

- 2.3** a) Variables: R, X, S, T
Terminals: a, b
Start Variable: R (Note: By convention it should be S , but here R makes more sense because starting from S we won't even be able to use the first rule)
- b) Example: ab, ba, aab
- c) Example: ϵ, a, b
- d) False. (can't go from T to aba in ONE step)
- e) True.
- f) False. (has to go one step)
- g) True.
- h) True.
- i) False. (one X can only generate one letter)
- j) True.
- k) True.
- l) False.
- m) All strings in the form $(a \cup b)^*$ that are not palindromes. (Note that S is used to ensure that there's a mismatch)

2.7 Design of PDA's Although you can design the CFG for this language first and then transform it into a PDA, you can also design a PDA directly. The design technique is fairly different. Remember that PDA is finite automaton with a stack which can remember information. Generally you're storing some information on the stack which you can later use to match something else. It's again useful to do a case analysis, according to the possible states and stack configurations you might encounter.

a) Main idea is to match two a 's with a single b , so we always duplicate b before we put it on the stack to match with future a 's. Case analysis is the following:

- $\epsilon \in L$

- current input read is b

if the stack is empty or there're some b 's on the stack, push two b 's onto the stack

if there're $\geq 2a$'s on the stack, pop two a 's out for this b

if there is only one a on the stack, push one b onto the stack (we need one more a to match this b)

- current input read is a

if the stack is empty or there're some a 's on the stack, push the a onto the stack

if there're some b 's on the stack, pop one b out for this a

And we accept when the stack is empty. Please see corresponding figure for the PDA diagram.

d) For this particular language, the design of PDA is actually a little easier than the design of CFG. Main idea is to non-deterministically guess what is to be matched with what. Ignore the input until we get to the part that is to be matched. Now push this part on the stack. Ignore the part in the middle. When we get to the matching part, we compare it with what we have on the stack. If we have a match, we basically ignore the rest of the input, checking to make sure that it is in the right format. Please see corresponding figure for the PDA diagram.

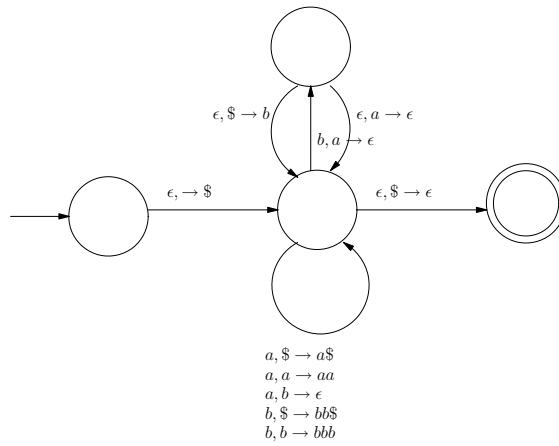


Figure 1: 2.7 (a)

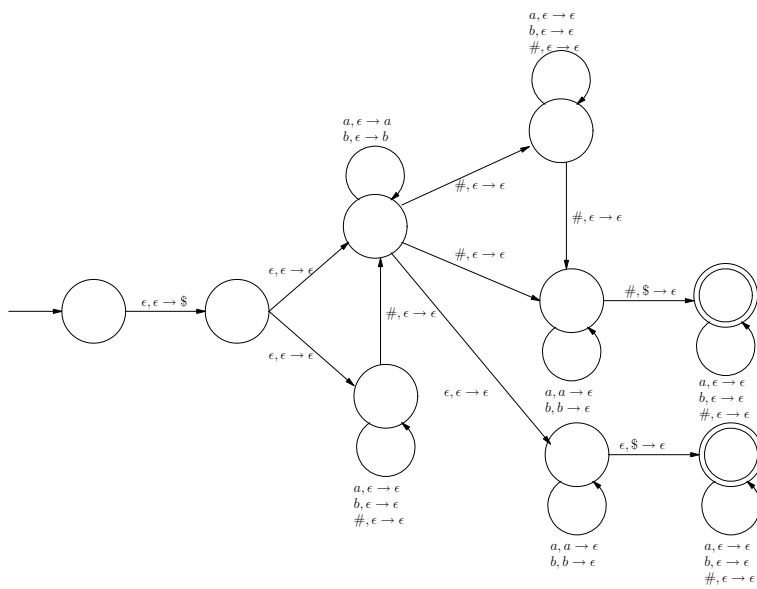


Figure 2: 2.7 (d)

2.13 a) Strings that contain exactly two #’s and any number of 0’s in any order, or strings that contain exactly one # such that the number of 0’s to the right of # is twice the number of 0’s to the left.

b) **Proof:** $L(G)$ is the union of the language generated by T concatenated with itself and the language generated by U .

T generates $0^* \# 0^*$, therefore TT generates $0^* \# 0^* \# 0^*$, which is regular.

U generates $\{0^n \# 0^{2n} : n \geq 0\}$, which is nonregular.

To show that $L(G)$ is not regular, we can first get rid of the regular part by intersection $L(G)$ with the regular expression $0^* \# 0^*$. Since regular languages are closed under intersection, if $L(G)$ is regular, then so must be the language we get after the intersection, which is the language generated by U . A simple pumping argument will now suffice to show that the language generated by U is not regular.

Assume the pumping length is p . Choose your string as $0^p \# 0^{2p}$. Pumping down once or pumping up will immediately destroy the dependency between the two groups of 0’s. This means that for any partition, this chosen string cannot be pumped. Therefore by the pumping lemma, we know that the language generated by U cannot be regular. Hence $L(G)$ is not regular either.

2.18 b) **Proof:** Assume this language L is context free. Let the pumping length be p . We show that $w = 0^p \# 0^{2p} \# 0^{3p}$ cannot be pumped. Consider any partition $w = uvxyz$. Recall the requirements that $|vy| > 0$ and $|vxy| \leq p$.

1) If v or y contains #, uv^2xy^2z will contain more than two #’s. Therefore $uv^2xy^2z \notin L$, and L fails to have the pumping property in this case.

2) If neither v nor y contains #, consider the three segments separated by #’s: $0^p, 0^{2p}, 0^{3p}$. At least one of the segment is not contained within either v or y . Then $uv^2xy^2z \notin B$ because the 1:2:3 length ratio cannot be maintained. So L fails to have the pumping property in this case also.

Therefore B is not context free.

c) **Proof:** Assume this language L is context free. Let the pumping length be p . We show that $w = a^p b^p \# a^p b^p$ cannot be pumped. Consider any partition $w = uvxyz$. Recall the requirements that $|vy| > 0$ and $|vxy| \leq p$.

1) If v or y contains #, uv^2xy^2z will contain more than one #’s. Therefore $uv^2xy^2z \notin L$, and L fails to have the pumping property in this case.

2) If both v and y are nonempty and occur on the left hand side of #, uv^2xy^2z cannot be in L because it’s longer the left hand side of #. Similarly if both strings occur on the right hand side of #, uv^0xy^0z cannot be in L because again it’s longer the left hand side of #.

3) If only one of v and y is nonempty, similar argument as in 2) works if we treat them as if both occur on the same side of #.

4) If both v and y are nonempty but occur on different sides of #, consider the length constraint $|vxy| \leq p$ and it has to be the case that v consists of only b ’s and y consists of only a ’s. Then $uv^2xy^2z \notin L$ because it contains more b ’s on the left hand side of #.

Therefore B fails to have the pumping property for any partition of $w \in B$ and is not context free.

2.26 Intuition: We know that PDA’s cannot check for equality of strings, i.e. they cannot test whether $x = y$. Why then is it possible for them to test for inequality? This is because to make sure that two strings are different, it suffices to make sure that they differ at **one** point. On the other hand, to check that two strings are equal, we have to ensure that they are the same everywhere.

Construction of PDA for $C = \{x \# y : x, y \in \{a, b\}^*\}$:

It operates by guessing a specific position on which the strings x and y differ. It reads the input at the same time as it pushes some symbols, say 1’s onto the stack, which are used to record the position. At some point it non-deterministically guesses that the current position is

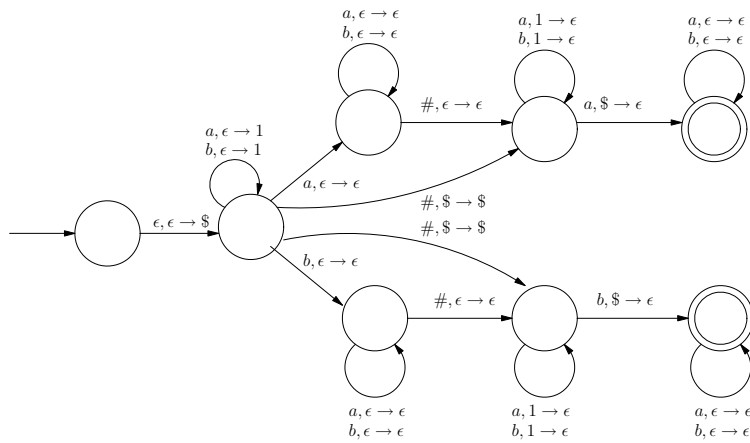


Figure 3: 2.26

the position in x where x and y differ and it records the symbol it is reading there in its finite memory and skips to the $\#$. Then it pops the stack while reading input symbols from the input until the stack is empty and checks that the symbol it is now reading is different from the symbol it had recorded. If so, it accepts. If something goes wrong, e.g. popping when the stack is empty, or getting to the end of the input prematurely, it rejects on that branch of nondeterminism. Please refer to corresponding figure for the *PDA* diagram.