$ for some string z:
    - x is a **proper prefix** if $x \neq y$
    - ε is a proper prefix for all strings, except itself
    - analogous definitions for suffixes
  - the notation $\#(a, w)$ gives the number of occurrences of symbol a in the word w
  - a **language** over an alphabet Σ is any set of strings over Σ:
    - ie. a subset of Σ^+ or equivalently $\in \mathcal{P}(\Sigma^+)$
    - note that $\emptyset$ is a language over any alphabet
    - a language is **prefix-free** if no member is a proper prefix of another member
    - a **family** of languages is a set of languages over a common alphabet Σ
      - eg. the finite state languages are a family of languages
  - given a string w over some alphabet, w^R is w in reverse order
  - a **run** of symbol x in w is all the consecutive instances of x in w
    - eg. `aabcabbb` contains only the following runs `a, aa, b, bbb, c`
- more language operations for some languages A, B:
  - the **union** $A \cup B = \{x \mid x \in A \vee x \in B\}$
  - the **concatenation** $A \circ B = \{xy \mid x \in A \wedge y \in B\}$
    - eg. $\emptyset \circ B = \emptyset$, $\{\varepsilon\} \circ B = B$

  - the **star** $A^* = \{x_1 x_2 ... x_k \mid k \geq 0 \wedge x_i \in A \ \forall \ i\}$:
    * equivalently, $A^* = A^+ \cup \{\varepsilon\}$
    * note that the empty string ε is always a member of A^*
  - a collection of objects is **closed** under some operation if applying that operation to members returns an object still in the collection
    * eg. the family of regular languages is closed under these three language operations
- closure properties of the FSL family:
  - union, concatenation, Kleene star and Kleene plus
  - complementation and intersection
    * and thus set difference, symmetric set difference, etc. via deMorgan's laws
  - reversal
  - note that reversal and complementation are *also* closure properties on the non-FSL family
- languages can be used to represent computation problems:
  - eg. the language of arithmetic contains `2+2=4, 1-0=1`
    * and does not contain `2+2=11, 2+2, 2+ =7`
  - representing the concept of arithmetic using *only* strings and languages
    * the language *represents* the computational problem of arithmetic
  - note that while a DFA would be able to hold the format of the language of arithmetic, it is unable to capture the mathematical calculations required in the language
    * eg. specifying the semantics in addition to the syntax
- some graph theory:
  - any acyclic graph can be formulated as a tree
  - any node of a tree (regardless of how it is drawn) can be formulated as a root
    * ie. *"picking"* up the tree from that point
  - a **directed rooted tree** is a directed graph that is acyclic even when you view it as undirected, and has a designated root node:
    * can be defined as $(V, E, r), r \in V$
    * in addition, all edges are directed from the root towards the leaves
  - a **directed rooted k-ary tree** is a directed rooted tree where the out-degree of every node $\leq k$

Definitions

- defining basics of a **deterministic finite automata (DFA)** ie. FSM:
  - black box with inputs and outputs
    * **inputs** are a finite sequence of symbols chosen from a predefined

alphabet
    - binary **output**, accepted or rejected input, AKA a decider
  - black box encapsulates the program and storage, as well as the deterministic and sequential attributes
  - the **program** is defined in a directed graph with labels on the edges corresponding to the symbols of the alphabet:
    - AKA the **state diagram** of the DFA
    - has an **initial state** denoted by an arrow
    - since the output is binary, one of the states will be an **accepting state** denoted by an extra circle around the node
    - each node of the graph has exactly one outgoing edge per symbol in the alphabet
      - this specification makes the program deterministic
- formal definition of a DFA:
  - a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$
  - Q is a finite set called the **states**
  - Σ is a finite set called the **alphabet**
  - $\delta : Q \times \Sigma \to Q$ is the **transition function**:
    - this constrains the transition to specify exactly one transition in every state for each possible input symbol
      - by convention, if a DFA does not have any legal move to make at some step before reaching the end of the input string, the computation *blocks* and rejects
    - ie. $\delta(q, a) = q'$ can be interpreted as in state q on input a, go to state q'
    - an additional recursive definition is to extend δ to $\delta^* : Q \times \Sigma^* \to Q$
      - ie. recursively apply the transition function on the length of the input string
  - $q_0 \in Q$ is the **start state**
  - $F \subseteq Q$ is the set of **accept states** or final states
    - thus a DFA can have zero accept states
  - if A is the set of all strings that machine M accepts, A is the **language of machine** M and $L(M) = A$
    - M recognizes or accepts A
  - note that this formal definition can be entirely encapsulated in a state diagram
- formal definition of computation:
  - if we have a machine $M = (Q, \Sigma, \delta, q_0, F)$ and string $w = w_1 w_2 ... w_n$, M **accepts** w if a sequence of states $r_0, r_1, ..., r_n$ in Q exists with three conditions:
    1. $r_0 = q_0$
    2. $\delta(r_i, w_{i+1}) = r_{i+1} \ \ \forall \ \ i \in \{0, ..., n-1\}$
    3. $r_n \in F$

  - a language is a **regular language** if some finite automaton recognizes it
- the DFA is a good model for computers with extremely limited memory:
  - eg. elevator controllers, watches, dishwashers, etc.
  - requires memory proportional to the number of states it has to track
  - DFAs have a probabilistic counterpart called **Markov chains**

Proving Closure with DFAs

Ex. Prove the family of regular languages is closed under complementation ie. for a regular language A, $\overline{A}$ is also a regular language:

- for A, we have a DFA $M = (Q, \Sigma, \delta, q, F)$ that accepts A:
  - a DFA M' that accepts $\overline{A}$ can be constructed by setting $M' = (Q, \Sigma, \delta, q, (Q - F))$
  - ie. swap the accepting and rejecting states

Ex. Prove the family of regular languages is closed under the union operation ie. if A_1, A_2 are regular languages, so is $A_1 \cup A_2$:

1. Since the languages are regular, we have DFAs M_1, M_2 that recognize A_1, A_2 respectively, where $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$. Note that we are assuming the languages have the same alphabets. We want to construct an M to recognize $A_1 \cup A_2$ where $M = (Q, \Sigma, \delta, q_0, F)$.
2. The set of states Q can be defined as:

$$Q = \{(r_1, r_2) \mid r_1 \in Q_1 \wedge r_2 \in Q_2\}$$

   - each state needs to keep track of the state each machine would be in if it had read up to this point, so we use the Cartesian product $Q_1 \times Q_2$ for new states Q

3. The transition function δ can be defined as:

$$\delta((r_1, r_2), a) = (\delta(r_1, a), \delta(r_2, a)) \ \ \forall \ \ (r_1, r_2) \in Q, a \in \Sigma$$

   - move from one tuple of pairs to the next based on M_1, M_2

4. Set the start state $q_0 = (q_1, q_2)$.
5. The set of accept states F can be defined as:

$$F = \{(r_1, r_2) \mid r_1 \in F_1 \vee r_2 \in F_2\}$$

   - equivalently, $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$
   - if instead $F = F_1 \times F_2$, we have proved the same theorem for the intersection operation

Ex. Prove the family of regular languages is closed under the concatenation operation ie. if A_1, A_2 are regular languages, so is $A_1 \circ A_2$:

- try to use a similar approach to construct M from M_1, M_2:
  - M must accept if its input can be broken into two pieces accepted by M_1, M_2
  - however, M would not know where to break its input
    - we need a new technique called **nondeterminism** to complete this proof

Nondeterministic Finite Automata

- there are certain problems a DFA cannot solve:
  - let alphabet $\Sigma = \{a, b, \#\}$ and language $L = \{w\#w^R \mid w \in \{a, b\}^*\}$
  - can we write some DFA M that accepts Σ?
- if L were an FSL, then by definition, there must be a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that accepts L :
  - for an input $w\#w^R$, when M reads $w\#$, there are $2^{|w|}$ possible prefixes
    - at this point of input, the remaining possible symbols in the string that would be accepted are now *fixed*
  - if $2^{|w|} > |Q|$:
    - by the pigeonhole principle, for some two indefinitely long, different strings w_1, w_2, the DFA will end up in the *same* state q after reading $w_1\#$ or $w_2\#$, since we do not have as many states as prefixes
      - note that w_1 may be of different length of w_2
    - a DFA is deterministic ie. only depends on the state it's in and the following characters
    - thus, we have a situation where $w_1\#w_2$ would be *accepted* by the DFA!
  - since $w_1\#w_2 \notin L$, this is a contradiction, and no such DFA can exist
    - having finite states has limitations
  - however, note that this proof is not constructive and thus not general
- what if we adjust the formal definition for a DFA, specifically the transition function δ?
  - ie. make the machine into a **nondeterministic finite automata (NFA)**
  - when the machine is in a given state and reads the next input symbol, several choices may exist for the next state at any point
- formal definition of an NFA:
  - an NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$
  - Q, Σ, q_0, F remain the same from the DFA definition

  - $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the new **transition function**:
    * $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$
    * for a state and input symbol, the δ returns a *set* of states, thus allowing for non-determinism
    * in the state diagram, an NFA state may have zero, one, or many arrows exiting it with the *same* label:
      · we can visualize the NFA as *"splitting"* into multiple copies and thus following all the possibilities in parallel
      · Σ_ε allows the NFA to split at a stage without reading any further input
      · ie. a kind of parallel computation
    * edges can have ε as a label, ie. the NFA can split without reading additional input
  - interestingly, the set of languages recognizable by an NFA is exactly the set of languages representable by a DFA
    * gives no additional expressive power
  - note that although the NFA seems as if it could continue to split infinitely, and thus give more expressive power than a DFA, it is upper *bounded* since the input is still finite:
    * number of states the NFA is in can go up or down for each symbol read in, but it cannot increase infinitely
    * this is the essence of the proof of equivalence representations between DFAs and NFAs
- other interpretations of nondeterminism:
  1. search NFA graph for path from q_0 to any $q \in F$ through a BFS algorithm
  2. every time you come to a nondeterministic choice, consider both paths in parallel
     - ie. a tree of possible computations
  3. NFA *guesses* which choices will lead to accepting state, but it must check before accepting
     - ie. "guess-and-check"
  4. "choosy generator" that generates a string
- formal definition of computation by an NFA:
  - if we have an NFA $M = (Q, \Sigma, \delta, q_0, F)$ and string $w = w_1 w_2 \dots w_n$, M **accepts** w if we can write w as $w = y_1 y_2 \dots y_m$ where $y_i \in \Sigma_\varepsilon$ and there exists a sequence of states $r_0, r_1, \dots, r_m$ in Q with three conditions:
    1. $r_0 = q_0$
    2. $r_{i+1} \in \delta(r_i, w_{i+1}) \ \ \forall \ \ i \in \{0, \dots, m-1\}$
    3. $r_m \in F$
  - ie. if *any* of the states the NFA can get to by the end of the input is in F, the string will be accepted
  - if all of the states the NFA can get to for an input is not in F, the NFA

does not accept the string

Ex. Prove that every NFA has an equivalent DFA.

- want to convert the NFA into an equivalent DFA that simulates the NFA:
  - note that if an NFA has k states, it has 2^k subsets of states, which corresponds to each possibility that the DFA must remember
  - ie. DFA will need to have 2^k states

1. Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing some language A. We want to construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing A.
2. The set of states can be defined as:

$$Q' = P(Q)$$

3. For some state R of M, R is a set of states of N. When M reads a symbol a in R, it shows where a takes each state *in* R to move to a new *set* of states, so we can thus define the transition function as:

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$

   - note that $\delta(r, a)$ is a set and $\bigcup$ performs set union

4. The start state can be defined as:

$$q_0' = \{q_0\}$$

   - note that every state in M is a *set* of states from N

5. The set of accept states can be defined as:

$$F' = \{R \in Q' \mid R \text{ contains at least one accepting state of } N\}$$

6. Handle the ε arrows:

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{for some} r \in R\}$$

   - $E(R)$ contains all the states that can be reached from R by traveling along 0 or more ε arrows, thus including the members of R themselves
   - thus the transition function should travel to all states reachable along ε arrows
   - we need to also adjust the start state $q_0' = E(\{q_0\})$

Proving Closure with NFAs

Ex. Prove the family of regular languages is closed under the union operation ie. if A_1, A_2 are regular languages, so is $A_1 \cup A_2$:

- since DFAs and NFAs are equivalent, regular languages are languages that can be accepted by a DFA or NFA:
  - although previously proved using DFAs, this can be proved more easily using NFAs instead
  - we can use an NFA that nondeterministically guesses which of the two original machines accepts the input by reading ε into each machine after an initial state q_0

1. Since the languages are regular, we have NFAs M_1, M_2 that recognize A_1, A_2 respectively, where $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$. Note that we are assuming the languages have the same alphabets. We want to construct an M to recognize $A_1 \cup A_2$ where $M = (Q, \Sigma, \delta, q_0, F)$.
   - a different approach from the DFA proof where the states of M are taken from the pairs of states $Q_1 \times Q_2$, and the start state is (q_1, q_2)
     - ie. now that we have an NFA we can utilize ε to compose the two DFAs together
2. The set of states Q can be defined as:

$$Q = \{q_0\} \cup Q_1 \cup Q_2$$

3. The transition function δ can be defined as:

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \wedge a = \varepsilon \\ \emptyset & q = q_0 \wedge a \neq \varepsilon \end{cases}$$

   - an initial state leads off to either submachine automatically through an input of ε
   - once inside either submachine, continue to move through their states

4. Set the start state to q_0.
5. The set of accept states F can be defined as:

$$F = F_1 \cup F_2$$

   - ie. accept if either submachine accepts

- if we change the $\cup$ to $\cap$, the same proof trivially holds for the intersection operation

Ex. Prove the family of regular languages is closed under the concatenation operation ie. if A_1, A_2 are regular languages, so is $A_1 \circ A_2$:

- we can use an NFA that nondeterministically guesses where to split the input into two parts such that each is accepted by each machine

1. Since the languages are regular, we have NFAs M_1, M_2 that recognize A_1, A_2 respectively, where $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$. Note that we are assuming the languages have the same alphabets. We want to construct an M to recognize $A_1 \circ A_2$ where $M = (Q, \Sigma, \delta, q_0, F)$.
2. The set of states Q can be defined as:

$$Q = Q_1 \cup Q_2$$

   - no more initial state

3. The transition function δ can be defined as:

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

   - branch into M_2 when we reach the input constitutes a string in A_1
     - still need to split and stay in M_1 depending on the following input

4. Set the start state q_0 to the start state of M_1.
5. The set of accept states F is equal to the accept states of M_2, F_2.

   - accept if we have completely consumed the input into two substrings

Ex. Prove the family of regular languages is closed under the star operation ie. for a regular language A, A^* is also a regular language:

- we can use an NFA that nondeterministically repeatedly returns to the start state from the accepting state

1. Since the language is regular, we have an NFA $M = (Q_1, \Sigma, \delta_1, q_1, F_1)$ that recognizes A. We want to construct an M' to recognize A where $M' = (Q, \Sigma, \delta, q_0, F)$.
2. The set of states Q can be defined as:

$$Q = \{q_0\} \cup Q_1$$

- need a new start state to handle $\varepsilon \in A^*$

3. The transition function δ can be defined as:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \wedge q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \wedge a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \wedge a = \varepsilon \\ \emptyset & q = q_0 \wedge a \neq \varepsilon \\ \{q_1\} & q = q_0 \wedge a = \varepsilon \end{cases}$$

   - when processing gets to a state that M accepts, loop back to the start state to try and read another string from A

4. Set the start state to q_0.

5. The set of accept states F can be defined as:

$$F = \{q_0\} \cup F_1$$

   - need to accept ε, so q_0 is added

Regular Expressions

- R is a **regular expression** if R is one of the following:
  1. a
     - semantically, indicates $\{a \mid a \in \Sigma\}$
  2. ε
     - semantically, indicates $\{\varepsilon\}$
  3. $\emptyset$
     - semantically, indicates $\{\}$
  4. $(R_1 \cup R_2)$ where R_1, R_2 are regular expressions
  5. $(R_1 \circ R_2)$ where R_1, R_2 are regular expressions
     - can be abbreviated as $R_1 R_2$
  6. R_1^* where R_1 is a regular expression
  - recursively defined, allows us to define patterns of strings ie. languages
- we have the following identities for any regular expression R:
  - $R \cup \emptyset = R$
  - $R \circ \varepsilon = R$
- in fact, regular expressions and finite automata are equivalent in their descriptive power:
  - any regular expression can be converted into a finite automaton, and vice versa
  - the set of all regular languages is equivalent to the set of all FSLs

Ex. Prove if a language is described by a regular expression, then it is regular.

- if we have a regular expression R describing some language A, we can convert R into an NFA recognizing A:
  - thus A would be regular
  - consider each of the six cases in the definition of a regular expression

1. When $R = a$ for some $a \in \Sigma$, an NFA that recognizes the language is trivial:

$$\begin{aligned} N &= (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\}) \\ \delta(q_1, a) &= \{q_2\} \\ \delta(r, b) &= \emptyset \ \ \forall \ \ r \neq q_1 \vee b \neq a \end{aligned}$$

2. When $R = \varepsilon$, an NFA that recognizes the language is trivial:

$$\begin{aligned} N &= (\{q_1\}, \Sigma, \delta, q_1, \{q_1\}) \\ \delta(r, b) &= \emptyset \ \ \forall \ \ r, b \end{aligned}$$

3. When $R = \emptyset$, an NFA that recognizes the language is trivial:

$$N = (\{q\}, \Sigma, \delta, q, \emptyset)$$
$$\delta(r, b) = \emptyset \ \ \forall \ \ r, b$$

4. When R is composed of other regular expressions through $\cup, \circ, *$, we can construct an accepting NFA through the previous constructive proofs for closure of regular languages under the regular operations.

GNFAs

- in order to prove that all regular expressions are described by a regular expression, we need to find an algorithm to convert a DFA into an equivalent regular expression
  - to accomplish this, we introduce a new type of finite automaton called a **generalized nondeterministic finite automaton (GNFA)**
- a GNFA is an NFA wherein the transition arrows may have any regular expressions as labels, instead of only ε or members of Σ:
  - thus the GNFA reads blocks of symbols ie. substrings w^i from the input instead of one symbol at a time
    - the block of symbols constitutes a string described by the regular expression on a transition arrow
  - still nondeterministic, so the GNFA may have several different ways to process the same input string
  - accepts input if its processing causes the GNFA to be in an accept state at the end of the input
  - formally, a GNFA is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$
    - transition function specifies the regular expressions on each transition $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to R$
  - ie. an automata that supports pattern matching
- special GNA form by definition:
  - start state has transition arrows going to every other state but no arrows coming in from any other state
  - there is only a single accept state, and it has arrows coming in from every other state but no arrow to any other state
    - note that the accept state is not the start state
  - except for the start and accept states, ie. for all *internal* states, one arrow goes from every state to every other state and also from each state to itself
  - ie. GNA is almost a complete graph
- converting a DFA to GNFA in special form:

1. add a new start state with an ε transition to the old start state, and a new accept state with ε arrows from the old accept states
2. if any arrows have multiple labels or multiple arrows between the same two states, replace with a single arrow whose label is a union of the other labels
3. add arrows labeled $\emptyset$ between states that had no arrows

- converting a GNFA into a regular expression:
  - say the GNFA has $k \geq 2$ states, with a different start and accept state
  - if $k > 2$, we can construct an equivalent GNFA with $k - 1$ states
    - repeat until $k = 2$
  - otherwise, if $k = 2$, the GNFA has a single arrow from the start to accept state
    - that single label is the equivalent regular expression
  - this procedure hinges on the ability to always construct an equivalent GNFA with one fewer state when $k > 2$
- constructing an equivalent GNFA with one fewer state, given a state to be removed q_{rip} that is different from q_{start} and q_{accept} and two connections to and from it q_i, q_j:
  - for the following conditions:
    1. q_i goes to q_{rip} with an arrow labeled R_1
    2. q_{rip} goes to itself with an arrow labeled R_2
    3. q_{rip} goes to q_j with an arrow labeled R_3
    4. q_i goes to q_j with an arrow labeled R_4
  - we can remove the state q_{rip} and its connected arrows, and replace the arrow going from q_i to q_j with the following:

$$R_1 R_2^* R_3 \cup R_4$$

  - this change needs to be made for each arrow from any state q_i to q_j, including the case where $i = j$
    - specifically, $q_i \in Q' - \{q_{accept}\}$ and $q_j \in Q' - \{q_{start}\}$ where $Q' = Q - \{q_{rip}\}$

Ex. Prove that for any GNFA G, the regular expression obtained from running the above procedure recursively is equivalent to G.

- we can argue using induction, starting with the trivial base case where $k = 2$ states:
  - here, the regular expression on the single transition describes exactly how to move from the start to accept state
  - thus, that regular expression is exactly equivalent to G
- for the inductive step, we first assume that the claim is true for $k - 1$ states, and want to prove that it holds for k states as well:
  - to do this, we will show that G and the generated G' recognize the same language

  - assume that G enters a sequence of states for some input w, $q_{start}, q_1, q_2, \cdots, q_{accept}$
    * then, we have several possible cases for the same sequence of states in G'

1. if none of the sequence of states is the removed state q_{rip}, G' trivially accepts w
   - the new regular expressions labeling the arrows of G' *still* contains the old regular expression between each state as part of a union
2. if q_{rip} does appear, removing each run of consecutive q_{rip} states from the sequence forms an accepting computation for G' as follows:
   - the states q_i, q_j *bracketing* a run have a new regular expression on the arrows between them that describes *all* string taking q_i to q_j via q_{rip} on G
   - thus G' also accepts w
3. conversely, suppose that G' accepts an input w:
   - each arrow between any two states in G' describes the collection of strings between the same two states in G, either directly or via q_{rip}
   - this G must also accept w, and G and G' are equivalent

- thus the induction step holds, since we can recursively use the induction hypotheses to generate a regular expression for G
  - this also proves equivalence between the class of regular languages and regular expressions
- map of proof of equivalence between DFAs, NFAs, and regular expressions:
  - DFAs $\subseteq$ NFAs trivially
  - NFAs $\subseteq$ DFAs from proof by construction
  - NFAs $\subseteq$ GNFAs from proof by construction above
  - GNFAs $\subseteq$ regular expressions from proof by construction above
  - regular expressions $\subseteq$ NFAs from construction by composition

Nonregular Languages

- finite automata cannot be recognized by any finite automaton:
  - eg. to recognize the language $\{0^n1^n \mid n \geq 0\}$, a DFA would have to remember how many 0s have been seen so far in the input
  - potentially an infinite number of states required to keep track of the possibilities:
    - however, this is not sufficient to prove a language is irregular
    - some languages may require an unlimited number of possibilities but are actually regular
  - eg. over the alphabet $\Sigma = \{0, 1\}$:
    - the language $\{w \mid w$ has an equal number of 0s and 1s$\}$ is not regular
    - the language $\{w \mid w$ has an equal number of occurrences of 01 and 10$\}$ is regular
- the **pumping lemma**, used to prove a language is not regular, states the following:
  - if A is a regular language, then there is a number p ie. the **pumping length** where if s is any string in A of length at least p, then s may be divided into three pieces $s = xyz$ such that:
    1. $xy^iz \in A \ \ \forall \ \ i \geq 0$
    2. $|y| > 0$
    3. $|xy| \leq p$
  - note that either x or z may be ε, but not y
    - the third condition states that the pieces x, y together have length at most p
  - note that p depends only on A, ie. p does not depend on s
  - to use the lemma to prove a language is not regular, first assume the language is regular in order to obtain a contradiction
    - ie. find a string that has length p or greater that cannot be pumped

Ex. Proof of the pumping lemma.

- this proof hinges on the application of the **pigeonhole principle**:
  - for some string s with length n, the sequence of states it goes through in some DFA that accepts it has length $n + 1$
  - because n must be at least p, we know that $n + 1$ is greater than p
    - thus, the sequence *must* contain a repeated state, say q'
  - thus we can divide s into three pieces where x is the part of s appearing before q', y is the part between the two appearances of q', and piece z is the remaining part of s
  - this satisfies the pumping lemma as follows:

1. we can see that any input of the pattern xy^iz would be accepted by this DFA
2. $|y| > 0$
3. as long as q' is the first repetition in the sequence, the third condition holds
    * by the pigeonhole principle, the first $p+1$ states in the sequence must contain a repetition

Ex. Prove the language $A = \{0^n1^n \mid n \geq 0\}$ is not regular.

1. Assume by contradiction A is regular, and thus has a pumping length p.
2. Choose the string s to be 0^p1^p, which can be broken down into xyz where the string $xy^iz \in A$ for any $i \geq 0$ by the pumping lemma.
3. Consider the following three cases for xyz:
    - If the string y consists of only 0s, the string $xyyz$ has more 0s than 1s and so is not a member of A, which is a contradiction.
    - If the string y consists of only 1s, we get a similar contradiction.
    - If the string y consists of both 0s and 1s, the string $xyyz$ will have the same number of 0s and 1s, but they will be out of order with some 1s before 0s, which is not a member of A and another contradiction.
    - Note that the third condition of the pumping lemma could be used to eliminate the second and third cases. Thus we have that A is not regular.

Ex. Prove the language $A = \{w \mid w \text{ has an equal number of 0s and 1s}\}$ is not regular.

1. Assume by contradiction A is regular, and thus has a pumping length p.
2. Choose the string s to be 0^p1^p, which can be broken down into xyz where the string $xy^iz \in A$ for any $i \geq 0$ by the pumping lemma.
3. By the third pumping lemma condition, $|xy| \leq p$, so y must consist of only 0s. Thus $xyyz \notin A$, so s cannot be pumped. This is a contradiction, so A must not regular.

Ex. Prove the language $A = \{ww \mid w \in \{0,1\}^*\}$ is not regular.

1. Assume by contradiction A is regular, and thus has a pumping length p.
2. Choose the string s to be 0^p10^p1, which can be broken down into xyz where the string $xy^iz \in A$ for any $i \geq 0$ by the pumping lemma.
3. By the third pumping lemma condition, $|xy| \leq p$, so y must consist of only 0s. Thus $xyyz \notin A$, so s cannot be pumped. This is a contradiction, so A must not be regular.

- Note that the choosing an s that violates the pumping lemma entails choosing a string that exhibits the *"essence"* of the nonregularity of the language.

Ex. Prove the language $A = \{1^{n^2} \mid n \geq 0\}$ is not regular.

1. Assume by contradiction A is regular, and thus has a pumping length p.
2. Choose the string s to be 1^{p^2}, which can be broken down into xyz where the string $xy^iz \in A$ for any $i \geq 0$ by the pumping lemma.
3. Consider the two string xyz and xy^2z, whose lengths differ by $|y|$.
    - By condition 3, $|xy| \leq p$ and thus $|y| \leq p$.
    - Then we have $|xyz| = p^2$ and $|xy^2z| \leq p^2 + p$.
    - However, $p^2 + p < (p+1)^2$, and condition 2 implies y is not empty.
    - This is a contradiction since the length of xy^2z lies strictly between two consecutive perfect squares, and cannot be a perfect square itself. Thus A is not regular.

Ex. Prove the language $A = \{0^i1^j \mid i > j\}$ is not regular.

1. Assume by contradiction A is regular, and thus has a pumping length p.
2. Choose the string s to be $0^{p+1}1^p$, which can be broken down into xyz where the string $xy^iz \in A$ for any $i \geq 0$ by the pumping lemma.
3. Rather than pumping *up*, we can try and pump *down* instead.
    - By condition 3, y consists of only 0s.
    - The string $xy^0z = xz$ thus decreases the number of 0s in s.
    - Since s only had one more zero to begin with, this is a contradiction since sz does not have more 0s than 1s. Thus A is not regular.
    - Note that if we were to try and pump up, we would only end up increasing the number of 0s in the string, and would be unable to reach a contradiction.

- tips for using the pumping lemma to prove a language is not a FSL:
    - p can only be named
    - select an approproiate string s, and ensuring it is in L
    - correctly using the constraints on x, y, z imposed by the lemma
    - showing that *all* possible cases of x, y, z have been covered
        - cannot assume how s is divided by x, y, z other than the lemma's constraints
    - using sound and complete logic in each case to show that s' cannot be in L

Context-Free Grammars

- **context-free grammars (CFGs)** are a representation of languages with roots in linguistics:
  - more powerful than finite automata or regular expressions
  - motivated by an attempt to express human language in mathematical terms
    - however, CFGs are not enough to represent intricacies and full meanings of human languages
  - however, grammars can represent most features of human language syntax and artificial languages
    - ie. useful for specification and compilation of programming languages
  - CFGs define the family of **context-free languages (CFLs)**
    - the family of FSLs is a proper subset of the family of CFLs
- formal definition for a grammar:
  - consists of an **alphabet** Σ AKA **terminals**, **variables** V AKA **nonterminals**, **substitution rules** R AKA **productions**, and a **start variable** $S \in V$
    - the rules have the form $A \to \alpha$ where $A \in V, \alpha \in (\varepsilon \cup V)^*$
  - the *context-free* part of the grammar comes from the fact that we can apply any rule for a given variable anywhere it appears
  - ex. the grammar with the single rule $S \to SS \mid (S) \mid ()$
    - one variable S that is also the start variable
    - the alphabet is $\Sigma = \{(,)\}$
    - the represented language is the one with balanced ie. matching parentheses, not allowing ε
    - eg. represents strings such as `(())`, `()()`
      - `S ⇒ (S) ⇒ (())`
      - `S ⇒ SS ⇒ ()S ⇒ ()()`
- the process of creating a string from rules is called a **derivation**:
  - if we always choose the left-most variable to rewrite, the derivation is called **left-most**:
    - analogously for **right-most** derivations
    - every left-most derivation has a corresponding right-most derivation that gives the same parse tree
    - for any given string, for one left-most derivation, it corresponds to one parse tree which then corresponds to one right-most derivation

        - however there may be multiple corresponding derivations / trees for the same string
      - ie. there is a bijection between left-most and right-most derivations, and between a derivation and its parse tree
  - left-most derivations fill in terminal symbols from the left, while right-most derivations fill in terminal symbols from the right
  - convention to underline the variable to be replaced in each step of a derivation
  - each string created in the process is called a **sentential form**

- on the other hand, **parsing** or **reducing** is the process of determing if a string belongs to a language by working backwards given the string

- formal derivation steps:

  1. write down start variable, by convention LHS of the top rule
  2. find a variable and a matching production, and replace the variable with the RHS
  3. repeat step 2 until no variables remain

- if we have rule $A \to w$, we can say that uAv **yields** uwv ie. $uAv \Rightarrow uwv$:

  - we can say that u **derives** v ie. $u \Rightarrow^* v$ if $u = v$ or if a sequence $u_{1,2}, \dots, u_k$ exists such that $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v$
  - the language of the grammar is thus $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$

- **Chomsky normal form** is a CFG where every rule is of the form:

  - $A \to BC$ or $A \to a$
  - where A, B, C are any variables, except B, C may not be the start symbol, and a is any terminal
  - note that the rule $S \to \varepsilon$ is permitted, where S is the start variable

- any CFG can be converted into Chomsky normal form as follows:

  1. add a new start variable S_0 and the rule $S_0 \to S$
  2. handle ε-rules $A \to \varepsilon$ where A is not the start symbol:
     - remove these rules, and then for each occurrence of an A on the RHS of a rule, add a new rule with that occurrence deleted
     - eg. if $R \to uAv$ is a rule, we add the rule $R \to uv$
  3. handle all unit rules $A \to B$:
     - remove these rules, and then whenever a rule $B \to u$ appears, add the rule $A \to u$
  4. convert all remaining rules into proper form:
     - for each rule $A \to u_1 u_2 \dots u_k$ where $k \geq 3$, replace it with the rules $A \to u_1 A_1, A_1 \to u_2 A_2, \dots, A_{k-2} \to u_{k-1} u_k$

  - u, v represent any strings of variables and terminals

Ambiguity

- a grammar may generate the same string in different ways:
  - eg. undesirable for certain scenarios in programming languages
  - eg. for the grammar $E \to E + E \mid E \times E \mid (E) | a$:
    - `a+aXa` can be generated ambiguously
    - `E → EXE → a+aXa` vs. `E → E+E → a+aXa`
- a string w is derived **ambiguously** in a CFG G if it has two or more different left-most derivations:
  - ie. there is more than one parse tree for the string
  - G is **ambiguous** if it generates some string ambiguously
- some CFLs can *only* be generated by ambiguous grammars and are **inherently ambiguous**
  - eg. $\{a^i b^j c^k \mid i = j \vee j = k\}$
    - whenever $i = j = k$, strings will always be generated ambiguously in *any* grammar in the language
- other examples of ambiguity:
  - the rule $S \to SS \mid x \mid \varepsilon$ leads to ambiguity
  - the rule $S \to xS \mid x \mid \varepsilon$ is potentially unambiguous
  - the rule $R \to AR \mid RB | \varepsilon$ leads to ambiguity
  - the rule $R \to AR \mid x, x \to xB \mid \varepsilon$ is potentially unambiguous

Designing CFGs

- many CFLs are the union of simpler CFLs
  - ex. we can get the grammar for the language $\{0^n 1^n\} \cup \{1^n 0^n\}$ from simpler grammars:
    - $S_1 \to 0S_1 1 \mid \varepsilon$
    - $S_2 \to 1S_2 0 \mid \varepsilon$
    - $S \to S_1 \mid S_2$
- construction a CFG for a language is simple given a DFA for that language:
  1. make a variable R_i for each state q_i of the DFA
  2. add the rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA
  3. add the rule $R_i \to \varepsilon$ if q_i is an accept state of the DFA
  4. the start variable R_0 corresponds to the start state q_0
- CFGs may contain strings with two substrings that are *linked* in that a machine for such a language would need to remember an unbounded amount of

information for one substring to apply to the other:

  - eg. $\{0^n 1^n\}$
  - we can construct a CFG to handle this situation by using a rule of the form $R \to uRv$
    * generates strings where the portion containing u's corresponds to the portion containing v's

- recursive structures can be represented by placing the variable symbols inside the RHS of productions
- design patterns for CFGs:
  - sequencing patterns:
    * $S \to AB \mid CD$
  - iterative patterns:
    * right associative eg. $B \to AB \mid T$ generates string of the pattern $AA \ldots T$
    * left associative eg. $A \to AB \mid T$ generates string of the pattern $TB \ldots B$
    * "freely associative" eg. $S \to SS \mid T$
  - recursive / nesting patterns:
    * $B \to ABC \mid T$
      · this pattern is generated "outside-in"
    * covers the iterative patterns as well through **tail recursion** or **head recursion**
    * note that what distinguishes CFLs from FSLs is this final nesting pattern

Closure Properties

- closure properties for CFLs:
  - concatenation
    * stack a rule
  - Kleene star and Kleene plus
    * use the iteration design pattern
  - union
    * add additional rules
  - reverse
    * write all RHS of rules backwards
- non-closure properties for CFLs (unlike FSLs):
  - intersection:
    * let $L_1 = \{a^n b^n c^m\}$ and $L_2 = \{a^m b^n c^n\}$

    - each are CFLs, but the intersection $L_1 \cap L_2 = \{a^n b^n c^n\}$ is *not*
  - complementation:
    - $L_3 = \{ww \mid w \in \{a,b\}^*\}$ is not context-free, but its complement *is*!
    - $S \to A \mid B \mid AB \mid BA$ is a CFG for $\overline{L_3}$
      - where A and B generate words of odd length with a and b in the center respectively

CFLs and Non-CFLs

- some known CFLs:
  - $\{a^n b^m \mid m = n\}$ or $m > n, m \leq n, m \neq n$
  - $\{a^n b c^n\}$
  - $\{a^n b^{2n}\}$
  - $\{w \mid w \in \{a,b\}^*$ where $\#(a,w) = \#(b,w)\}$
  - $\{ww^R \mid w \in \Sigma^*\}$ AKA even palindromes
    - $\{w \in \Sigma^* \mid w = w^R\}$ AKA all palindromes
  - $\{a^i b^j c^k \mid i = j \vee i = k\}$ and other similar combinations eg. $i = j \vee i \neq k$
  - $\{a^i b^j c^k d^l \mid i = j \wedge k = l\}$:
    - as well as $i = l \wedge j = k$
    - these groups can be handled separately in a CFG with no constraint *between* them
- some known non-CFLs:
  - $\{a^n b^n c^n\}$
  - $\{a^i b^j c^k d^l \mid i = k \wedge j = l\}$:
    - has to do with the way the symbols are interleaved and *not* nested
    - interleaved groups are constrained
  - $\{a^n b^n c^m d^m \mid n > m\}$ etc.
    - adding a constraint across the groups
  - $\{a^n \mid n$ is a prime number$\}$
  - $\{a^n b^m \mid m = n^2\}$
  - $\{ww \mid w \in \Sigma^*\}$
- the **pumping lemma for CFLs** is used to show that a language is not a CFL, and states the following:
  - if L is a CFL over Σ, then there exists a number p depending only on L such that $s = uvxyz$ and $u, v, x, y, z \in \Sigma^*$ where:
    1. $uv^i xy^i z \in L \ \forall \ i \geq 0$
    2. $|vy| > 0$ ie. $|v| > 0$ or $|y| > 0$ or both
    3. $|vxy| \leq p$
       - note that the substring vxy could be anywhere in s

Ex. Proof of the pumping lemma for CFLs.

- again hinges on the application of the pigeonhole principle:
  - for a very long string in a CFL, it has a corresponding tall parse tree
  - for some long path from the start variable to one of the terminal symbols, some variable symbol R must repeat due to the pigeonhole principle
    - allows us to replace the subtree under the second occurrence with the subtree under the first occurrence, and thus get the pattern uv^ixy^iz

Ex. Prove the language $L = \{a^nb^nc^n \mid n \geq 0\}$ is not a CFL.

1. Assume by contradiction L is a CFL and thus has a pumping length p.
2. Choose the string $s = a^pb^pc^p$. Since $|S| > p$, we can write $s = uvxyz$ where $uv^ixy^iz \in L$ for any $i \geq 0$. We know that $|vy| \geq 1, |vxy| \leq p$.
3. Consider the following possible cases where vxy could fall:
   - could fall entirely in the block of a, b, or c
   - could straddle two of the blocks
   - could actually formulate the argument as a single general case
     - vxy can straddle at *most* two blocks, therefore when we create $s' = uv^ixy^iz, i \neq 1$, there will always be one unaffected block, so that block will habe p symbols, at least one other block won't

Push Down Store Automaton

- the family of CFLs can also be represented by a machine-like model called a **push down store automaton (PDA)**:
  - essentially an NFA with an extra component
  - need a way to store information that can expand with more input
  - needs to be well suited to handling nesting
  - "push down store" is synonomous with a LIFO stack, which summarizes the PDA functionality through an extra component called the **stack**:
    - PDA can only see top of stack, and to see symbols underneath it must pop the top of the stack
    - this gets to why the PDA cannot handle interleaving, the stack has been fully popped off and is a *clean* state
  - in mathematics, the stack is held as a string:
    - stack is a string of symbols from the stack alphabet Γ
    - initial stack is ε
    - the "tape" ie. string is semi-infinite
  - in contrast, the input tape is finite-length but can be indefinitely long
  - similar to an NFA, if any computation goes into or passes through an accepting state after reading the last symbol of the input string, we accept
    - otherwise, not
- possible moves at each step:
  - input tape operations:
    - read one symbol from the tape
    - ignore input tape AKA ε-move
  - stack operations:
    - pop the top of the stack symbol
    - push a new symbol on top of the stack
    - replace the top of the stack with a different symbol
    - nothing, leave the stack alone
  - then, move to the next state, nondeterministically
- the PDA **program** is a labeled digraph similar to an NFA, but we need to add formal notation for stack opreations:
  - the transition label $w_i, a \to b$ from q_1 to q_2 indicates that:
    1. for an input symbol $w_i \in \Sigma \cup \{\varepsilon\}$, read w_i from the input tape
       - if ε, ignore input ie. don't even look at the input tape
    2. read the top of the stack symbol $a \in \Gamma \cup \{\varepsilon\}$ and replace it with stack symbol $b \in \Gamma \cup \{\varepsilon\}$:
       - if $a = \varepsilon$, we ignore the current top of the stack ie. don't even look at the stack

        * we alllow the shorthand where $b \in \Gamma^*$ meaning the PDA replaces a with b where the first symbol of b is the new top of stack
      3. go to next state q_2
  - δ thus takes q_1, w_i, a as input and gives q_2, b as output
    * replacing a or b with ε can produce pushes, pops, or a no-op
  - note that we cannot explicitly test for an empty stack
- formal definition of a PDA:
  - a PDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$
    * Q, Σ, Γ, F are all finite sets
  - Q, Σ, q_0, F remain the same from the DFA and NFA definition
  - Γ is the stack alphabet
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function
- formal definition of computation by a PDA:
  - if we have PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ and string w, M accepts w if we can write w as $w = w_1 w_2 ... w_m$ where $w_i \in \Sigma_\varepsilon$ and there existsa sequence of states $r_0, r_1, ..., r_m \in Q$ and strings $s_0, s_1, ..., s_m \in \Gamma^*$ with three conditions:
    1. $r_0 = q_0$ and $s_0 = \varepsilon$ ie. starts in start state with an empty stack
    2. for $i = 0, ..., m-1$ we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Sigma_\varepsilon$ and $t \in \Gamma^*$
    3. $r_m \in F$
- stack operations:
  - popping is represented by $a \to \varepsilon$, as long as a is on the top
    * the shorthand $\Gamma \to \varepsilon$ always pops the top of the stack
  - pushing is represented by $\varepsilon \to b$
  - swapping is represented by $a \to b$
  - peeking is represented by $a \to a$
  - the stack no-op is represented by $\varepsilon \to \varepsilon$

Ex. For $\Sigma = \{(,)\}$, a PDA for the language L with balanced parentheses including ε can be defined as follows.

- ε , $\varepsilon \to \$$ transition between q_0, q
  - push the $ symbol, ignoring input, to mark the beginning of a substring of balanced parentheses
- (, $\varepsilon \to)$ transition between q and itself
  - push an open parentheses for each open parentheses in the input, ignoring the top of the stack
-) , $(\to \varepsilon$ transition between q and itself
  - pop an open parentheses from the stack for each closing parentheses
- ε , $\$ \to \varepsilon$ transition between q to q_0:
  - reset to the beginning of balanced parentheses
  - can be followed by more balanced parentheses

- note that the only nondeterministic choice in this PDA occurs from state q
  - could either read (or ε (after a sequence of perfectly balanced parentheses)

Ex. General pseudocode for a PDA that accepts the language $L = \{a^i b^j \mid i \geq j\}$ is as follows.

1. mark the bottom of the stack
   - ε should be accepted, but the first symbol cannot be b
2. for each input symbol a, push a counter onto the stack until input is b
3. for each input symbol b, pop one counter off the stack until we see an a or reach the end of string:
   - if we reach the end of input before seeing the bottom of the stack marker, we accept ie. $n > m$
   - if we reach the end of input and see the bottom of the stack marker, we accept ie. $n = m$
   - if we see the bottom of stack marker but there is remaining input, we do not accept

- similarly, we can see that the PDA can recognize the nonregular language $\{0^n 1^n\}$

Equivalence with CFGs

Ex. Prove if a language A is context-free with a corresponding CFG G, then some PDA P recognizes it.

- idea is to design the PDA to determine whether some series of substitutions using the rules of G can lead from the start variable to the input string:
  - we can exploit the nondeterminism in the PDA to guess which rule to substitute
  - when the PDA arrives at a string during substitution that contains only terminal symbols, it should check if this is identical to the input to possibly accept
  - one difficulty is storing intermediate strings between substitutions:
    - cannot use the stack for the entire string since the PDA can only access the top symbol on the stack
    - instead, only keep part of the intermediate string on the stack ie. the symbols starting with the first variable in the intermediate string
    - any terminals before the first variable are matched immediately with symbols in the input string
- informal description of P:

1. place the marker symbols \$ and the start variable on the stack
2. repeat the following:
   - if the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the RHS of the rule
   - if the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a
     * if they do not match, reject on this branch of the nondeterminism
   - if top of stack is the symbol \$ enter the accept state
     * this step accepts the input if it has all been read
   - the shorthand used to push an entire string onto the stack at the same time is useful in this construction
     * without the shorthand, would use a series of states with $(q_i, \varepsilon, \varepsilon)$ that feed the string onto the stack

- formal description of P:
  - $Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$
    * E is the set of states used for implementing the shorthand for pushing a string onto the stack
  - $q_0 = q_{start}$
  - $F = \{q_{accept}\}$
  - $\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_{loop}, S\$)\}$
  - $\delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w) \mid A \to w \in R\}$
    * R is the rule set for G
  - $\delta(q_{loop}, a, a) = \{(q_{loop}, \varepsilon)\}$ for all terminals a
  - $\delta(q_{loop}, \varepsilon, \$) = \{(q_{accept}, \varepsilon)\}$

Ex. Prove if a PDA P recognizes some language A, it is context-free.

- simplification of P for proof purposes:
  - P has a single accept state, trivial to enforce
  - P empties its stack before accepting, trivial to enforce
  - each transition in P either pushes or pops a symbol, but not both
    * replace each transition that pops and pushes with a two transition sequence
    * replace each transitions that does neither with a two transition sequence that pushes and pops an arbitrary stack symbol
- we want to build a CFG G that generates all strings P accepts:
  - for each pair of states p, q in P, G should have a variable A_{pq}:
    * A_{pq} should generate all strings that can take P from p with an empty stack to q with an empty stack ie. alternatively leaving the stack at q in the same condition as at p
    * note that since the stack starts and ends empty, the first move by P that generates one of these desired strings x must be a push and

the last must be a pop
  - two possibilities occur during P's computation on x:
    1. the symbol popped at the end is the symbol that was pushed at the beginning:
       * stack is *only* empty at the beginning and end of the computation on x
       * can be represented by the rule $A_{pq} \to aA_{rs}b$, where a and b are the inputs read at the first and last move, respectively, r is the state following p, and s is the state preceding q
    2. the symbol popped at the end is *not* the symbol pushed at the beginning:
       * the initially pushed symbol must get popped at some point before the end, thus making the stack empty at this point
       * can be represented by the rule $A_{pq} \to A_{pr}A_{rq}$ where r is the state when the stack becomes empty
- formally, for a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$, in order to construct G:
  - the variables are $\{A_{pq} \mid p, q \in Q\}$
  - start variable is $A_{q_0,q_{accept}}$
  - rules can be described as follows:
    * for each $p, q, r, s \in Q, u \in \Gamma, a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ε), put the rule $A_{pq} \to aA_{rs}b$
    * for each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr}A_{rq}$
    * for each $p \in Q$, put the rule $A_{pp} \to \varepsilon$
- thus we have shown that PDAs recognize the class of CFLs

Deterministic PDAs

- the **deterministic PDA (DPDA)** is a special PDA that accepts languages from the family of **deterministic CFLs (DCFLs)**:
  - unlike DFAs and NFAs, PDAs and DPDA are *not* equivalent
  - in addition, the class of inherently ambiguous CFLs is *disjoint* from the class of languages represented by DPDAs
    * the intuition behind this comes from the fact that a DPDA has exactly one legal move at every step ie. there is no ambiguity in its options
- difference with the PDA:
  - transition function can return $\emptyset$ instead of only a pair from $Q \times \Gamma_\varepsilon$
    * $\emptyset$ indicates there is no legal move for some state given an input and top of stack
  - a fully specified DPDA has *exactly* one legal move at every step:
    * ie. exactly one of the four possible cases for the arguments of δ is

not $\emptyset$

- formal definition of a DPDA:
  - a PDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$
    - Q, Σ, Γ, F are all finite sets
  - Q, Σ, q_0, F remain the same from the DFA and NFA definition
  - Γ is the stack alphabet
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ is the transition function
  - for every $q \in Q, a \in \Sigma, x \in \Gamma$, exactly one of the following values must not be $\emptyset$:

$$\delta(q, a, x), \delta(q, a, \varepsilon), \delta(q, \varepsilon, x), \delta(q, \varepsilon, \varepsilon)$$

Ex. For $\Sigma = \{(,)\}$, a DPDA for the language L with balanced parentheses including ε can be defined as follows.

- idea is to use a special identifier $[$ on the stack to denote the beginning of a string of balanced ie. matched parentheses
  - this point is where nondeterminism in our original PDA implementation stems from
- $($, $\varepsilon \to [$ transition between q_0, q
  - mark the beginning of a sequence of balanced parentheses with a special identifier
- $($, $\varepsilon \to ($ transition between q and itself
  - push an open parentheses for each open parentheses in the input, ignoring the top of the stack
- $)$, $(\to \varepsilon$ transition between q and itself
  - pop an open parentheses from the stack for each closing parentheses
- $)$, $[\to \varepsilon$ transition between q to q_0
  - reset to the beginning of balanced parentheses upon seeing the special identifier
- no more nondeterminism
  - two branches from q upon reading input $)$, but the transitions depend on different tops of the stack
- examples of DCFLs:
  - $\{a^i b^j \mid j = i\}$, push one counter for each a
  - $\{a^i b^j \mid j = 2i\}$, push two counters for each a
- examples of non-DCFLs:
  - $\{a^i b^j \mid j = i \vee j = 2i\}$:
    - when reading the a's, we cannot tell whether to push one counter or two counters for the future j's
    - note that DCFLs are *not* closed over union
  - note that unlike the DFA and NFA models, DCFLs and CFLs are *not* equivalent
    - no trick to make the DCFL represent the stack complexity of a CFL

Table 1: Comparison of Closure Properties

Operations	FSLs	CFLs	DCFLs
$\cup$	Y	Y	N
L^*, L^+	Y	Y	N
$\cap$	Y	N	N
$\overline{L}$	Y	N	Y
L^R	Y	Y	N

- proof idea for closure of DCFLs under complementation:
  - similar to DFAs, we can swap the accept and non-accept states
  - except for the special case when the DPDA enters both accept and non-accept states at the end of the input string
    - can modify the DPDA by only entering an accepting state when it is about to read the next symbol
  - unlike regular CFLs, where swapping the accept and non-accept states does not result in another CFL due to the nondeterminism of a PDA
    - if a nondeterministic model gets into a non-accepting state, there may be a parallel computation that does get accepted
- note that DCFLs are closed under complementation, but *not* under reversal:
  - consider the language $L = \{a^i b^j ? \mid j = i\} \cup \{a^i b^j . \mid j = 2i\}$
  - L is *not* a DCFL, while L^R *is* a DCFL

DCFGs

- **deterministic context-free grammars (DCFGs)** are the grammars associated with DPDAs:
  - these models are equivalent in power, provided only **endmarked** languages are considered ie. where all strings are terminated with some special symbol $\dashv$
    - note that a language A is a DCFL if and only if $A \dashv$ is a DCFL
  - to define determinism in a grammar, we want to constrain derivations
    - computations in automata are analogous to derivations in grammars
- derivations in CFGs begin with the start variable and proceed top-down with a series of substitutions:
  - on the other hand, for defining DCFGs, we take a bottom-up approach to employ a series of reduce steps until reaching the start variable
  - each **reduce step** is a reversed substitution where we replace a string of terminals and variables with the corresponding LHS of a rule

        - the string replaced is the **reducing string** and the entire reversed derivation is a **reduction**
    - in a **leftmost reduction**, at every reduction step, each reducing string is reduced only after all other reducing strings that lie *entirely* to its left *in that reduction process* have been replaced:
        - leftmost reductions correspond to rightmost derivations, and vice versa
    - a **valid string** is a string that appears in a leftmost reduction of some string in a language
- Ex. given the grammar $S \to SS \mid (S) \mid \varepsilon$, perform reductions on the following strings:
    - for the string $()()$, we have a leftmost reduction as follows:
        - $()() \to (\varepsilon)() \to (S)() \to S() \to S(\varepsilon) \to S(S) \to SS \to S$
        - this corresponds with a similar rightmost derivation, in reverse
- a reduction from u to v is a sequence of reducing steps:
    - in a CFG, we can have a leftmost reduction from a string w to the start variable S
    - in the reduced form $u_i = xhy$ and subsequent reduced step $u_{i+1} = xTy$, h together with its reducing rule $T \to h$ is a **handle** of u_i
        - AKA xhy *reduces* to xTy
    - consider ambiguous grammars, where some strings have several handles:
        - selecting a specific handle may require advance knowledge of which parse tree derives the string, which is unavailable to the DPDA
    - to get unambiguity, we stipulate that each step of reduction determines the next:
        - thus w determines the entire leftmost reduction
        - ie. unambiguous grammars generate strings by one parse tree only and therefore have unique handles
    - to get determinism, we need that the next reduce step must be *uniquely* determined by the prefix up through and including the reducing string of that reduce step:
        - ie. a leftmost reduce step doesn't depend on the symbols to the right of its reducing string
        - ie. if we are reading a valid string from left to right, as soon as we read the handle we know we have it
            - this type of handle is a **forced handle**
    - formally, a forced handle h of a string $v = xhy$ is the unique handle in *every* valid string $xh\hat{y}, \hat{y} \in \Sigma^*$
    - thus a DCFG is a CFG such that every valid string has a forced handle
- reductions vs. derivations:
    - in derivations, we have two choices:

        - which variable to replace
        - which rule to apply
    - in reductions, we have a different set of choices:
        - which string h to replace with a variable ie. which reducing string to choose
            - the possible reducing strings in a step can overlap
        - which rule to apply (backwards)
- **LR(k)** grammars are CFGs such that the handle of every valid string is forced by lookahead k:
    - thus a DCFG is the same as an LR(0) grammar
    - however, for every k, we can convert LR(k) grammars into DPDAs:
        - thus LR(k) grammars are equivalent in power for all k and describe exactly the DCFls

Turing Machines

- **Turing machines (TMs)** accept a class of languages called the **recursively enumerable (RE)** languages:
  - the class of **recursive languages** is a subset of the RE languages
  - recursive languages include the CFLs and FSLs as simple subsets
- comparison with other automata:
  - DFAs and NFAs have access to a read-only input tape
  - PDAs and DPDAs have access to a read-only input tape as well as a read-write, restricted, stack data structure
  - even if we could move the head of the input tape back and forth, this does not give additional representative power
  - however, by allowing a language to read *and* write to the input tape ie. work tape, we do gain additional power:
    - no longer has directional restrictions of the stack, but is limited in finite space
    - even *without* a stack and with this limited finite space on the work tape, has more representative power than CFGs
      - however, this limited finite space makes this hypothetical model less powerful than a normal TM
- to begin with, we will consider deterministic TMs:
  - single tape TMs
  - multi-tape TMs
- a **single tape TM** has the following:
  - a start state q_0, a single accepting state q_{accept}, and a single rejecting state q_{reject}:
    - there are no outgoing states for the accepting and rejecting states
    - if we ever enter either of these states, we are done with the computation
  - a collection of other intermediate states and their associated *deterministic* transitions
  - a read-write **work tape** that is *indefinitely* long:
    - the initial contents of the work tape is the input string w
      - note that we can read and write beyond the end of w
    - ie. input tape is the same as the work tape
  - the work tape contains:
    - symbols from the input alphabet Σ
    - symbols from the work tape alphabet $\Gamma \supseteq \Sigma \cup \{\textcyr{b}\}$
      - $\textcyr{b}$ represents a blank symbol
- formal definition of a turing machine (with a single tape):
  - TM is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

    - Q, Σ, Γ are all finite sets
  - Q is the set of states
  - Σ is the input alphabet not containing $\textvisiblespace$
  - Γ is the tape alphabet, where $\textvisiblespace \in \Gamma$ and $\Sigma \subseteq \Gamma$
  - $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function:
    - when the machine is in a certain state q and the tape head is over a tape symbol a, if $\delta(q, a) = (r, b, L)$, the machine writes b to replace a and goes to r
    - the third component L or R indicates a move to the left or right, respectively
  - $q_0 \in Q$ is the start state
  - $q_{accept} \in Q$ is the accept state
  - $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$
- a TM M computes as follows:
  1. initially, M receives its input $w = w_1 ... w_n \in \Sigma^*$ on the leftmost n squares of the work tape, and the rest of the infinite tape is filled with $\textvisiblespace$:
     - the head starts on the leftmost square of the tape
     - note that the first blank on the tape marks the end of the input
  2. the computation will then proceed according to the rules described in the transition function
     - if M ever tries to move its head to the left off the end of the state, the head stays in the same place for that move
  3. the computation continues until it enters either the accept or reject state, at which point it halts
     - if neither occurs M goes on forever
  - a setting of the current state, tape contents, and current head location is called a **configuration** of the TM:
    - configurations can be written as uqv, where the current stack contents is uv, the current state is q, and the current head location is the first symbol of v
    - the start configuration of M on input w is $q_0 w$
  - then, we can define how a configuration is yielded from the transition as follows:
    - if $\delta(q_i, b) = (q_j, c, L)$, the configuration uaq_ibv yields uq_jacv
    - if $\delta(q_i, b) = (q_j, c, R)$, the configuration uaq_ibv yields $uacq_jv$
- a **multi-tape TM** is the same as the single tape TM, except:
  - has multiple work tapes
    - ie. in addition to the input tape, the TM has access to the other initially blank work tapes
  - note that there are a fixed finite number of tapes for a multi-tape TM
    - however, each tape has unbounded memory
  - it turns out that single tape TMs and multi-tape TMs are equivalent

        - both end up using unbounded memory, but finite memory at each step of the computation
- accepting TM computations for an input w:
    - if a Turing machine M reaches q_{accept}, it halts and accepts w, even if M does not read all of w
    - if a Turing machine M reaches q_{reject}, it halts and rejects w, even if M does not read all of w
    - if M never reaches either state, the M does not accept w
    - the language of a TM is all strings accepted by the TM
    - a language is RE if some TM accepts ie. recognizes it
- TMs which halt (and accept or reject) for *all* inputs are called **algorithms** or **always-halting**:
    - the family of languages represented by TMs which halt for all inputs is called **recursive**
    - ie. algorithms never end up in an infinite loop
    - the family of languages accepted by TM algorithms are the **recursive** AKA **decidable** AKA **Turing-decidable** languages
        - the TM *decides* its language if it is an algorithm
    - general TMs that do not necessarily halt for all inputs are called **procedures**:
        - procedures are a superset of algorithms
        - the family of languages accepted by TM procedures are the **recursively enumerable** languages
        - the TM *recognizes* its language if it is a procedure
    - the **RE and not-recursive** languages are languages that are RE but not recursive
        - eg. the problem of whether a given computer program contains an infinite loop is a language in this family
    - **non-RE (NRE)** languages represents languages that *cannot* be represented by a TM ie. languages that are *not* computable
- the **Church-Turing thesis** states that the TM model and any equivalent models is essentially equivalent to our mental notion of a computer
    - ie. they represent everything computable

Ex. Example TM for $L = \{xx | x \in \Sigma^*\}$:

- let $\Sigma = \{0, 1\}, \Gamma = \{0, 1, \$, \#\}$
- the following is pseudocode for a 2-tape TM algorithm for L:

    1. write $ on the first position of the work tape
    2. until the input is blank, write one # onto the work tape for each two input symbols
        - if the length is odd, reject and halt ie. if we try to read two input symbols, but only one remains

3. until work tape symbol is \$, shift both heads left:
   - thus both heads move left $\frac{|w|}{2}$ positions
   - the input tape head will be at the beginning of the second half of w
4. move work tape head one position to the right
   - until input tape symbol is blank, copy second half of the input onto work tape, overwriting #'s
5. until work tape is \$, shift work tape head one position left and input head two positions left
   - this positions the work tape head at the beginning of the second half copy and input tape head at the beginning of w
6. until difference is found or we reach the blank at the end of the work tape, compare first half of w to copy of second half on work tape character by character
   - if there is a difference, reject
7. otherwise, accept

Computations vs. Language Recognition

- how can we move beyond language recognition as our model for computation problems?
  - TMs can be represented as several different types of models
- deciders and recognizers are typical TMs:
  - read-write input tape, black box of transitions, single final accepting and rejecting state
  - read-write work tapes (with number ≥ 0), with a fixed number for a given TM
  - all tapes heads can move left and write
  - these TMs are a special case of the transducer that could write 0 or 1 as output
- a **transducer** is similar to a typical TM, but produces an output tape:
  - has an *output*, write-only tape
    - head on output tape can only move left to right
  - has a final halting state that indicates the output is completed for a valid input
    - if output is infinite, the transducer may never halt
  - could say that it has a rejecting state for invalid inputs
- an **enumerator** generates all strings in some infinite language:
  - like the transducer but ignores input tape
    - has no halting states, simply runs in an infinite loop
  - has an *output*, write-only tape
    - head on output tape can only move left to right

  - prints out every string in its language in an infinite loop, separated by a blank:
    * does this somehow based on its work tapes
    * eg. print Fibonacci numbers, all $w \in \Sigma^*$, etc.
  - does not have to write words in any particular order, and can write some words more than once
- moving beyond artificial problems using **encodings**:
  - binary values $\{0, 1\}$
  - integer values can be encoded by a binary string over $\Sigma = \{0, 1\}$
    * have to follow some conventions, eg. most significant bit position, leading zeros
  - lists of integers eg. `n1#n2#...#nk`
  - set of integers as a list, with some conventions for order and duplicates
  - order pairs of integers eg. `#n1#n2#`
    * lists of ordered pairs eg. `#u1#v1##u2#v2##...##uk#vk#`
  - graphs as a set of vertices and edges:
    * can even represent vertices as the number of nodes
    * eg. `m###u1#v1##...`
  - labeled digraphs
  - labeled digraphs with sets of designated nodes
  - DFA, NFA, PDA, DPA, TM can *all* be syntactically represented by sets of strings over a suitable alphabet
  - the TM can also perform types of *static* analysis similar to syntactical checks on a model:
    * eg. does a given DFA accept the empty set?
    * eg. does a given DFA accept an infinite set?
    * eg. does a given DFA accept a given string?
      · could also be considered as dynamic analysis
  - but could a TM represent the *semantics* of each of these machines?
    * ie. performing *dynamic* analysis or an interpreter (run-time rather than compile-time)
    * eg. given an input representing a DFA and a string w, could a TM answer if A accepts w?
    * how about a PDA?
    * how about a TM?
    * yes, a TM can do all of these:
      · the **universal TM (UTM)** is one that accepts a representation of a TM and a string and answers whether the string is in the TM's language
      · generally, this universal TM functions by simulating the computation step-by-step, rather than predicting the output
      · published by Turing in 1936

Recursively Enumerable Languages

- the **Halting problem** is a famous problem that is RE but not recursive:
  - let the language $HP = \{(M, w) \mid M \text{ encodes a TM } M \text{ that halts}\}$
    - here, M represents both the TM and its encoding
  - ie. we can recognize HP with a procedure, but we *probably* cannot decide HP with an algorithm
    - we can try to determine what a given TM would do on a given input, however, generally the best we can do is to simulate its behavior with the UTM and see what happens
  - for the Halting problem, there is no way to tell if a program will go into an infinite loop, since we must simulate it for infinity in case it exits from a seemingly infinite loop
- before CS theory, mathematicians used to consider all computable problems to be recursive problems:
  - at this point, mathematical computations were either closed forms, infinite series and sequences, or recursions
  - Turing and others showed that there were problems that were computable, but *not* recursive
    - called these sets of problems recursively enumerable
- say we have a recursive ie. decidable language L_R over Σ:
  - let M be a decider for L_R ie. an algorithm
  - then, we can create an enumerator as follows:
    1. generate Σ^* by writing strings onto a work tape sequentially
    2. after each word, we pause the program and let the decider read the current word
    3. eventually the decider will halt and accept or reject
       - if it accepts, we copy the word to the special enumerator output tape
  - note that this enumerator is unique in that it outputs words in *predictable*, short-length order
    - we can also tell if a string belongs to a language based on if we *pass* that string
  - if we could *not* generate the strings of a language in a "nice" order, it *cannot* be recursive and is instead called recursively enumerable

Ex. Given two recursive languages L_1 and L_2, prove by construction that $L_1 \cap L_2$ is recursive.

- we want to construct a TM M for $L_1 \cap L_2$
  - let M_1 and M_2 be TM deciders for L_1 and L_2, respectively
- pseudocode approach:
  1. use UTM to simulate M_1 on the given input w:

  - because M_1 is an algorithm, it will eventually halt and accept or reject
  - if it halts and rejects, M halts and rejects
2. if it halts and accepts, use UTM to simulate M_2 on w:
  - if it halts and rejects, M halts and rejects
  - if it halts and accepts, M halts and accepts

Non-RE Languages

- not all infinities are equal:
  - for example, any set which can be put in a 1:1 correspondence with the integers, is **countably infinite**:
    - eg. even positive integers are countably infinite since they can be associated with each normal integer
    - eg. rational numbers are also countably infinite
    - eg. Σ^* is countably infinite, since we can order the strings over the alphabet in a certain ordering, regardless of the symbols in Σ
      - eg. $\varepsilon, 0, 1, 00, 01, ...$
  - sets that cannot be put in such a correspondence are **uncountably infinite**:
    - eg. the set of real numbers
    - eg. the set of all languages over a given alphabet
      - no set can be put in a 1:1 correspondence with its own power set

Ex. Prove the set of all languages over a given alphabet Σ ie. $\mathcal{P}(\Sigma^*)$is uncountably infinite. This proof is known as **Cantor's Diagonal**.

Suppose for contradiction that it were countably infinite.

Then, we could write out the words of Σ^* and associate each with one language over Σ ie. put them in a 1:1 correspondence with each other.

Then, we can write out a matrix where we have words over the columns and languages over the rows. The columns represent which languages each string belongs to. The rows represent the membership status for that language, ie. an entry containing `Y` at row i col j indicates that L_i contains word j. If the entry contains `N` instead, L_i does not contain word j.

Consider the main diagonal of this matrix. This diagonal is known as Cantor's Diagonal (CD) and it represents a language, since it says whether each word in Σ^* belongs to this language. Let this language be some L_i.

Note that this language is in the matrix twice, since it will have its own row in the

matrix in addition to Cantor's diagonal.

Where is the *complement* of this language in the matrix?

It cannot be the first row, since the entry will disagree with the first entry on the Cantor diagonal. Similar for the second entry, etc. Thus, *every* row k of the matrix disagrees with the hypothetical row of the complement of CD exactly at column entry k of that row, because of the diagonal. In other words, a contradiction is always found on the diagonal.

Thus the complement of CD is *not* a row on this matrix, which is a contradiction. Therefore, we cannot list all languages over Σ. This means the set of all languages is not countably infinite, so it is uncountably infinite.

This is a similar argument to why the set of real numbers is uncountably infinite.

- thus, we have proved that all languages over Σ are uncountably infinite
  - but all TMs are countably infinite, because TMs can be represented as finite strings
    - and thus all TM languages are countably infinite
  - thus there are "more" languages than TMs:
    - uncountable infinite is *uncountably* infinitely larger than countable infinity
      - infinitely many kinds of infinity
    - the probability of picking a language from the set of all languages that is RE, is zero!
    - many of these languages are "random" languages with no determinable pattern we could write down and understand
- RE languages:
  - $A_{TM} = \{(M, w) \mid \text{TM } M \text{ accepts } w\}$
  - halting problem language HP
  - $L_{self} = \{M \mid \text{TM } M \text{ accepts its own encoding }\}$
    - ie. $L_{self} = \{(M, M) \text{ is in } A_{TM}\}$
- however, there are "interesting" languages with finite representation we could *write* down and understand that are *not* RE:
  - $\overline{L_{self}} = \{M \mid \text{TM } M \text{ does not accept its own encoding }\}$
    - interesting and not RE!
- to see that $\overline{L_{self}}$ is not RE, construct another matrix of all TM encodings as columns against all TMs (in the same order):
  - ie. column w_i represents the encoding for M_i
    - alternatively, we can think of the columns as holding all TM encodings or all TM languages, since both of these are countably infinite
  - this time, both the set of columns and rows are countably infinite
  - consider the row corresponding to L_{self}:
    - note that once again, L_{self} is *also* reflected on the diagonal

    - the diagonal exactly is the definition of L_{self} ie. applying an encoding of a TM to that same TM
- next, if $\overline{L_{self}}$ were RE, it would appear as one of the rows:
    - the diagonal represents exactly L_{self}, but $\overline{L_{self}}$ must have the opposite entry on that diagonal
    - ie. the column for $\overline{L_{self}}$ on the diagonal will always contradict
- thus there cannot be a TM for $\overline{L_{self}}$

Complexity Theory

- FSLs consume constant $O(1)$ space and linear $O(n)$ time proportional to the length of the input
- how can we formalize the idea that some recursive problems are harder to decide than others?
  - we can perform **reductions** between computational problems
    - ie. mapping the output space of one problem to another problem
  - if there is a function that transforms instances of a problem p to instances of a problem p' which preserves the answer for p in the answer for $f(instance)$ in p', we say that p is reducable to p', and write $p \leq p'$
  - ie. p is no harder to solve than p' because we can use p' and $f()$ to solve p
  - we can think of the UTM as the most general problem to which every other RE problem can be reduced to
    - UTM is an interpreter as well as a problem that *includes* every other problem