Proofs

Detailed Proofs - Kleene's Theorem, Pumping Lemmas & Key Theorems

🧠 Proof Ideas Summary

Core Proof Techniques in Automata Theory:

1. **State Elimination (Kleene Part 1)** - Systematically remove states while preserving language, building up regex complexity
2. **Inductive Construction (Kleene Part 2)** - Build NFAs for complex expressions from simple base cases
3. **Pigeonhole Principle (Pumping Lemmas)** - Force repetition in finite structures to create "pumpable" segments
4. **Diagonalization (Halting Problem)** - Self-reference paradox to prove undecidability
5. **Reduction (Rice's Theorem)** - Transform one problem into another to transfer (un)decidability
6. **Constructive Proof (Union, Arden)** - Explicitly build the desired object (grammar, regex, automaton)
7. **Simulation (TM Equivalence)** - Show how one model can mimic another with encoding tricks

Key Insight Patterns:

- **Finite vs Infinite:** Use finiteness to force repetition (pumping)
- **Equivalence Proofs:** Build bijections between different representations
- **Undecidability:** Self-reference creates paradoxes
- **Closure:** Combine existing objects to create new ones with desired properties

Kleene's Theorem - Part 1

Statement: If a language L is accepted by a finite automaton, then L can be described by a regular expression.

💡 Proof Idea:

The key insight is to systematically eliminate states from the automaton while keeping track of all possible paths using regular expressions. Each eliminated state is "absorbed" into the transitions between remaining states.

Proof (State Elimination Method):

Step 1: Convert the given DFA/NFA to a Generalized NFA (GNFA) where:

- Transitions are labeled with regular expressions (not just symbols)
- Exactly one start state with no incoming edges
- Exactly one accept state with no outgoing edges
- At most one transition between any two states

Step 2: Systematically eliminate states (except start and accept states):

For each state q to be eliminated:

1. For every pair of states (qi, qj) where qi → q → qj:
2. Add/modify transition qi → qj with label: $R_1(R_2)^*R_3$
   - R_1 = regex from qi to q
   - R_2 = regex from q to q (self-loop)
   - R_3 = regex from q to qj

Step 3: Continue until only start and accept states remain.

Step 4: The regular expression on the final transition is the answer.

Example:

```
DFA: States {q₀, q₁}, q₀ start, q₁ accept
     q₀ --a--> q₁
     q₁ --b--> q₁

Eliminating: No intermediate states
Final RE: a·b*
```

Kleene's Theorem - Part 2

Statement: If a language L is described by a regular expression, then L is accepted by some finite automaton.

💡 Proof Idea:

Build NFAs recursively for each regex operation. The beauty is that NFAs naturally handle the non-deterministic choices needed for union and Kleene star, while ε-transitions elegantly connect subautomata.

Proof (Thompson's Construction):

Base Cases:

1. **∅:** NFA with start state, no accept state, no transitions
2. **ε:** NFA with start state = accept state, no transitions
3. **a (symbol):** NFA with start state, accept state, transition start --a--> accept

Inductive Cases:

Union ($R_1 \cup R_2$):

```
    ε      N₁       ε
q₀ ---> q₁ --- q₂ ---> q₃
 |              ε       ↑
 |       ε      N₂     ε |
 └----> q₄ --- q₅ -----┘
```

Concatenation ($R_1 \cdot R_2$):

```
q₀ ---> N₁ ---> q₁ ---> N₂ ---> q₂
                ε
```

*_Kleene Star (R_1)_:**

```
    ε
┌-------┐
↓       |
q₀ ---> q₁ ---> N₁ ---> q₂ ---> q₃
|        ε              ε       ↑
|                      ε        |
└------------------------------ ┘
```

Properties of Thompson's Construction:

- Exactly one start state, one accept state
- No transitions into start state
- No transitions out of accept state
- At most 2ε transitions from any state

Pumping Lemma for Regular Languages

Statement: If L is regular, then $\exists p > 0$ such that $\forall w \in L$ with $|w| \geq p$, w can be written as w = xyz where:

1. $|xy| \leq p$
2. $|y| > 0$
3. $\forall i \geq 0$, $xy^i z \in L$

💡 Proof Idea:

A DFA has finite states, so any long enough string must revisit some state, creating a loop. This loop can be repeated (pumped) or skipped without affecting acceptance.

Proof:

Step 1: Since L is regular, ∃ DFA M = (Q, Σ, δ, q_0, F) that accepts L.

Step 2: Let p = |Q| (number of states in M).

Step 3: Consider any w ∈ L with |w| ≥ p. Let w = $a_1a_2...a_n$ where n ≥ p.

Step 4: Consider the sequence of states: q_0, δ(q_0,a_1), δ(q_0,a_1a_2), ..., δ(q_0,$a_1...a_n$)

Step 5: This sequence has n+1 ≥ p+1 > |Q| states.

Step 6: By Pigeonhole Principle, some state repeats. Let q_i = q_j where 0 ≤ i < j ≤ p.

Step 7: Decompose w:

- x = $a_1...a_i$ (brings us to q_i)
- y = $a_{i+1}...a_j$ (loop from q_i back to q_i)
- z = $a_{j+1}...a_n$ (continues to accept state)

Step 8: Verify conditions:

1. |xy| = j ≤ p ✓
2. |y| = j - i > 0 ✓ (since i < j)
3. xy^i z ∈ L for all i ≥ 0 ✓ (can repeat or skip the loop)

Pumping Lemma for Context-Free Languages

Statement: If L is context-free, then ∃p > 0 such that ∀s ∈ L with |s| ≥ p, s can be written as s = uvwxy where:

1. |vwx| ≤ p
2. |vx| ≥ 1
3. ∀i ≥ 0, uv^i wx^i y ∈ L

💡 Proof Idea:

In a parse tree for a long string, some variable must appear twice on a path from root to leaf (pigeonhole principle). This creates a "nested" structure that can be pumped by repeating the inner pattern.

Proof:

Step 1: Since L is CFL, ∃ CFG G = (V, Σ, R, S) in CNF that generates L.

Step 2: Let p = 2^(|V|+1) where |V| is number of variables.

Step 3: Consider s ∈ L with |s| ≥ p. Any parse tree for s has height ≥ |V| + 1.

Step 4: Consider a longest path from root to leaf (length ≥ |V| + 1).

Step 5: This path has ≥ |V| + 2 nodes, so ≥ |V| + 1 variables.

Step 6: By Pigeonhole Principle, some variable A repeats on this path.

Step 7: Choose the lowest repetition of A. Let the tree structure be:

```
      S
     / \
    /   \
   u     y
        / \
       /   \
      A
     /|\
    / | \
   v  A  x
     /|\
    / | \
   w
```

Step 8: This gives decomposition s = uvwxy where:

- u, y: parts outside both A's
- v, x: parts between the two A's
- w: part inside inner A

Step 9: Verify conditions:

1. |vwx| ≤ p: The subtree rooted at outer A has yield vwx, and its height is ≤ |V| + 1, so |vwx| ≤ 2^(|V|+1) = p
2. |vx| ≥ 1: Since A → vAx is a production in CNF, at least one of v, x is non-empty
3. uv^i wx^i y ∈ L: Replace outer A with i copies of the pattern

---

Arden's Theorem

Statement: Let X be a language variable, and let A, B be regular languages with ε ∉ A. Then the equation X = AX ∪ B has a unique solution X = A*B.

💡 Proof Idea:

This theorem solves "language equations" - it's like solving X = aX + b in algebra, but for languages. The key insight is that A* captures all possible ways to repeatedly prepend strings from A.

Proof:

Part 1: X = A*B is a solution

We need to show A*B = A(A*B) ∪ B.

LHS: A*B = (ε ∪ A ∪ AA ∪ AAA ∪ ...)B = B ∪ AB ∪ AAB ∪ AAAB ∪ ...

RHS: A(A*B) ∪ B = A(B ∪ AB ∪ AAB ∪ ...) ∪ B = AB ∪ AAB ∪ AAAB ∪ ... ∪ B = B ∪ AB ∪ AAB ∪ AAAB ∪ ...

Therefore LHS = RHS, so A*B is indeed a solution.

Part 2: Uniqueness

Suppose Y is any solution to X = AX ∪ B. We'll prove Y = A*B.

Step 1: From Y = AY ∪ B, we get Y ⊇ B.

Step 2: From Y = AY ∪ B and Y ⊇ B, we get: Y = AY ∪ B ⊇ AB ∪ B So Y ⊇ AB ∪ B.

Step 3: By induction, $Y \supseteq A^nB \cup A^{n-1}B \cup \ldots \cup AB \cup B$ for all $n \geq 0$.

Step 4: Therefore Y ⊇ ∪_{n≥0} A^nB = A*B.

Step 5: Now we prove Y ⊆ A*B. Since Y = AY ∪ B, every string in Y is either:

- In B, hence in A*B (since ε ∈ A*)
- Of form aw where a ∈ A and w ∈ Y

Step 6: By strong induction on string length: If w ∈ Y, then w has form $a_1a_2 \ldots a_kv$ where $a_i \in A$, $v \in B$. Since ε ∉ A, this process terminates, giving w ∈ A*B.

Therefore Y = A*B.

Applications of Arden's Theorem:

Converting DFA to Regular Expression (Alternative to State Elimination):

For DFA with states $\{q_1, q_2, \ldots, q_n\}$, create equations:

- q_i = Σ_{$\delta(q_j,a)=q_i$} ($q_j \cdot a$) ∪ (ε if q_i is start state)

Example:

```
DFA: q₀ --a--> q₁ --b--> q₁, q₁ is accepting

Equations:
q₀ = ε
q₁ = q₀·a ∪ q₁·b

Substituting: q₁ = ε·a ∪ q₁·b = a ∪ q₁·b

Using Arden's theorem (A = b, B = a):
q₁ = b*a

Therefore L(DFA) = b*a
```

Theorem: Myhill-Nerode Theorem (Complete Proof)

Statement: L is regular iff the equivalence relation $\equiv_L$ has finitely many equivalence classes.

Definition: $x \equiv_L y$ iff $\forall z \in \Sigma^*$, $(xz \in L \iff yz \in L)$

💡 Proof Idea:

The equivalence classes of $\equiv_L$ naturally correspond to DFA states. If there are finitely many classes, we can build a DFA with one state per class. Conversely, if L is regular, the DFA states bound the number of distinguishable string prefixes.

Proof:

(⇒) If L is regular, then $\equiv_L$ has finitely many classes:

Step 1: Let $M = (Q, \Sigma, \delta, q_0, F)$ be DFA accepting L.

Step 2: Define relation ~: $x \sim y$ iff $\delta^*(q_0, x) = \delta^*(q_0, y)$

Step 3: Claim: $x \sim y \implies x \equiv_L y$

- Proof: If $\delta^*(q_0, x) = \delta^*(q_0, y) = q$, then for any z: $\delta^*(q_0, xz) = \delta^*(q, z) = \delta^*(q_0, yz)$
- So $xz \in L \iff yz \in L$

Step 4: Since ~ has |Q| classes and ~ refines $\equiv_L$, $\equiv_L$ has $\leq |Q|$ classes.

(⇐) If $\equiv_L$ has finitely many classes, then L is regular:

Step 1: Let $\{C_1, C_2, ..., C_n\}$ be the equivalence classes of $\equiv_L$.

Step 2: Construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ where:

- $Q = \{C_1, C_2, \ldots, C_n\}$
- q_0 = class containing ε
- $F = \{C_i :$ some (hence all) strings in C_i are in $L\}$
- $\delta(C_i, a)$ = class containing wa where $w \in C_i$

Step 3: δ is well-defined: If $w_1, w_2 \in C_i$, then $w_1a \equiv\_L\ w_2a$

Step 4: M accepts L: $x \in L$ iff $\delta^*(q_0, x) \in F$ by construction.

Theorem: CFL Closure Under Union (Constructive Proof)

Statement: If L_1 and L_2 are context-free, then $L_1 \cup L_2$ is context-free.

💡 Proof Idea:

Create a new grammar that can generate strings from either original grammar by adding a new start symbol with productions that "choose" between the two languages.

Proof:

Step 1: Let $G_1 = (V_1, \Sigma_1, R_1, S_1)$ generate L_1 and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ generate L_2.

Step 2: WLOG, assume $V_1 \cap V_2 = \emptyset$ (rename variables if necessary).

Step 3: Construct $G = (V, \Sigma, R, S)$ where:

- $V = V_1 \cup V_2 \cup \{S\}$ (S is new start symbol)
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $R = R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$

Step 4: Prove $L(G) = L_1 \cup L_2$:

⊆: If $w \in L(G)$, then $S \Rightarrow^* w$. The derivation starts $S \to S_1$ or $S \to S_2$.

- If $S \to S_1$, then $S_1 \Rightarrow^* w$ using only R_1, so $w \in L_1$
- If $S \to S_2$, then $S_2 \Rightarrow^* w$ using only R_2, so $w \in L_2$
- Therefore $w \in L_1 \cup L_2$

⊇: If $w \in L_1 \cup L_2$:

- If $w \in L_1$, then $S_1 \Rightarrow^* w$, so $S \to S_1 \Rightarrow^* w$
- If $w \in L_2$, then $S_2 \Rightarrow^* w$, so $S \to S_2 \Rightarrow^* w$
- Therefore $w \in L(G)$

Theorem: Equivalence of Multi-tape and Single-tape TMs

Statement: Every k-tape Turing Machine can be simulated by a single-tape Turing Machine.

💡 Proof Idea:

Encode multiple tapes on a single tape by interleaving symbols and using markers to track head positions. Each multi-tape step requires scanning the entire single tape twice - once to read current symbols, once to update them.

Proof (Construction):

Step 1: Encoding Multiple Tapes on Single Tape

- Use tape alphabet $\Gamma' = (\Gamma \cup \{\#\})^k \times \{0,1\}^k$
- Each cell stores k symbols and k bits (indicating head positions)
- Separate tape contents with # symbols

Step 2: Initial Configuration

```
Original k tapes:    |a|b|c| □ □ ...
                     |d|e| □ □ □ ...

Single tape:    #(a,d,1,1)#(b,e,0,0)#(c,□,0,0)#(□,□,0,0)#...
```

Step 3: Simulation of One Step For each transition $\delta(q, a_1,\ldots,a_k) = (q', b_1,\ldots,b_k, D_1,\ldots,D_k)$:

1. **Scan Phase:** Scan entire tape to find head positions and read symbols
2. **Update Phase:** Scan again to:
   - Write new symbols $b_1,\ldots,b_k$ at head positions
   - Update head position markers according to $D_1,\ldots,D_k$
   - If head moves right past end, extend tape

Step 4: Time Complexity Analysis

- If k-tape TM runs in time $T(n)$, single-tape simulation takes $O(T(n)^2)$
- Each step requires 2 full scans of length $O(T(n))$

Step 5: Correctness

- Initial configuration correctly represents k blank tapes
- Each simulation step correctly implements one k-tape step
- Acceptance condition preserved

Theorem: Undecidability of the Halting Problem

Statement: The language H = {⟨M,w⟩ | M is a TM that halts on input w} is undecidable.

💡 Proof Idea:

Create a paradox using self-reference. If we could decide halting, we could build a machine that does the opposite of what our halting decider predicts when run on itself - a logical contradiction.

Proof (Diagonalization):

Assume for contradiction that H is decidable. Then ∃ TM D that decides H:

- D(⟨M,w⟩) = accept if M halts on w
- D(⟨M,w⟩) = reject if M doesn't halt on w

Step 1: Construct TM H' that behaves as follows on input ⟨M⟩:

```
H'(⟨M⟩):
1. Run D(⟨M,⟨M⟩⟩)
2. If D accepts: loop forever
3. If D rejects: halt and accept
```

Step 2: What happens when we run H' on its own encoding ⟨H'⟩?

Case 1: Suppose H' halts on ⟨H'⟩

- Then D(⟨H',⟨H'⟩⟩) should accept (since H' halts on ⟨H'⟩)
- But then H' loops forever by its definition
- Contradiction!

Case 2: Suppose H' doesn't halt on ⟨H'⟩

- Then D(⟨H',⟨H'⟩⟩) should reject (since H' doesn't halt on ⟨H'⟩)
- But then H' halts and accepts by its definition
- Contradiction!

Step 3: Since both cases lead to contradiction, our assumption is false. Therefore, H is undecidable.

Theorem: Rice's Theorem

Statement: Let P be any non-trivial property of recursively enumerable languages. Then {⟨M⟩ | L(M) has property P} is undecidable.

Definition: Property P is non-trivial if:

- Some r.e. language has property P
- Some r.e. language doesn't have property P

💡 Proof Idea:

Use the Halting Problem as a "universal reducer." For any semantic property P, we can construct machines whose language depends on whether a given machine halts - effectively reducing the Halting Problem to property P.

Proof:

Step 1: WLOG, assume ∅ doesn't have property P (if it does, consider $\bar{P}$).

Step 2: Since P is non-trivial, ∃ language L_0 that has property P. Let M_0 be a TM with $L(M_0) = L_0$.

Step 3: Assume for contradiction that S_P = {⟨M⟩ | L(M) has property P} is decidable.

Step 4: We'll use this to decide the Halting Problem, contradicting its undecidability.

Step 5: Construction: For any ⟨M,w⟩, construct TM M_{M,w} as follows:

```
M_{M,w}(x):
1. Simulate M on input w for |x| steps
2. If M halts within |x| steps: simulate M₀ on input x
3. If M doesn't halt within |x| steps: reject
```

Step 6: Key Observation:

- If M halts on w: M_{M,w} eventually simulates M_0 on all inputs, so L(M_{M,w}) = L_0
- If M doesn't halt on w: M_{M,w} rejects all inputs, so L(M_{M,w}) = ∅

Step 7: Reduction: Since L_0 has property P and ∅ doesn't:

- ⟨M,w⟩ ∈ H ⟺ L(M_{M,w}) has property P ⟺ ⟨M_{M,w}⟩ ∈ S_P

Step 8: If S_P were decidable, we could decide H:

```
Algorithm for H:
Input: ⟨M,w⟩
1. Construct M_{M,w}
2. Test if ⟨M_{M,w}⟩ ∈ S_P
3. Return same answer
```

Step 9: This contradicts undecidability of H, so S_P is undecidable.