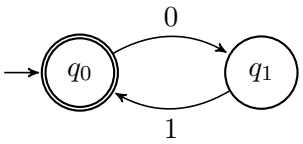
For each of these statements, say if it is true or false. Give a proof or provide a counterexample for your answer.

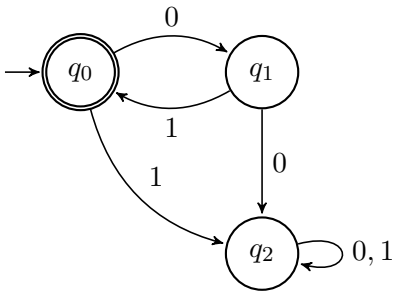
(1) There is a 2-state NFA for the language $(01)^*$.

True. Here it is:



(2) There is a 2-state DFA for the language $(01)^*$.

False. This is a 3-state DFA for $(01)^*$, and all pairs of states are distinguishable (q_0, q_1) and (q_0, q_2) by ε, (q_1, q_2) by 1. So the minimal DFA for $(01)^*$ has 3 states.



(3) If L is regular over $\Sigma = \{0, 1\}$, then L' is also regular, where

$$L' = \{x \mid x \in L \text{ and } x \text{ starts and ends with the same symbol}\}.$$

True. We can write L' as the intersection of L and the regular language $0(0+1)^*0+1(0+1)^*1$. Since regular languages are closed under intersection, L' must be regular.

(4) The language $L = \{wxw^Rx^R \mid x, w \in \Sigma^*\}$ is context-free over alphabet $\Sigma = \{a, b\}$.

False. We prove it using the pumpling lemma for context-free languages. Let n be the pumping length and consider the string $0^n1^n0^n1^n$, which is in L. Notice that every string in L has the same number of 0s in the first and second half, and the same for 1s. Consider any partition of this string as $uvwxy$, where $|vwx| < n$ and vx is not empty. If vx intersects the initial block 0^n, after pumping up we obtain a string that has more 0s in the first half. Similarly, if vx intersects the final block 1^n, after pumping up we obtain a string that has more 1s in the last half.

If neither of these cases happens, then v and x must both come from the middle part 1^n0^n. If vx contains more 0s than 1s, after pumping down we get a string with a shortage of 0s in the second half. If it contains more 1s than 0s, after pumping down we get a string with a shortage of 1s in the first half. If it has the same number of 0s and 1s, after pumping down we get $0^n1^m0^m1^n$, where $m < n$, which is not in L.

(5) If L_1 and L_2 are regular languages, then the following language is context-free:

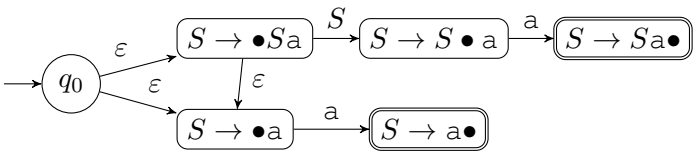
$$L = \{xy \mid x \in L_1,\ y \in L_2, \text{ and } |x| = |y|.\}$$

True. Let D_1 and D_2 be DFAs for L_1 and L_2, respectively. We design a PDA for L. Intuitively, the PDA will first simulate D_1, then D_2, using the stack to make sure that both of them are simulated for the same number of steps. The PDA will nondeterministically guess the middle of the input.

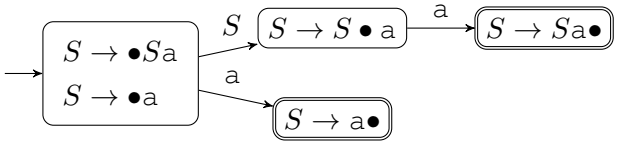
More precisely, here is how the PDA works: First, push the bottom marker \$ on the stack. Start reading the input and simulate the DFA D_1. Every time a symbol is read, push an a onto the stack. Every time D_1 reaches an accept state, allow a non-deterministic ε-transition to the start state of D_2. As you continue reading the input, simulate the transitions of D_2, popping an a from the stack every time. If the bottom of the stack \$ is reached in an accept state of D_2, then accept, otherwise reject.

(6) The grammar $S \to S\mathtt{a} \mid \mathtt{a}$ is $LR(0)$. Explain your answer with a DFA.

True. Here is the corresponding NFA:



And the DFA:



All other transitions go to a dead state that is not shown above for clarity. There are no shift-reduce or reduce-reduce conflicts.

(7) The following language is decidable:

$$L = \{\langle R\rangle \mid \text{Regular expression } R \text{ generates at least one string of even length.}\}$$

True. Let $\Sigma = \{a_1, \ldots, a_k\}$ be the alphabet for R. The following Turing Machine decides:

L: On input a regular expression $\langle R\rangle$
construct the regular expression $R' = ((a_1 + \cdots + a_k)(a_1 + \cdots + a_k))^*$
convert $R \cap R'$ into a DFA
minimize this DFA to get a DFA D_1
construct a DFA that accepts no strings
minimize this DFA to get a DFA D_2
accept if D_1 and D_2 are different

Since R' represents strings of even length, $R \cap R'$ generates at least one string if and only if R contains at least one string of even length.

(8) The following language is decidable:

$$L = \{\langle G_1, G_2\rangle \mid \text{CFG } G_2 \text{ generates some string that CFG } G_1 \text{ does not generate.}\}$$

False. We reduce from ALL_{CFG}. Assume L is decidable and let M be a decider for it. We use M to construct a decider T for ALL_{CFG} as follows:

T: On input G over alphabet $\Sigma = \{a_1, \ldots, a_k\}$
construct the following CFG G': $S' \to a_1 S' \mid \cdots \mid a_k S' \mid \varepsilon$.
Run M on input $\langle G, G'\rangle$
If M rejects $\langle G, G'\rangle$, then accept; otherwise reject

By construction, G' generates all possible strings over Σ. So if G generates all strings, G and G' will generate the same strings, so M will reject and T will accept. Otherwise, G will fail to generate some string, so G' generates more strings than G, so M will accept and T will reject. It follows that T decides ALL_{CFG}, contradicting the undecidability of ALL_{CFG}. Therefore L is undecidable.

(9) The following language is decidable:

$$L = \{\langle M, k\rangle \mid \text{TM } M \text{ accepts at most } k \text{ inputs.}\}$$

False. We reduce from A_{TM}. Assume L is decidable and let S be a decider for it. We use M to construct a decider T for A_{TM} as follows:

T: On input $\langle M, w\rangle$, where M is a Turing machine and w is a string
Construct the following Turing machine M':
M': On input x
Simulate M on input w
If M accepts w, accept; otherwise reject
Run S on $\langle M', 0\rangle$
If S accepts $\langle M', 0\rangle$, reject; otherwise accept

If M accepts w, then M' accepts all x, so S rejects $\langle M', 0\rangle$ and T accepts. Otherwise, M does not accept w, then M' does not accept any x, so S accepts $\langle M', 0\rangle$ and T rejects. Therefore T decides A_{TM}, which is impossible.

(10) The following language is NP-complete:

$$L = \{\langle \phi, \psi\rangle \mid \text{Boolean formulas } \phi \text{ and } \psi \text{ share a common satisfying assignment.}\}$$

True. First, L is in NP. The following is a description of a polynomial-time verifier TM for L: On input $\langle \phi, \psi, a\rangle$, where ϕ and ψ are formulas, and a is a candidate assignment, check that a satisfies both ϕ and ψ. Because checking if a formula satisfies an assignment can be done in polynomial time, this is a polynomial-time verifier for L, and so L is in NP.

We show that L is NP-hard. We show that SAT reduces to L. Consider the following transformation that takes an instance $\langle \phi\rangle$ of **SAT** and produces an instance $\langle \phi', \psi\rangle$ of L: Set $\phi' = \phi$ and construct ψ to be a formula over the same variables as ϕ that accepts all strings. For example, you can set $\psi = \text{true}$ (or if it makes you more comfortable, set $\psi = (x_1 \vee \overline{x}_1) \wedge \cdots \wedge (x_n \vee \overline{x}_n)$).

This transformation runs in polynomial-time, as the formula ψ can be constructed very quickly (in linear time). Moreover, if ϕ is satisfiable, then ϕ and ψ share a satisfying assignment, namely any satisfying assignment of ϕ. If ϕ is not satisfiable, then ϕ and ψ cannot share any satisfying assignments.

We showed that SAT reduces to L in polynomial time, therefore L is NP-complete.