The University of Melbourne
School of Computing and Information Systems
COMP30026 Models of Computation

# Assignment 2, 2019, feedback to Challenges 1–3

For feedback to Challenges 4–6, go to Grok.

## Challenge 1

The main problems were grammars that did not generate the right language.

- Some were missing strings. For example, the grammar:

$$\begin{array}{lcl} S & \to & \mathsf{a}\ A\ \mathsf{a} \mid \mathsf{b}\ B\ \mathsf{b} \\ A & \to & C\ A\ C \\ B & \to & C\ B\ C \\ C & \to & \mathsf{a} \mid \mathsf{b} \end{array}$$

  does not generate all the required strings, like $\mathsf{a}$ or $\mathsf{bbb}$.

- Others were too generous. For example, the grammar:

$$\begin{array}{lcl} S & \to & \mathsf{a}\ X\ \mathsf{b}\ X \mid X\ \mathsf{b}\ X\ \mathsf{a} \\ X & \to & \mathsf{a}\ X \mid \mathsf{a} \mid \epsilon \end{array}$$

  generates strings like $\mathsf{aba}$: $S \Rightarrow \mathsf{a}X\mathsf{b}X \Rightarrow \mathsf{aba}$.

- Some students still assume that different occurrences of a grammar variable on the RHS of a production must be rewritten to the same string. That, of course, is not the case.

For Challenge 1a, the easiest way to solve it is to distinguish, at the top level of the grammar, the two different cases: Either $\mathsf{a}$ is in the middle, or $\mathsf{b}$ is in the middle:

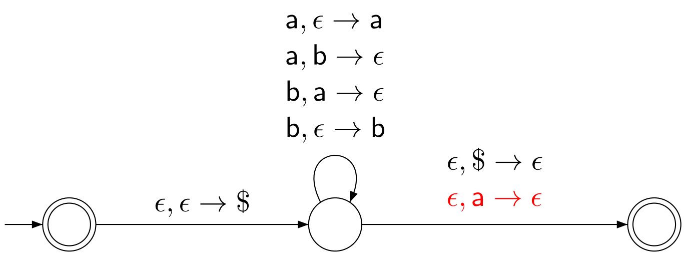
$$\begin{array}{lcl} S & \to & \mathsf{a} \mid \mathsf{b} \mid \mathsf{a}\ A\ \mathsf{a} \mid \mathsf{b}\ B\ \mathsf{b} \\ A & \to & \mathsf{a} \mid C\ A\ C \\ B & \to & \mathsf{b} \mid C\ B\ C \\ C & \to & \mathsf{a} \mid \mathsf{b} \end{array}$$

Challenge 1b is quite similar to tutorial exercise 80, once we realise that $i \neq j$ is equivalent to $i > j \vee j > i$:

$$\begin{array}{lcl} S & \to & T \mid U \\ T & \to & A\ B \\ U & \to & B\ A \\ A & \to & \mathsf{a} \mid \mathsf{a}\ A \\ B & \to & \mathsf{b} \mid \mathsf{a}\ B\ \mathsf{a} \end{array}$$

## Challenge 2a

Many students did not understand the way a push-down automaton works, and/or they misinterpret '$\epsilon$'. This leads to proposed solutions like this one:


$\mathsf{a}, \epsilon \to \mathsf{a}$
$\mathsf{a}, \mathsf{b} \to \epsilon$
$\mathsf{b}, \mathsf{a} \to \epsilon$
$\mathsf{b}, \epsilon \to \mathsf{b}$
$\epsilon, \epsilon \to \$$
$\epsilon, \$ \to \epsilon$
$\epsilon, \mathsf{a} \to \epsilon$

This PDA will accept bba, which should be rejected. Here is what the PDA will do, step by step:

a. Consume nothing from the input, move from state 0 to 1, push a \$ onto the stack;

b. Consume a b from the input, stay in state 1, push a b to the stack;

c. Consume a b from the input, stay in state 1, push a b to the stack;

d. Consume an a from the input, stay in state 1, push an a to the stack;

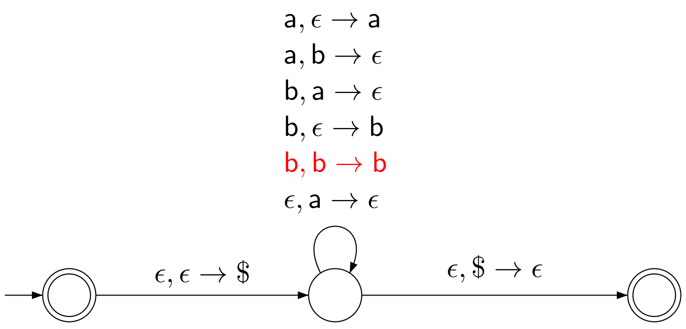
e. Consume nothing from the input, transit from state 1 to 2, pop an a from the stack;

f. Since all input has been consumed and the PDA is now in an accept state, it will accept.

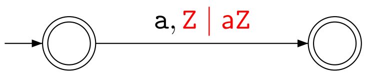
Some students reacted to the marking of this type of machine by saying, "But that's not how my PDA is meant to operate. In step d it was obviously meant to pop a b, rather than push an a, and so it would end up rejecting." It is really important to understand how the nondeterminism works. The given PDA will accept bba *because it can*. It is biased towards acceptance.

In a rule like $a, \epsilon \rightarrow a$, $\epsilon$ does not stand for "the bottom of the stack". The rule says "consume, if you want, an a from input, which replacing *nothing* on the stack by an a; or, in other words, push a. Nor is $\epsilon$ used as a marker of "end of input".

Another example of a misconception: The red transition allows the following PDA to consume multiple bs from the input for free, as long as the top of the stack is a b:

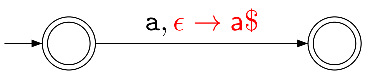


Some were using unconventional representations without a clear definition or explanation. For example, we cannot possibly know how to read this:



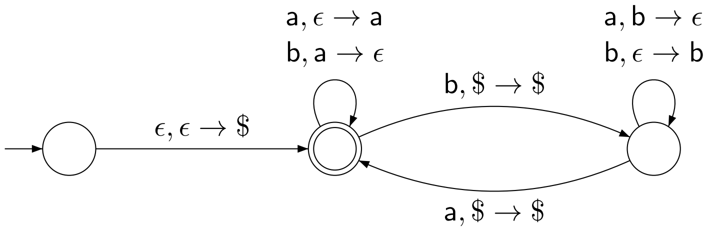
Precision and clarity in the use of formal notation is not optional; you cannot just invent your own notation and hope the reader will guess the intended meaning.

Many students resorted to "transitions" that would push or pop several symbols in one go, like this:

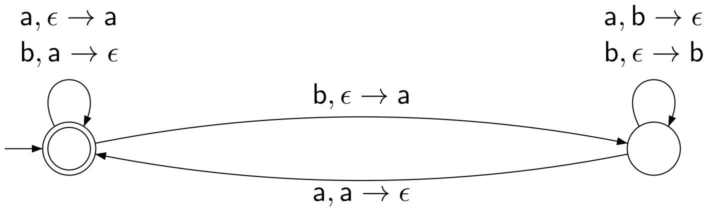


The way we have defined a PDA, it cannot do that.

Here is a correct 3-state PDA which recognises $C$:



This was the solution we had in mind. I was amazed to see that a 2-state PDA can do the same job. A couple of students found this solution, avoiding the use of $:



Brilliant.

## Challenge 2b

The most common comments from the markers were:

- Missing a as a base case.
- Informal proof. For example, clear statement of the assumption (that is, the immediate substructure is “a-leaning”) in each induction step.
- Inexplicit induction hypotheses.
- Not proving in both directions.

Here is an example of a valid proof:

Say that a string $w \in \{\mathtt{a}, \mathtt{b}\}^*$ is *a-leaning* if it contains at least as many as as bs. We show that $G$ generates the language of a-leaning strings.

(1) That every string in $L(G)$ is a-leaning can be shown by structural induction. Clearly $\epsilon$ and a are a-leaning. For the induction step, note that if $w$ and $w'$ are a-leaning, so are $\mathtt{a}w\mathtt{b}$, $\mathtt{b}w\mathtt{a}$, and $ww'$.

(2) That every a-leaning string is in $L(G)$ can be shown by induction on the length of strings. Consider any a-leaning $w$. If $|w| \leq 1$ then $w = \epsilon$ or $w = \mathtt{a}$ and hence $w$ can be generated from $S$. For the induction step, assume that every a-leaning $w$ of length less than $n$ can be generated from $S$. We show that any a-leaning $w$ with $|w| = n$ can be generated from $S$.

Case (2.1): If $w$ starts and ends with different symbols, then $w = \mathtt{a}w'\mathtt{b}$ or $w = \mathtt{b}w'\mathtt{a}$ for some a-leaning $w'$. By the induction hypothesis, $w'$ can be generated from $S$. But then so can $w$, owing to either rule $S \rightarrow \mathtt{a}S\mathtt{b}$ or rule $S \rightarrow \mathtt{b}S\mathtt{a}$.

Case (2.2): If $w$ starts and ends with b then some non-empty proper prefix $w_1$ of $w$ must have the same number of as and bs. Hence $w = w_1w_2$ with $w_1 \neq \epsilon$, $w_2 \neq \epsilon$, both of $w_1$ and $w_2$ a-leaning and of length less than $n$. By the induction hypothesis, both can be generated from $S$. But then so can $w$, owing to rule $S \rightarrow SS$.

Case (2.3): Otherwise $w$ starts and ends with $\mathsf{a}$. If $w$ has the same number of $\mathsf{a}$s and $\mathsf{b}$s then some non-empty proper prefix of $w$ must have the same number of $\mathsf{a}$s and $\mathsf{b}$s as well, and the argument then goes as in case (2.2). Otherwise $w$ has strictly more $\mathsf{a}$s than $\mathsf{b}$s, in which case $w = \mathsf{a}w'$ for some a-leaning $w'$. But $\mathsf{a}$ can be generated from $S$, and so can $w'$, by the induction hypothesis. Hence $w$ can be generated from $S$.

In all cases, $w \in L(G)$.

## Challenge 3a

No dramas there. $R^3$ is simply the concatenation of three regular languages. The class of regular languages is closed under concatenation, so $R^3$ is regular.

## Challenge 3b

Here the main issue was sloppy use of the pumping lemma.

*thrice*$(R)$ is not necessarily regular, and this should not come as a surprise, since we already know that $\{ww \mid w \in \Sigma^*\}$ is not regular (or even context-free). We can prove the claim formally with the pumping lemma for regular languages. For example, consider $R = \{\mathsf{0}, \mathsf{1}\}^*$, that is, the set of all finite binary strings. We use the pumping lemma for regular languages to show that *thrice*$(R)$ is not regular in this case.

Assume *thrice*$(R)$ is regular. Let $p$ be the pumping length, and consider the string

$$s = \mathsf{0}^p\mathsf{1}\mathsf{0}^p\mathsf{1}\mathsf{0}^p\mathsf{1}$$

Clearly, $s \in$ *thrice*$(R)$ and $s$ has length greater than $p$. The pumping lemma tells us that $s$ can be split into three segments, $s = xyz$, with $y \neq \epsilon$ and $|xy| \leq p$, in such as way that $xy^iz \in$ *thrice*$(R)$ for all $i \in \mathbb{N}$. From $|xy| \leq p$ it follows that (the non-empty) $y$ consists of zeros only. Hence $xz$ cannot be of the form required to be in *thrice*$(R)$. It follows that *thrice(R)* is not regular.

## Challenge 3c

This was the trickiest challenge up to this point, and quite a few simply skipped this sub-question. There are lots of possible solutions—all we have to do is find a single regular language $R$ and show that *snip*$(R)$ is not regular. There is no algorithm that I know of that will help find that $R$; it is a matter of trying out some simple (and maybe less simple) candidate $R$s and understand what *snip*$(R)$ becomes. This challenge is a good example of how we can make use of closure properties to good effect. For example, here is a solution:

Take the regular language $R = \mathsf{a}^*\mathsf{b}\mathsf{c}^*$ over alphabet $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. The language *snip*$(R)$ contains strings of two different forms:

a. $\mathsf{a}^i\mathsf{c}^i$, with $i \in \mathbb{N}$.

b. $\mathsf{a}^i\mathsf{b}\mathsf{c}^j$, with $i, j \in \mathbb{N}$ (and $i \neq j$).

The first form comes from those strings in $R$ for which the length is a multiple of 3, and $\mathsf{b}$ falls in the middle third of the string. The second form comes from those strings in $R$ for which the length is a multiple of 3, and $\mathsf{b}$ falls in the first or last third.

Hence *snip*$(R) \cap \mathsf{a}^*\mathsf{c}^* = \{\mathsf{a}^i\mathsf{c}^i \mid i \in \mathbb{N}\}$. That is, when we intersect *snip*$(R)$ with the regular language $\mathsf{a}^*\mathsf{c}^*$, we get a non-regular language. Since the class of regular languages is closed under intersection, this means *snip*$(R)$ is not regular.