1 Book 1.32 [parallel addition]

Let Σ_3 be defined as the set of column vectors with three entries, where all entry values are either 0 or 1. A string of symbols in Σ_3 gives three rows of 0s and 1s. Consider each row to be a binary number and let

$$B = \{w \in \Sigma_3^* \mid \text{the bottom row of } w \text{ is the sum of the top two rows in binary}\}.$$

Show that B is regular. (Hint: Working with B^R is easier, where w^R is the reverse of the string w. You may assume that if w is regular, then w^R is regular).

Solution:

To prove that B is regular, we will instead demonstrate that B^R is regular, utilizing the provided hint. Since B^R consists of strings where the rightmost column is read first (considering B), we can define a DFA to process B^R. By proving that B^R is regular, and given that the reverse of a regular language is regular, we can conclude that B is also regular.

We first define our DFA, D as follows. Our alphabet is defined as $\Sigma = \{000, 011, 101, 110\}$, which represents the only four valid column configurations where the last digit is the sum of the first two. The states of the D can be written as $Q = \{q_0, q_1\}$, where q_0 is the state when there is no carry from the previous column, and q_1 is the state when there is a carry present from the previous column.

We define our transition function δ to be

- From q_0:
  - on 000: stay in q_0, since this represents $0 + 0 = 0$ with no carries.
  - on 011: stay in q_0, since this represents $0 + 1 = 1$ with no carries.
  - on 101: stay in q_0, since this represents $1 + 0 = 1$ with no carries.
  - on 110: move to q_1, since this represents $1 + 1 = 0$ with a carry of 1.
- From q_1:
  - on 011: move to q_0, since this represents $0 + 1 = 1$, with an additional carry.
  - on 101: move to q_0, since this represents $1 + 0 = 1$, with an additional carry.
  - on 110: stay in q_1, since this represents $1 + 1 = 0$, with an additional carry.

Naturally, our start state is q_0, and our set of accept states is $F = \{q_0\}$. This is because we start with no carries, and a valid configuration of Σ_3^* must also have no carries.

Since D can recognize strings in B^R by processing each column vector from right to left (considering the string in B). Thus, B^R is regular. Given that the reverse of a regular language is regular, B is also regular. This completes our proof.

2 Book 1.41 [regular closure under perfect shuffle]

For languages A and B, let the **perfect shuffle** pf A and B be the language

$$\{w | w = a_1 b_1 \dots a_k b_k, \text{ where } a_1 \dots a_k \in A \text{ and } b_1 \dots b_k \in B, \text{ each } a_i, b_i \in \Sigma\}$$

Show that the class of regular languages is closed under perfect shuffle.

Solution:

To show that the class of regular languages is closed under the perfect shuffle operation, we need to demonstrate that if A and B are regular languages, then their perfect shuffle $pf(A, B)$ is also a regular language.

Given that A and B are regular, there must exist DFAs $M_A = (Q_A, \Sigma, \delta_A, q_{0A}, F_A)$ and $M_B = (Q_B, \Sigma, \delta_B, q_{0B}, F_B)$ that recognize A and B, respectively.

Let us construct a new DFA M_{pf} that recognizes $pf(A, B)$. The idea is to simulate both M_A and M_B simultaneously, by walking through the shuffled string in a synchronized manner.

We define our M_{pf} as follows. Let the states be $Q_{pf} = Q_A \times Q_B$, such that each state in M_{pf} corresponds to a pair of states, one from M_A and one from M_B. We define the alphabet of M_{pf} to be $\Sigma_{pf} = \Sigma \times \Sigma$. The transition function is $\delta_{pf}((p, q), (a, b)) = (\delta_A(p, a), \delta_B(q, b))$, where $(p, q) \in Q_{pf}$ and $a, b \in \Sigma$. Finally, our start state is $q_{0pf} = (q_{0A}, q_{0B})$, and the set of accepting states are $F_{pf} = \{(p, q) | p \in F_A \text{ and } q \in F_B\}$.

For a string $w = a_1 b_1 \dots a_k b_k$ in the perfect shuffle of A and B, M_{pf} starts in state q_{0pf} and processes the pairs a_i, b_i using the transition function. It will end up in an accepting state if and only if the a_i sequence leads M_A to an accepting state and the b_i sequence leads M_B to an accepting state.

Thus, M_{pf} recognizes $pf(A, B)$ and since $pf(A, B)$ can be recognized by a DFA, it is regular. This means that the class of regular languages is closed under the perfect shuffle operation, and this concludes our proof.

3 Book 1.53 [sequential addition]

Let $\Sigma = \{0, 1, +, =\}$ and

$$ADD = \{x = y + z | x, y, z \text{ are binary integers, and } x \text{ is the sum of } y \text{ and } z\}.$$

Show that ADD is not regular.

Solution:

To show that ADD is not regular, we use the Pumping Lemma for regular languages. We assume for the sake of contradiction that ADD is regular. Then, there exists a pumping length p. Let us choose the string s to be $10^p = 1^p + 1^p$, which is a string that represents a valid addition in binary.

Now let us apply the Pumping Lemma. Given the structure of s, if we write $s = xyz$, where the conditions from the Pumping Lemma hold, the pump-able substring y can have several forms:

1. y contains only 1s from the leftmost section (before the = sign): $y = 1^k$ for some $k > 0$.
2. y contains only 0s: $y = 0^k$ for some $k \leq p$.
3. y spans across the = or + signs. But this cannot happen due to the Pumping Lemma restriction that $|xy| \leq p$.
4. y contains only 1s from the section after the + sign: $y = 1^k$ for some $k > 0$.

Now consider the pumped string xy^2z for each scenario. We omit the third scenario from consideration as it cannot occur.

If y contains only 1s from the leftmost section, then the pumped string will increase the value on the left side of the equation without changing the right side, resulting in an invalid equation. For the second scenario, if y consists of only 0s, then the pumped string will again increase the value on the left side of the equation without changing the right side, leading to an invalid equation. Finally, if y contains only 1s from the section after the + sign, then the pumped string will increase one of the values on the right side of the equation without changing the left side, leading to an invalid equation.

In all the possible configurations for y, pumping y to form xy^2z results in a string that is not in ADD. This is a contradiction, and thus our initial assumption that ADD is regular must be false. Therefore, this concludes our proof that ADD is not regular.

4 Book 1.60 [small NFA for: $\{w|w$ contains an **a** which exactly k places from the end$\}$]

Let $\Sigma = \{a, b\}$. For each $k \geq 1$, let C_k be the language consisting of all strings that contain an a exactly k places from the right-hand end. Thus, $C_k = \Sigma^* a \Sigma^{k-1}$. Describe an NFA with $k + 1$ states that recognizes C_k in terms of both a state diagram and a formal description.

Solution:

Given $k \geq 1$, we want to design an NFA to accept strings where there is an a exactly k positions from the end of the string. This means that the NFA will need to use non-deterministic branching every time it sees an a and continue transitioning for $k-1$ more characters, at which point it should be in the accept state if the input string is in the language.

Non-determinism is required since it is not known if the encountered a is the one that is $k-1$ steps from the end of the string. If the NFA reaches the end of the input string before this, then that branch of the computation rejects. If at least one branch accepts, the NFA accepts, otherwise it rejects the input string.

We formally describe the NFA below.

- **States**: $Q = \{q_0, q_1, \ldots, q_{k+1}\}$
- **Alphabet**: $\Sigma = \{a, b\}$
- **Transitions**:
  - $\delta(q_0, a) = \{q_0, q_1\}$ (the read a could be one that is $k-1$ steps from the end, or not).
  - $\delta(q_0, b) = \{q_0\}$ (we ignore characters until we see the first occurrence of a).
  - $\delta(q_1, a) = \{q_1, q_2\}$ (either this a or the previous one could be $k-1$ steps from the end).
  - $\delta(q_1, b) = \{q_2\}$ (if $k > 1$) (it would be an accepting state if $k = 1$).
  - For $2 \leq i < k$:
    - $\delta(q_i, a) = \{q_1, q_{i+1}\}$ (the read a could be one that is $k-1$ steps from the end, or it is part of the final $k-1$ characters).
    - $\delta(q_i, b) = \{q_{i+1}\}$ (the input forms part of the final $k-1$ characters).
  - $\delta(q_k, a) = \delta(q_k, b) = \{q_{k+1}\}$ (always accepts if the final character is valid).
- **Start State**: q_0
- **Accept State**: q_{k+1}

A state diagram follows naturally from the above description.

5 Book 1.61 [any DFA for above language must be large]

Consider the languages C_k defined in Problem 1.60. Prove that for each k, no DFA can recognize C_k with fewer than 2^k states.

Solution:

Suppose for the sake of contradiction that there exists a DFA D that recognizes C_k and has fewer than 2^k states. Consider running the DFA D on all strings of length k. Since there are 2^k such strings and D has fewer than 2^k states, by the pigeonhole principle, there must be at least two different strings, say w_1 and w_2, that cause D to end up in the same state after processing the strings.

The state must have the same behavior regarding any further inputs that are the same, since the DFA doesn't have memory of past inputs beyond its current state. Consider the following possible cases if the strings were extended in length (i.e. $w_1 b^{k-1}$ and $w_2 b^{k-1}$).

If both w_1 and w_2 end in an a, appending b^{k-1} to both will result in both $w_1 b^{k-1}$ and $w_2 b^{k-1}$ being in C_k. D would accept both strings.

If both w_1 and w_2 do not end in an a, appending b^{k-1} to both will result in neither of the strings being in C_k. D would reject both strings.

Now suppose without loss of generality that w_1 ends in an a and w_2 doesn't. If we append b^{k-1} to both, $w_1 b^{k-1}$ is in C_k, but $w_2 b^{k-1}$ isn't. However, since D is in the same state after reading both w_1 and w_2, it cannot differentiate between these two cases and will either accept both strings or reject both, which gives our desired contradiction.

Thus, no DFA can recognize C_k with fewer than 2^k states.

6 Book 2.24 [very tricky CFL]

Let $E = \{a^i b^j | i \neq j \text{ and } 2i \neq j\}$. Show that E is a context-free language.

Hint: Think about this language in a different way.

Solution:

To show that a language is context-free, one common technique is to find a CFG that generates it. Given E as defined in the problem statement, we use the hint and consider what the language is not. Since the language of E is defined by what it is not equal to, we consider the complement $\overline{E}$, which contains the strings where $i = j$ or $2i = j$.

$\overline{E}$ can be described by the languages $F_1 = \{a^i b^i | i \geq 0\}$ (where $i = j$), and $F_2 = \{a^i b^{2i} | i \geq 0\}$ (where $2i = j$). It is trivial to show that the CFG defined by $S_1 \to aS_1b|\epsilon$ generates F_1, and the CFG defined by $S_2 \to aS_2bb|\epsilon$ generates F_2.

Consider the regular language $R = a^*b^*$. This is the set of all strings with any number of as followed by any number of bs. E consists of strings in R that are not in $F_1 \cup F_2$. This can be framed as a set difference of $R \setminus (F_1 \cup F_2)$. While CFLs aren't closed under set difference, we are performing a set difference with a regular language, which has more flexible properties.

Given that regular languages are closed under complementation and CFLs are closed under intersection with regular languages, the set difference of a regular language and CFL can be equivalently represented by an intersection of the complement of the CFL (within the regular language) and the regular language itself.

So consider the complement of $F_1 \cup F_2$ within R. This complement is regular since it is defined within the regular language R. Then E can be equivalently defined as $R \cap (\overline{R} \cap (\overline{F_1 \cup F_2}))$. To elaborate, $\overline{R}$ is the complement of R within the universe of all strings. Since $\overline{R}$ is also a regular language (because regular languages are closed under complementation) and the intersection of a regular language and CFL is a CFL, the resulting language is context-free.

Since R is a regular language and $\overline{R} \cap (\overline{F_1 \cup F_2})$ is context-free, so then E is a context-free language. This concludes our proof.

7 Book 2.27 [if-then-else ambiguous grammar]

Let $G = (V, \Sigma, R, \langle STMT \rangle)$ be the following grammar.

$$\begin{aligned}
\langle STMT \rangle &\to \langle ASSIGN \rangle | \langle IF-THEN \rangle | \langle IF-THEN-ELSE \rangle \\
\langle IF-THEN \rangle &\to \text{if condition then } \langle STMT \rangle \\
\langle IF-THEN-ELSE \rangle &\to \text{if condition then } \langle STMT \rangle \text{ else } \langle STMT \rangle \\
\langle ASSIGN \rangle &\to \text{ a := 1}
\end{aligned}$$

$$\begin{aligned}
\Sigma &= \{\text{if, condition, then, else, a:= 1}\} \\
V &= \{\langle STMT \rangle, \langle IF-THEN \rangle, \langle IF-THEN-ELSE \rangle, \langle ASSIGN \rangle\}
\end{aligned}$$

G is a natural-looking grammar for a fragment of a programming language, but G is ambiguous.

1. Show that G is ambiguous.
2. Give a new unambiguous grammar for the same language.

You do not need to prove your grammar works, but adding a few comments about why it does work might help the grader.

Solution:

To show that G is ambiguous, we need to prove that there exists a string that can be derived in two different parse trees. The ambiguity int his grammar arises due to the "dangling else" problem. Let us consider the following statement: `if condition then if condition then a:=1 else a:= 1`. This statement can be interpreted in the two following ways:

1. The "else" belongs to the first "if":

$$\begin{aligned}
\langle STMT \rangle &\to \langle IF-THEN-ELSE \rangle \\
&\to \text{ if condition then } \langle STMT \rangle \text{ else } \langle STMT \rangle \\
&\to \text{ if condition then } \langle IF-THEN \rangle \text{ else } \langle STMT \rangle \\
&\to \text{ if condition then (if condition then } \langle STMT \rangle) \text{ else } \langle STMT \rangle \\
&\to \text{ if condition then (if condition then a := 1) else a := 1}
\end{aligned}$$

2. The "else" belongs to the second "if":

$$\begin{aligned}
\langle STMT \rangle &\to \langle IF-THEN \rangle \\
&\to \text{ if condition then } \langle STMT \rangle \\
&\to \text{ if condition then } \langle IF-THEN-ELSE \rangle \\
&\to \text{ if condition then (if condition then } \langle STMT \rangle \text{ else } \langle STMT \rangle) \\
&\to \text{ if condition then (if condition then a := 1 else a := 1)}
\end{aligned}$$

Both grammars are valid according to G, proving that G is ambiguous.

We now turn to giving a new unambiguous grammar for the same language. We can fix the ambiguity by ensuring that every "else" clause is associated with the closest preceding "if" that doesn't already have an "else". One way to do this is by distinguishing between statements that end with an "else" clause and those that do not:

$$\begin{aligned}
\langle STMT \rangle &\to \langle MATCHED \rangle | \langle UNMATCHED \rangle \\
\langle MATCHED \rangle &\to \langle ASSIGN \rangle | \langle MATCHED - IF - THEN - ELSE \rangle \\
\langle MATCHED - IF - THEN - ELSE \rangle &\to \text{if condition then } \langle MATCHED \rangle \text{ else } \langle MATCHED \rangle \\
\langle UNMATCHED \rangle &\to \text{if condition then } \langle STMT \rangle | \\
&\qquad \text{if condition then } \langle MATCHED \rangle \text{ else } \langle UNMATCHED \rangle \\
\langle ASSIGN \rangle &\to \text{ a} := 1
\end{aligned}$$

A $\langle MATCHED \rangle$ statement is either an assignment or an if-then-else where both the "then" and "else" parts are also $\langle MATCHED \rangle$ statements. On the other hand, an $\langle UNMATCHED \rangle$ statement is an if-then statement where the "then" part can be any statement or an if-then-else where the "then" part is a $\langle MATCHED \rangle$ statement and the "else" part is an $\langle UNMATCHED \rangle$ statement.

This ensures that the "else" is always matched with the nearest "if", thereby removing the ambiguity.

8 (optional) Book 1.59 [synchronizing sequence]

Solution: