1 Finite State Automata and Regular Language

The NFA $A$:

![Diagram of NFA A](image)

(i) The DFA $B$:

![Diagram of DFA B](image)
(ii) Our initial split differentiates non-final and final states:

\[ \left[ [ S, S_2 ], [ S_0, S_1, S_{12} ] \right] \]

Testing the non-final group with \( b \) produces a split, as for \( S \) we stay in the non-final group, while for \( S_2 \) we go to the final group:

\[ \left[ [ S ], [ S_2 ], [ S_0, S_1, S_{12} ] \right] \]

Testing the final group with \( a \) produces another split, as for \( S_0 \) we go to the group \([ S ]\), while for \( S_1 \) and \( S_{12} \) we go to the group \([ S_2 ]\):

\[ \left[ [ S ], [ S_2 ], [ S_0 ], [ S_1, S_{12} ] \right] \]

The only non-singleton group left, \([ S_1, S_{12} ]\), cannot be split by any test: on \( a \) we go to \([ S_2 ]\), while on \( b \) we stay in \([ S_1, S_{12} ]\). The algorithm therefore terminates, with \( S_1 \) and \( S_{12} \) the only equivalent states.

(iii) We convert \( B \) above into the minimal \( C \) by deleting the state \( S_{12} \) and re-assigning the arc from \( S_1 \) to \( S_{12} \) to be a loop on \( S_1 \):

\[
\begin{array}{c}
\{a, b\} \\
S \\
S_0 \quad S_1 \quad S_2 \\
\end{array}
\]

(iv) Note that a string is not in the language \( L \) if and only if

- it has \( a \) as its first letter, or
- it has \( a \) as its last letter, or
- it contains the substring \( aa \).

**Subgoal one:** If a string \( w \) is in \( L \), then \( C \) accepts \( w \).

**Subgoal two:** If \( C \) accepts a string \( w \), then \( w \in L \).

As suggested by the hint, we reformulate both these goals for proof by contrapositive:

**Subgoal one (contrapositive):** If \( C \) rejects a string \( w \), then \( w \notin L \).
Subgoal two (contrapositive): If \( w \notin L \), then \( C \) rejects \( w \).

Proof of subgoal one (contrapositive): If \( C \) rejects \( w \) then \( N^*(S_0, w) = S_2 \) or \( N^*(S_0, w) = S \). We consider each case separately.

\( N^*(S_0, w) = S_2 \): the only arc into \( S_2 \) is labelled by \( a \), so \( w \) must have the form \( \alpha a \) for some string \( \alpha \). But any string ending in \( a \) is not in \( L \).

\( N^*(S_0, w) = S \): because \( S \) is not the start state, the DFA must have transitioned into \( S \) from other state at some stage. The only arcs into \( S \) from other states are labelled by \( a \), so \( w \) must have the form \( \alpha a \beta \) for some strings \( \alpha, \beta \). There are two subcases then to consider, depending on which incoming arc was used: \( N^*(S_0, \alpha) = S_0 \) or \( N^*(S_0, \alpha) = S_2 \). We consider each subcase separately.

\( N^*(S_0, \alpha) = S_0 \): there are no arcs into \( S_0 \) except the ‘start state’ arc, so the only way to validate this equation is if \( \alpha \) is the empty string \( \epsilon \). But then \( w = \alpha a \beta = a \beta \), and any string starting with \( a \) is not in \( L \).

\( N^*(S_0, \alpha) = S_2 \): as we argued above, this can only hold if \( \alpha = \gamma a \) for some string \( \gamma \). But then \( w = \alpha a \beta = \gamma aa \beta \), and any string \( w \) containing a substring \( aa \) is not in \( L \).

Proof of subgoal two (contrapositive):

First, observe the lemma

\[
N^*(S, w) = S
\quad \text{(Lemma)}
\]

for all strings \( w \). This holds because all arcs out of \( S \) go back to \( S \), so there is ‘no escape’.

Suppose \( w \notin L \). There are, as noted above, three cases by which this could be, and we consider each separately.

\( w \) starts with \( a \):

\[
N^*(S_0, w) = N^*(S_0, a\alpha) \quad \text{(for some string \( \alpha \))}
\]

\[
= N^*(N(S_0, a), \alpha) \quad \text{(Definition of \( N^* \))}
\]

\[
= N^*(S, \alpha)
\]

\[
= S \quad \text{(Lemma)}
\]

and \( S \) is a non-final state, so \( w \) is rejected.

\( w \) ends with \( a \): here \( w = \alpha a \) for some \( \alpha \). Now \( N^*(S_0, \alpha a) = N(N^*(S_0, \alpha), a) \) by the corollary to the Append Theorem. We have no idea what \( N^*(S_0, \alpha) \) is, so we consider all four possible cases:

\[
N^*(S_0, a) = S \quad N^*(S_2, a) = S
\]

\[
N^*(S_1, a) = S_2 \quad N^*(S, a) = S
\]

\( S \) and \( S_2 \) are both non-final, so \( w \) is rejected.
$w$ contains substring $aa$: here $w = \alpha aa \beta$ for some $\alpha, \beta$. Now $N^*(S_0, \alpha aa \beta) = N^*(N^*(S_0, \alpha), aa \beta) = N^*(N^*(N^*(S_0, \alpha), aa), \beta)$ by the Append Theorem. We do not know what $N^*(S_0, \alpha)$ is, so we investigate what $N^*(N^*(S_0, \alpha), aa)$ might be by all four cases again:

$$
\begin{align*}
N^*(S_0, aa) &= S & N^*(S_2, aa) &= S \\
N^*(S_1, aa) &= S & N^*(S, aa) &= S
\end{align*}
$$

So $S$ is the answer no matter what. Now

$$
\begin{align*}
N^*(S_0, w) &= N^*(N^*(N^*(S_0, \alpha), aa), \beta) \\
&= N^*(S, \beta) \\
&= S \quad \text{(Lemma)}
\end{align*}
$$

$S$ is non-final, so $w$ is rejected.

(v) The automaton below will be at state $S_2$ if it has just read $b$; it will be at state $S_1$ if it has just read $a$ without $b$ to its left (in this case we will reject unless the next token is $b$); it will be at $S_0$ if the string has just started, or we have just read $ba$ (in this case reading an $a$ next is legal, but will put us ‘in danger’ of rejection by transitioning to $S_1$).

![Automaton Diagram]
2 Pushdown Automata and Context-Free Language

1 Grammars

(i) Find a right-linear grammar which accepts the same language as this NFA. Your grammar may (unlike the definition in lectures) contain productions of the form $A \rightarrow B$, thus it may contain the following sorts of productions:

$$A \rightarrow a \quad \text{or} \quad A \rightarrow aB \quad \text{or} \quad A \rightarrow \epsilon \quad \text{or} \quad A \rightarrow B$$

(It is understood that upper case letters are non-terminals, and lower-case letters are terminals).

**Solution**  The natural way of doing this is where a non-terminal can be expanded to the strings which would be accepted by the machine starting at the corresponding state. Thus we get

$$S \rightarrow S_0 \quad S_1 \rightarrow bS_2$$
$$S_0 \rightarrow aS_1 \quad S_2 \rightarrow S_1$$
$$S_0 \rightarrow bS_2 \quad S_2 \rightarrow aS_2$$
$$S_1 \rightarrow \epsilon$$

(ii) Describe how to convert a grammar containing productions of the form $A \rightarrow B$ into a grammar which contains no productions of this form, but accepts the same language.

**Solution**  Firstly, if there are any sets of productions which form a “cycle”, such as

$$A \rightarrow B \quad B \rightarrow C \quad C \rightarrow A$$

(so that $A \Rightarrow^* A$), replace each of $A, B, C$ wherever it occurs by $A$.

Secondly, for any production $A \rightarrow B$, where the grammar also contains productions $B \rightarrow \beta_i$ (for some set of $i$s) replace $A \rightarrow B$ by $A \rightarrow \beta_i$ for each such $i$. (Do not delete $B \rightarrow \beta_i$).
(An alternative to this second step is to say: for any production \( A \to B \), where the grammar also contains productions \( C_i \to \alpha_i \) where \( \alpha_i \) contains \( A \), let \( \beta_i \) be \( \alpha_i \) with \( A \) replaced by \( B \). Then replace \( A \to B \) by \( C_i \to \beta_i \) for each \( i \). However this step doesn’t quite work when \( A \) is the starting symbol \( S \)).

**Question (continued)** If your answer to part (a) contains productions of this form, convert your grammar to a right-linear grammar as defined in lectures (ie, not containing productions of the form \( A \to B \)), which accepts the same language.

**Solution** Considering the approach above the first step does not apply. So we get

\[
\begin{align*}
S & \to aS_1 & S_1 & \to bS_2 \\
S & \to bS_2 & S_2 & \to \epsilon \\
S_0 & \to aS_1 & S_2 & \to BS_2 \\
S_0 & \to bS_2 & S_2 & \to aS_2 \\
S_1 & \to \epsilon & S_2 & \to S_2 \\
\end{align*}
\]

Now notice that \( S_0 \) cannot appear in any sentential form (ie, you can’t get a string containing \( S_0 \) from \( S \)). So you can delete productions \( S_0 \to \ldots \) without changing the language. So this gives

\[
\begin{align*}
S & \to aS_1 & S_1 & \to bS_2 \\
S & \to bS_2 & S_2 & \to \epsilon \\
S_1 & \to \epsilon & S_2 & \to bS_2 & \\
S_2 & \to aS_2 \\
\end{align*}
\]

(iii) Find a left-linear grammar which accepts the same language as this NFA. Your grammar may (unlike the definition in lectures) contain productions of the form \( A \to B \), thus it may contain the following sorts of productions:

\[
A \to a \quad \text{or} \quad A \to Ba \quad \text{or} \quad A \to \epsilon \quad \text{or} \quad A \to B
\]

**Solution** The natural way of doing this is where a non-terminal can be expanded to the strings which would be accepted by the machine if the corresponding state were a final state. Thus we get

\[
\begin{align*}
S & \to S_1 & S_2 & \to S_0b \\
S_1 & \to S_0a & S_2 & \to S_1b \\
S_1 & \to S_2 & S_2 & \to S_2a \\
S_0 & \to \epsilon
\end{align*}
\]
2 A Content Comparison Problem

Here is a deterministic PDA that accepts the language over \{a, b\} where each string in the language has more a’s than b’s. The input strings string will be terminated by a hash symbol (\#). (Thus the input alphabet is \{a, b, \#\}.)

For this PDA, strings will be deemed to be accepted when the PDA consumes the input and empties the stack.

\[
\begin{align*}
\delta(q_1, a, Z) & \mapsto (q_1, aZ) & \delta(q_1, b, Z) & \mapsto (q_1, bZ) \\
\delta(q_1, a, a) & \mapsto (q_1, aa) & \delta(q_1, b, a) & \mapsto (q_1, \epsilon) \\
\delta(q_1, a, b) & \mapsto (q_1, \epsilon) & \delta(q_1, b, b) & \mapsto (q_1, bb) \\
\delta(q_1, \#, a) & \mapsto (q_2, \epsilon) & \delta(q_2, \#, Z) & \mapsto (q_2, \epsilon) \\
\delta(q_2, \#, a) & \mapsto (q_2, \epsilon) & &
\end{align*}
\]

The idea of the above design is that

i) The PDA will remain in state \(q_1\) while the input is scanned only going to state \(q_2\) when the end-of-input symbol is reached (with an \(a\) on the stack).

ii) If, in state \(q_1\), \(n\) more of one letter have been seen that the other letter then the stack will contain \(n\) copies of that letter.

iii) When the PDA goes to state \(q_2\) excess copies of \(a\) will be erased and the string will be accepted.

An alternative presentation of the next state relation is preferred by some people.

\[
\begin{array}{cccccccccccc}
& a, Z & b, Z & \#, Z & a, a & b, a & \#, a & a, b & b, b & \#, b & \epsilon, a & \epsilon, Z \\
q_1 & q_1, aZ & q_1, bZ & q_1, \epsilon & q_1, aa & q_1, \epsilon & q_1, \epsilon & q_1, \epsilon & q_1, bb & q_1, \epsilon & q_1, \epsilon & q_1, \epsilon \\
q_2 & \quad - & \quad - & \quad - & \quad - & \quad - & \quad - & \quad - & \quad - & \quad - & \quad - & \quad - \\
\end{array}
\]

3 Palindrome Acceptance

The exercise is to design an NPDA (Nondeterministic PDA) that recognises nonempty palindromes over the alphabet \{a, b\}. (Palindromes are strings that read the same forwards or backwards). The strings that it is required to accept include \(ababbaba\) and \(ababa\).

Strings are deemed to be accepted when the input is consumed and the stack is empty.

Palindrome checking with a PDA has an obvious strategy involving one state in which the symbols up to the mid-point are pushed onto the stack and another state in which a check is made that the remaining symbols in the input match those on the stack (in reverse order). Non-determinism is inevitably required in this problem to ‘guess’ the midpoint.

The palindrome -recognising PDA (with stack alphabet \{a, b, Z\} and initial state \(q_0\)) is defined by its next state relation.
... and here it is in tabular form.

<table>
<thead>
<tr>
<th></th>
<th>a, Z</th>
<th>b, Z</th>
<th>a, a</th>
<th>b, a</th>
<th>a, b</th>
<th>b, b</th>
<th>e, a</th>
<th>e, b</th>
<th>e, Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>q₀, aZ</td>
<td>q₀, bZ</td>
<td>q₀, aa</td>
<td>q₀, ba</td>
<td>q₀, ab</td>
<td>q₀, bb</td>
<td>q₁, ε</td>
<td>q₁, ε</td>
<td>—</td>
</tr>
<tr>
<td>q₁</td>
<td>—</td>
<td>—</td>
<td>q₁, ε</td>
<td>—</td>
<td>—</td>
<td>q₁, ε</td>
<td>—</td>
<td>—</td>
<td>q₁, ε</td>
</tr>
</tbody>
</table>

3 Turing Machine and Computability

1 Palindrome Acceptance

The exercise is to construct a Turing Machine that recognizes palindromes over the alphabet \{a, b\}.

The precondition for the computation is that the tape of the TM will just have a string made up of a’s and b’s with the read head somewhere over the string.

The machine will halt if (and only if) the string is a palindrome. In the following solution, when the machine halts the tape will have just a single \(T\) (for \(T\)rue) on the tape and the read head above it.

The machine works by iteratively checking that the leftmost and rightmost symbols on the tape are the same. After each successful match those two symbols are erased. The looping terminates successfully when either

a) At the start of an iteration the leftmost symbol is also the rightmost one.

b) After erasure of matching symbols the tape is blank.
2 Computability

Suppose for contradiction that there exists a TM, $T$, that decides whether an arbitrary TM, $X$, ever prints a given letter when provided input, $w$. Let $M$ be a TM which will construct a new TM, $X'$, as a copy of $X$ whose alphabet is the same as that of $X$ with one new character, $\alpha$. Let $M$ modify all transitions in $X'$ which cause the machine to halt to also write $\alpha$ to the tape. By the assumption, $M$ could now run $T$ with $X$ and $w$ as input and decide whether it ever writes $\alpha$ to the tape. However, this would mean that $M$ provides a solution to the Halting Problem (since $X'$ printing $\alpha$ would only occur if $X$ had halted). Since the halting problem is undecidable, we have a contradiction, hence, this problem is undecidable.