DFA: is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Q is a finite set of states. Σ is a finite alphabet. $\delta : Q \times \Sigma \to Q$ is the transition function. $q_0 \in Q$ is the start state. $F \subseteq Q$ is the set of accept states.

NFA: is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. $\delta : Q \times \Sigma_\epsilon \to P(Q)$ is the transition function

PDA: is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Γ is the stack alphabet, $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$.

NFA to DFA: The states for the DFA will be $P(Q)$. The start state becomes the start state from the NFA plus everything that can be reached from empty transitions. The accept states become everything that contains an accept state from the NFA. A transition example would be if NFA has $2 \to_a 3$ and $2 \to_a 2$ then the DFA would have $\{2\} \to_a \{2, 3\}$ so long as neither 2 or 3 have any empty transitions. If DFA is missing some transition because NFA does not have them they go to the state $\emptyset$.

RegEx to FA: a becomes $\to q_0 \to_a q_1$. $A \cup B$ add a new start state and make empty transition from that to the old start states. $A \circ B$ make the accept states of A regular states and make empty transitions from those to the start state of B. A^* add a new start state which is also a final state, then for each final state make an empty transition from the old start state.

FA to RegEx: Transform to GNFA. 1. The start state has transition arrows going to every other state but no arrows coming in from any other state. 2. There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is no the same as the start state. 3. Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.

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

Then in the new machine, the arrow from q_i to q_j gets the label $(R_1)(R_2)^*(R_3) \cup (R_4)$. We make this change for each arrow going from any state q_i to any state q_j, including the case where $q_i = q_j$. The new machine recognizes the original language.

Pumping lemma for reg lang: If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions: 1. for each $i \geq 0, xy^iz \in A$. 2. $|y| > 0$. 3. $|xy| \leq p$.

CFG: A context-free grammar is a 4-tuple (V, Σ, R, S), where V is a finite set called the variables, Σ is a finite set, disjoint from V, called the terminals, R is a finite set of rules, with each rule being a variable and a string of variables and terminals and $S \in V$ is the start variable.

Chomsky normal form: A context-free grammar is in Chomsky normal form if every rule is of the form $A \to BC$ or $A \to a$. Where a is any terminal and A, B and C are any variables–except that B and C may not be the start variable. In addition, we permit that rule $S \to \epsilon$, where S is the start variable.

CFG to CNF: We add a new start variable S_0 and the rule $S_0 \to S$, where S was the original start variable.

We take care of all ϵ-rules. We remove an ϵ-rule $A \to \epsilon$, where A is not the start variable. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. In other words, if $R \to uAv$ is a rule in which u and v are strings of variables and terminals, we add rule $R \to uv$. We do so for each occurrence of an A, so the rule $R \to uAvAw$ causes us to add $R \to uvAw$, $R \to uAvw$, and $R \to uvw$. If we have the rule $R \to A$, we add $R \to \epsilon$ unless we had previously removed the rule $R \to \epsilon$. We repeat these steps until we eliminate all ϵ-rules not involving the start variable.

We handle all unit rules. We remove a unit rule $A \to B$. Then, whenever a rule $B \to u$ appears, we add the rule $A \to u$ unless this was a unit rule previously removed. As before, u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules.

We convert all remaining rules into the proper form. We replace each rule $A \to u_1u_2 \ldots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol, with the rules $A \to u_1A_1$, $A_1 \to u_2A_2$, $A_2 \to u_3A_3$, ..., and $A_{k-2} \to u_{k-1}u_k$. The A_i's are new variables. We replace any terminal u_i in the preceding rule(s) with the new variable U_i and add the rules $U_i \to u_i$.

CFG to PDA: Place the marker symbol \$ and the start variable on the stack. Repeat the following steps forever. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the chosen production. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism. If the

top of stack is the symbol $, enter the accept state.

PDA to CFG: First, we simplify our task by modifying P slightly to give it the following three features. Tt has a single accept state q_{accept}. Tt empties its stack before accepting. Each transition either pushes a symbol onto the stack or pops one of the stack, but it does not do both at the same time.

Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$ and construct G. The variables of G are $\{A_{pq} | p, q \in Q\}$. The start variable is $A_{q_0, q_{accept}}$. Now we describe G's rules in three parts.

1. For each $p, q, r, s \in Q, u \; in \Gamma$ and $a, b \in \Sigma_\epsilon$. If $(r, u) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, u)$. Put the rule $A_{pq} \to aA_{rs}b$ in G.

2. For each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr}A_{rq}$ in G.

3. Finally, for each $p \in Q$, put the rule $A_{pp} \to \epsilon$ in G.

pumping lemma for cfl: If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions. 1. for each $i \geq 0, uv^ixy^iz \in A$. 2. $|vy| > 0$. 3. $|vxy| \leq p$.

SOS: To summarize, a structural operational semantics consists of the following.

- A definition of the abstract syntax of the languages with
  - a listing of the syntactic categories used in the language
  - for each syntatic cotegory, a set of formation rules
- Definitions of additional sets and auxiliary functions, if needed
- Definitions of all transition systems. For each transition system $(\Gamma, \to, T)$ there must be
  - a definition of the set of configurations Γ and the set of terminal configurations T
  - a set of syntax-directed transition rules defining the transition relation $\to$

Transition induction: We use this technique, if we want to prove a claim of the form 'For all transitions P holds', where P is some property. We then proceed as follows. For each axiom show that P holds. For each transition rule show that P is preserved by the rule, that is, show that if we assume that P holds for all Premises in the rule, then P will also hold for its conclusion. This is induction on k, where k is the height of our derivation tree.

Induction on the length of transition sequences: We use this technique for a small-step semantics, if we want to prove a claim of the form 'For all transition sequences P holds'. This, too, is an induction in the normal sense, in that we show the claim 'For all k it is the case that P holds for every transition sequence of length k' by induction on k.