Homework 04

Aaron Wang

February 21 2025

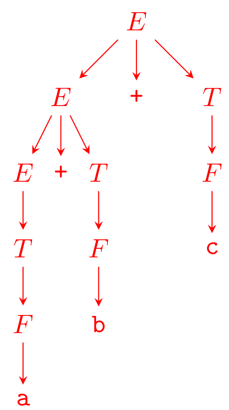
1. **Arithmetic expressions**. Consider the grammar G_4 (page 105) for arithmetic expressions, with start symbol E:

$$
\begin{aligned}
E &\to E \texttt{+} T \mid T \\
T &\to T \texttt{*} F \mid F \\
F &\to \texttt{(} E \texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{aligned}
$$

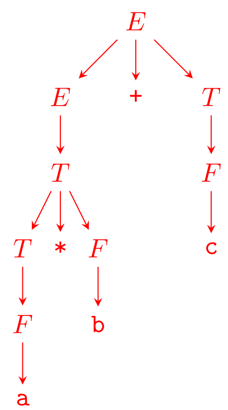
   (a) [cf. Exercise 2.1] Give derivations for the following strings. You may write them either as a sequence of rewrites ($E \implies \cdots$) or as a tree.

   i. `a + b + c`
   ii. `a * b + c`
   iii. `a * (b + c)`

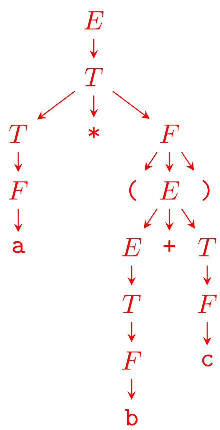
i.



ii.



iii.



   (b) Modify G_4 to allow an exponentiation operator $\uparrow$.

   - It should have *higher precedence* than multiplication; that is, in the derivation of the string `a * b` $\uparrow$ `c`, there should be a nonterminal that rewrites to `b` $\uparrow$ `c`, and there should not be a nonterminal that rewrites to `a * b`.
   - It should be (unlike `*` and `+`) *right-associative*; that is, in the derivation of the string `a` $\uparrow$ `b` $\uparrow$ `c`, there should be a nonterminal that rewrites to `b` $\uparrow$ `c`, and there should not be a nonterminal that rewrites to `a` $\uparrow$ `b`.

$$
\begin{aligned}
E &\to E \texttt{+} T \mid T \\
T &\to T \texttt{*} F \mid F \\
F &\to O \uparrow F \mid O \\
O &\to \texttt{(} E \texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{aligned}
$$

2. Write both a PDA and a CFG for the language (page 80):

$$C = \{w \in \{0,1\}^* | w \text{ has an equal number of } \texttt{0}\text{s and } \texttt{1}\text{s}\}.$$

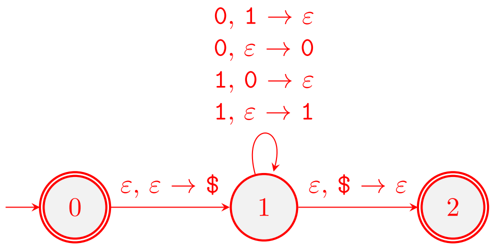
Please include a brief explanation of why they work. (If you design a PDA and then convert it to a CFG, your explanation for the CFG can simply be, "I converted my PDA to a CFG," and similarly if you convert a CFG to a PDA.)

The intuition for the following CFG is this. We need to design a CFG that accepts only balanced strings and every balanced string. This CFG clearly only accepts balanced strings as it only adds 0 and 1 simultaneously. Now for the other half, a balanced string must fall into at least one of two categories. The first is that $w = \texttt{0}x\texttt{1}$ or $w = \texttt{1}x\texttt{0}$ s.t. x is a balanced string (it is composed of a balanced substring wrapped by 0 and 1). The other is $w = xy$ s.t. x, y are balanced strings (It is a concatenation of two balanced substrings). Both of these cases are covered by the following CFG.

CFG for C with starting state S.

$$S \to \texttt{0}S\texttt{1} \mid \texttt{1}S\texttt{0} \mid SS \mid \varepsilon$$

The intuition for the following PDA is this. At each occurence of a character, we either pop the opposing character off the stack (if possible) or we add the current character onto the stack. This way, the stack will always be empty[1] if and only if an equal amount of occurences of each character exist.



[1] airquotes because it will contain the $ which signifies empty stack.

3. [Exercise 2.6b] Write both a PDA and a CFG for the language

$$L_3 = \overline{\{0^n 1^n | n \geq 0\}}.$$

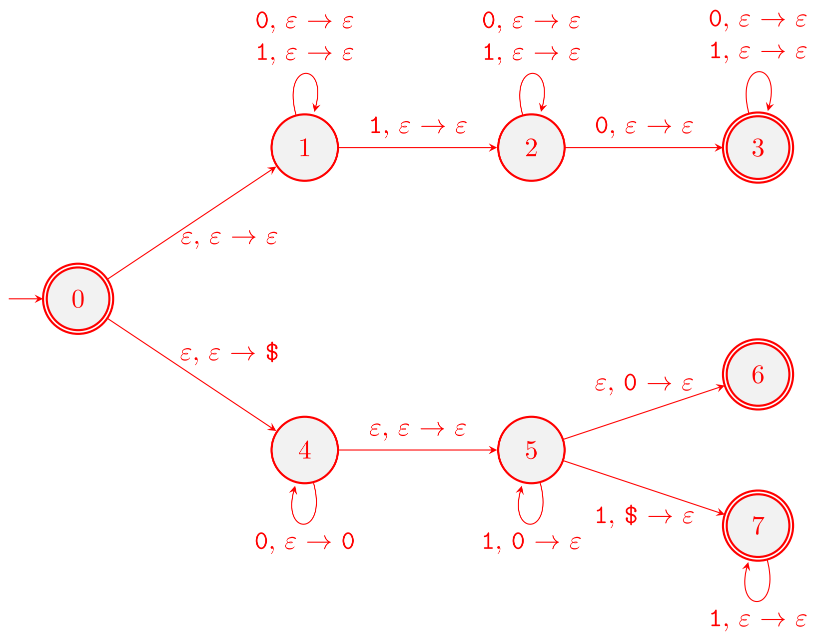
For example, $\texttt{000111} \notin L_3$. Please include a brief explanation of why they work. (If you design a PDA and then convert it to a CFG, your explanation for the CFG can simply be, "I converted my PDA to a CFG," and similarly if you convert a CFG to a PDA.)

Hint: First prove that this is equal to $\{0^m 1^n | m \neq n\} \cup \overline{0^* 1^*}$.

The intuition for the following CFG is this. The language L_3 accepts $\texttt{0}^m \texttt{1}^n$ s.t. $m \neq n$ and any string that contains a 1 before a 0. For the first half of this (Rules A, B, C starting with A), we will set the rules for the CFG such that we will create a string with equal $\texttt{0}$s and $\texttt{1}$s with a terminal in the middle. At the point when the min(m,n) is reached, we will then transfer to a non-terminal B that must add a $\texttt{0}$ and can only add $\texttt{0}$s if n is reached. The same applies for $\texttt{1}$s with non-terminal C. Now for the other half (Rules D, E starting with D),, we will have a starting state which must build a string that requires a $\texttt{1}$ to come before a $\texttt{0}$. After that we can add whatever characters we want however. Now union these two grammars to encapsulate L_3. The following CFG with start state S recognizes L_3

$$\begin{aligned}
S &\to A \mid D \\
A &\to \texttt{0}A\texttt{1} \mid B \mid C \\
B &\to \texttt{0}B \mid \texttt{0} \\
C &\to \texttt{1}C \mid \texttt{1} \\
D &\to E\texttt{1}E\texttt{0}E \\
E &\to \texttt{0}E \mid \texttt{1}E \mid \varepsilon
\end{aligned}$$

The intuition for the following PDA is this. The top half will recognize any string in which a $\texttt{1}$ comes before $\texttt{0}$ exactly as an NFA would. The bottom half works by creating a stack to push all the $\texttt{0}$s onto. It then uses all the $\texttt{1}$s to pop $\texttt{0}$s off the stack until there are no more $\texttt{1}$s or the stack only contains $\texttt{\$}$.[2] In the first case, this PDA uses an ε symbol and eats a $\texttt{0}$ off the stack to ensure that the stack was not empty, meaning $m > n$. In the second case, a $\texttt{1}$ eats the $\texttt{\$}$ off the stack (meaning $n > m$) and goes through the remaining $\texttt{1}$s in the string.



[2]The case of any strings not in the form $\texttt{0}^m\texttt{1}^n$ are also accounted for. if a $\texttt{0}$ ever comes after a $\texttt{1}$ it will automatically reject.