3.12: Closure Properties of Regular Languages

In this section, we show how to convert regular expressions to finite automata, as well as how to convert finite automata to regular expressions.

3.12: Closure Properties of Regular Languages

In this section, we show how to convert regular expressions to finite automata, as well as how to convert finite automata to regular expressions.

As a result, we will be able to conclude that the following statements about a language L are equivalent:

- L is regular;
- L is generated by a regular expression;
- L is accepted by a finite automaton;
- L is accepted by an EFA;
- L is accepted by an NFA; and
- L is accepted by a DFA.

Introduction

Also, we will introduce:

- operations on FAs corresponding to union, concatenation and closure;
- an operation on EFAs corresponding to intersection; and
- an operation on DFAs corresponding to set difference.

Introduction

Also, we will introduce:

- operations on FAs corresponding to union, concatenation and closure;
- an operation on EFAs corresponding to intersection; and
- an operation on DFAs corresponding to set difference.

As a result, we will have that the set **RegLan** of regular languages is closed under union, concatenation, closure, intersection and set difference. I.e., we will have that, if $L, L_1, L_2 \in$ **RegLan**, then $L_1 \cup L_2$, $L_1 L_2$, L^*, $L_1 \cap L_2$ and $L_1 - L_2$ are in **RegLan**.

Introduction

Also, we will introduce:

- operations on FAs corresponding to union, concatenation and closure;
- an operation on EFAs corresponding to intersection; and
- an operation on DFAs corresponding to set difference.

As a result, we will have that the set **RegLan** of regular languages is closed under union, concatenation, closure, intersection and set difference. I.e., we will have that, if $L, L_1, L_2 \in$ **RegLan**, then $L_1 \cup L_2$, $L_1 L_2$, L^*, $L_1 \cap L_2$ and $L_1 - L_2$ are in **RegLan**.

The book shows several additional closure properties of regular languages.

Operations on FAs

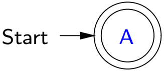
We write **emptyStr** for the DFA

and **emptySet** for the DFA

Thus, we have that $L(\textbf{emptyStr}) = \{\%\}$ and $L(\textbf{emptySet}) = \emptyset$.
Of course **emptyStr** and **emptySet** are also NFAs, EFAs and FAs.

Operations on FAs

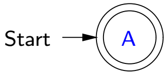
We write **emptyStr** for the DFA



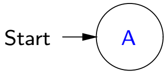
and **emptySet** for the DFA

Thus, we have that $L(\textbf{emptyStr}) = \{\%\}$ and $L(\textbf{emptySet}) = \emptyset$.
Of course **emptyStr** and **emptySet** are also NFAs, EFAs and FAs.

We write **emptyStr** for the DFA



and **emptySet** for the DFA



Thus, we have that $L(\textbf{emptyStr}) = \{\%\}$ and $L(\textbf{emptySet}) = \emptyset$.
Of course **emptyStr** and **emptySet** are also NFAs, EFAs and FAs.

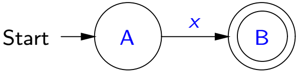
Functions for Building Simple FAs

Next, we define a function $\mathbf{strToFA} \in \mathbf{Str} \to \mathbf{FA}$ by: $\mathbf{strToFA}\, x$ is the FA

Thus, for all $x \in \mathbf{Str}$, $L(\mathbf{strToFA}\, x) = \{x\}$.

Functions for Building Simple FAs

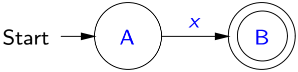
Next, we define a function $\mathbf{strToFA} \in \mathbf{Str} \to \mathbf{FA}$ by: $\mathbf{strToFA}\, x$ is the FA



Thus, for all $x \in \mathbf{Str}$, $L(\mathbf{strToFA}\, x) = \{x\}$.

Functions for Building Simple FAs

Next, we define a function $\textbf{strToFA} \in \textbf{Str} \to \textbf{FA}$ by: $\textbf{strToFA}\, x$ is the FA



Thus, for all $x \in \textbf{Str}$, $L(\textbf{strToFA}\, x) = \{x\}$.

It is also convenient to define a function $\textbf{symToNFA} \in \textbf{Sym} \to \textbf{NFA}$ by: $\textbf{symToNFA}\, a = \textbf{strToFA}\, a$. Of course, $\textbf{symToNFA}$ is also an element of $\textbf{Sym} \to \textbf{EFA}$ and $\textbf{Sym} \to \textbf{FA}$. Furthermore, for all $a \in \textbf{Sym}$, $L(\textbf{symToNFA}\, a) = \{a\}$.

Unions of FAs

Next, we define a function/algorithm **union** $\in$ **FA** $\times$ **FA** $\to$ **FA** such that $L(\textbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$, for all $M_1, M_2 \in$ **FA**. If $M_1, M_2 \in$ **FA**, then **union**(M_1, M_2) is the FA N such that:

- $Q_N =$
- $s_N =$
- $A_N =$
- $T_N =$

Unions of FAs

Next, we define a function/algorithm $\mathbf{union} \in \mathbf{FA} \times \mathbf{FA} \to \mathbf{FA}$ such that $L(\mathbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$, for all $M_1, M_2 \in \mathbf{FA}$. If $M_1, M_2 \in \mathbf{FA}$, then $\mathbf{union}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle 1, q\rangle \mid q \in Q_{M_1}\,\} \cup \{\, \langle 2, q\rangle \mid q \in Q_{M_2}\,\}$;
- $s_N =$
- $A_N =$
- $T_N =$

Unions of FAs

Next, we define a function/algorithm $\textbf{union} \in \textbf{FA} \times \textbf{FA} \to \textbf{FA}$ such that $L(\textbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$, for all $M_1, M_2 \in \textbf{FA}$. If $M_1, M_2 \in \textbf{FA}$, then $\textbf{union}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle 1, q \rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q \rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \mathsf{A}$;
- $A_N =$
- $T_N =$

Unions of FAs

Next, we define a function/algorithm $\textbf{union} \in \textbf{FA} \times \textbf{FA} \to \textbf{FA}$ such that $L(\textbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$, for all $M_1, M_2 \in \textbf{FA}$. If $M_1, M_2 \in \textbf{FA}$, then $\textbf{union}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle 1, q \rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q \rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\, \langle 1, q \rangle \mid q \in A_{M_1} \,\} \cup \{\, \langle 2, q \rangle \mid q \in A_{M_2} \,\}$; and
- $T_N =$

Unions of FAs

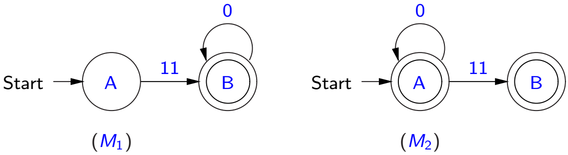
Next, we define a function/algorithm **union** $\in$ **FA** $\times$ **FA** $\to$ **FA** such that $L(\mathbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$, for all $M_1, M_2 \in \mathbf{FA}$. If $M_1, M_2 \in \mathbf{FA}$, then $\mathbf{union}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle 1, q\rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\, \langle 1, q\rangle \mid q \in A_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in A_{M_2} \,\}$; and
- $T_N =$

$$
\begin{aligned}
&\{\mathsf{A}, \% \to \langle 1, s_{M_1}\rangle\} \\
\cup\, &\{\mathsf{A}, \% \to \langle 2, s_{M_2}\rangle\} \\
\cup\, &\{\, \langle 1, q\rangle, a \to \langle 1, r\rangle \mid q, a \to r \in T_{M_1} \,\} \\
\cup\, &\{\, \langle 2, q\rangle, a \to \langle 2, r\rangle \mid q, a \to r \in T_{M_2} \,\}.
\end{aligned}
$$

Union Example

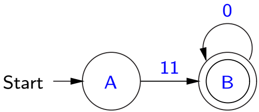
For example, if M_1 and M_2 are the FAs



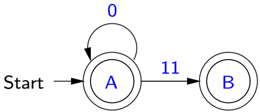
then **union**(M_1, M_2) is the FA

Union Example

For example, if M_1 and M_2 are the FAs

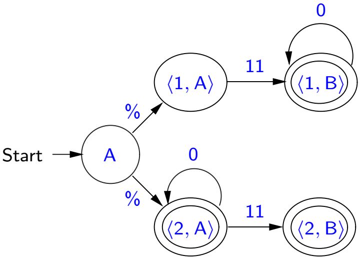


(M_1)



(M_2)

then **union**(M_1, M_2) is the FA



Proposition 3.12.1

For all $M_1, M_2 \in \mathbf{FA}$*:*

- $L(\mathbf{union}(M_1, M_2)) = L(M_1) \cup L(M_2)$; *and*
- $\mathbf{alphabet}(\mathbf{union}(M_1, M_2)) = \mathbf{alphabet}\, M_1 \cup \mathbf{alphabet}\, M_2$.

Proposition 3.12.2

For all $M_1, M_2 \in \mathbf{EFA}$, $\mathbf{union}(M_1, M_2) \in \mathbf{EFA}$.

Concatenations of FAs

Next, we define a function/algorithm $\textbf{concat} \in \textbf{FA} \times \textbf{FA} \to \textbf{FA}$ such that $L(\textbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$, for all $M_1, M_2 \in \textbf{FA}$. If $M_1, M_2 \in \textbf{FA}$, then $\textbf{concat}(M_1, M_2)$ is the FA N such that:

- $Q_N =$
- $s_N =$
- $A_N =$
- $T_N =$

Concatenations of FAs

Next, we define a function/algorithm $\mathbf{concat} \in \mathbf{FA} \times \mathbf{FA} \to \mathbf{FA}$ such that $L(\mathbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$, for all $M_1, M_2 \in \mathbf{FA}$. If $M_1, M_2 \in \mathbf{FA}$, then $\mathbf{concat}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\, \langle 1, q\rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in Q_{M_2} \,\}$;
- $s_N =$
- $A_N =$
- $T_N =$

Concatenations of FAs

Next, we define a function/algorithm $\textbf{concat} \in \textbf{FA} \times \textbf{FA} \to \textbf{FA}$ such that $L(\textbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$, for all $M_1, M_2 \in \textbf{FA}$. If $M_1, M_2 \in \textbf{FA}$, then $\textbf{concat}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\, \langle 1, q\rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q\rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \langle 1, s_{M_1}\rangle$;
- $A_N =$
- $T_N =$

Concatenations of FAs

Next, we define a function/algorithm **concat** $\in$ **FA** $\times$ **FA** $\rightarrow$ **FA** such that $L(\mathbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$, for all $M_1, M_2 \in \mathbf{FA}$.
If $M_1, M_2 \in \mathbf{FA}$, then $\mathbf{concat}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\, \langle 1, q \rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q \rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \langle 1, s_{M_1} \rangle$;
- $A_N = \{\, \langle 2, q \rangle \mid q \in A_{M_2} \,\}$; and
- $T_N =$

Concatenations of FAs

Next, we define a function/algorithm **concat** $\in$ **FA** $\times$ **FA** $\rightarrow$ **FA** such that $L(\textbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$, for all $M_1, M_2 \in \textbf{FA}$.

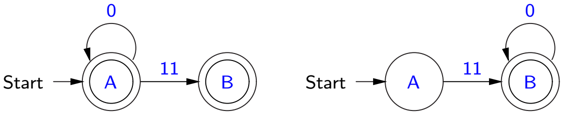
If $M_1, M_2 \in \textbf{FA}$, then $\textbf{concat}(M_1, M_2)$ is the FA N such that:

- $Q_N = \{\, \langle 1, q \rangle \mid q \in Q_{M_1} \,\} \cup \{\, \langle 2, q \rangle \mid q \in Q_{M_2} \,\}$;
- $s_N = \langle 1, s_{M_1} \rangle$;
- $A_N = \{\, \langle 2, q \rangle \mid q \in A_{M_2} \,\}$; and
- $T_N =$

$$\begin{aligned} &\{\, \langle 1, q \rangle, \% \rightarrow \langle 2, s_{M_2} \rangle \mid q \in A_{M_1} \,\} \\ \cup &\{\, \langle 1, q \rangle, a \rightarrow \langle 1, r \rangle \mid q, a \rightarrow r \in T_{M_1} \,\} \\ \cup &\{\, \langle 2, q \rangle, a \rightarrow \langle 2, r \rangle \mid q, a \rightarrow r \in T_{M_2} \,\}. \end{aligned}$$

Concatenation Example

For example, if M_1 and M_2 are the FAs

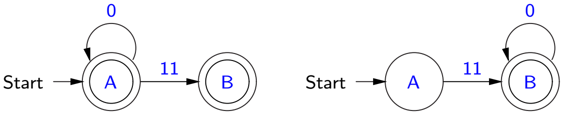


(M_1) (M_2)

then **concat**(M_1, M_2) is the FA

Concatenation Example

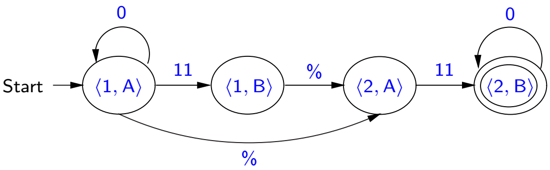
For example, if M_1 and M_2 are the FAs



(M_1)

(M_2)

then **concat**(M_1, M_2) is the FA



Concatenation

Proposition 3.12.3
For all $M_1, M_2 \in \mathbf{FA}$*:*

- $L(\mathbf{concat}(M_1, M_2)) = L(M_1)L(M_2)$*; and*
- $\mathbf{alphabet}(\mathbf{concat}(M_1, M_2)) = \mathbf{alphabet}\, M_1 \cup \mathbf{alphabet}\, M_2$*.*

Proposition 3.12.4
For all $M_1, M_2 \in \mathbf{EFA}$, $\mathbf{concat}(M_1, M_2) \in \mathbf{EFA}$*.*

Closures of FAs

Next, we define a function/algorithm **closure** $\in$ **FA** $\rightarrow$ **FA** such that $L(\textbf{closure}\ M) = L(M)^*$, for all $M \in \textbf{FA}$. If $M \in \textbf{FA}$, then **closure** M is the FA N such that:

- $Q_N =$
- $s_N =$
- $A_N =$
- $T_N =$

Closures of FAs

Next, we define a function/algorithm **closure** $\in$ **FA** $\to$ **FA** such that $L(\textbf{closure}\, M) = L(M)^*$, for all $M \in \textbf{FA}$. If $M \in \textbf{FA}$, then **closure** M is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle q \rangle \mid q \in Q_M \,\}$;
- $s_N =$
- $A_N =$
- $T_N =$

Closures of FAs

Next, we define a function/algorithm **closure** $\in$ **FA** $\to$ **FA** such that $L(\textbf{closure}\, M) = L(M)^*$, for all $M \in \textbf{FA}$. If $M \in \textbf{FA}$, then **closure** M is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle q \rangle \mid q \in Q_M \,\}$;
- $s_N = \mathsf{A}$;
- $A_N =$
- $T_N =$

Closures of FAs

Next, we define a function/algorithm **closure** $\in$ **FA** $\rightarrow$ **FA** such that $L(\textbf{closure}\ M) = L(M)^*$, for all $M \in \textbf{FA}$. If $M \in \textbf{FA}$, then **closure** M is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle q \rangle \mid q \in Q_M \,\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\mathsf{A}\}$; and
- $T_N =$

Closures of FAs

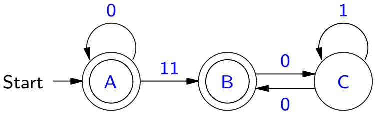
Next, we define a function/algorithm **closure** $\in$ **FA** $\to$ **FA** such that $L(\textbf{closure}\, M) = L(M)^*$, for all $M \in \textbf{FA}$. If $M \in \textbf{FA}$, then **closure** M is the FA N such that:

- $Q_N = \{\mathsf{A}\} \cup \{\, \langle q \rangle \mid q \in Q_M \,\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\mathsf{A}\}$; and
- $T_N =$

$$
\begin{aligned}
&\{\mathsf{A}, \% \to \langle s_M \rangle\} \\
\cup\, &\{\, \langle q \rangle, \% \to \mathsf{A} \mid q \in A_M \,\} \\
\cup\, &\{\, \langle q \rangle, a \to \langle r \rangle \mid q, a \to r \in T_M \,\}.
\end{aligned}
$$

Closure Example

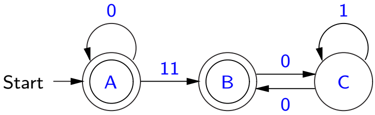
For example, if M is the FA



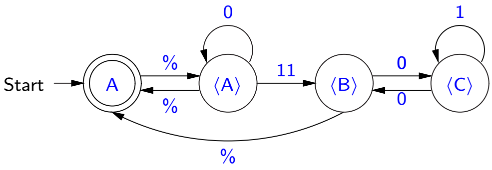
then **closure** M is the FA

Closure Example

For example, if *M* is the FA



then **closure** *M* is the FA



Closure

Proposition 3.12.5
For all $M \in \textbf{FA}$,

- $L(\textbf{closure}\, M) = L(M)^*$; *and*
- $\textbf{alphabet}(\textbf{closure}\, M) = \textbf{alphabet}\, M$.

Proposition 3.12.6
For all $M \in \textbf{EFA}$, $\textbf{closure}\, M \in \textbf{EFA}$.

Conversion Algorithm

We define a function/algorithm $\textbf{regToFA} \in \textbf{Reg} \to \textbf{FA}$ by well-founded recursion on the height of regular expressions, as follows. The goal is for $L(\textbf{regToFA}\,\alpha)$ to be equal to $L(\alpha)$, for all regular expressions α.

- $\textbf{regToFA}\,\% =$
- $\textbf{regToFA}\,\$ =$
- for all $\alpha \in \textbf{Reg}$, $\textbf{regToFA}(\alpha^*) =$
- for all $\alpha, \beta \in \textbf{Reg}$,
  $\textbf{regToFA}(\alpha + \beta) =$

Conversion Algorithm

We define a function/algorithm $\textbf{regToFA} \in \textbf{Reg} \to \textbf{FA}$ by well-founded recursion on the height of regular expressions, as follows. The goal is for $L(\textbf{regToFA}\,\alpha)$ to be equal to $L(\alpha)$, for all regular expressions α.

- $\textbf{regToFA}\,\% = \textbf{emptyStr}$;
- $\textbf{regToFA}\,\$ =$
- for all $\alpha \in \textbf{Reg}$, $\textbf{regToFA}(\alpha^*) =$
- for all $\alpha, \beta \in \textbf{Reg}$,
  $\textbf{regToFA}(\alpha + \beta) =$

Conversion Algorithm

We define a function/algorithm **regToFA** $\in$ **Reg** $\to$ **FA** by well-founded recursion on the height of regular expressions, as follows. The goal is for L(**regToFA** α) to be equal to $L(\alpha)$, for all regular expressions α.

- **regToFA** % = **emptyStr**;
- **regToFA** \$ = **emptySet**;
- for all $\alpha \in$ **Reg**, **regToFA**$(\alpha^*) =$
- for all $\alpha, \beta \in$ **Reg**,
  regToFA$(\alpha + \beta) =$

Conversion Algorithm

We define a function/algorithm **regToFA** $\in$ **Reg** $\to$ **FA** by well-founded recursion on the height of regular expressions, as follows. The goal is for L(**regToFA** α) to be equal to $L(\alpha)$, for all regular expressions α.

- **regToFA** % = **emptyStr**;
- **regToFA** \$ = **emptySet**;
- for all $\alpha \in$ **Reg**, **regToFA**(α^*) = **closure**(**regToFA** α);
- for all $\alpha, \beta \in$ **Reg**,
  regToFA($\alpha + \beta$) =

Conversion Algorithm

We define a function/algorithm **regToFA** $\in$ **Reg** $\to$ **FA** by well-founded recursion on the height of regular expressions, as follows. The goal is for L(**regToFA** α) to be equal to $L(\alpha)$, for all regular expressions α.

- **regToFA** % = **emptyStr**;
- **regToFA** \$ = **emptySet**;
- for all $\alpha \in$ **Reg**, **regToFA**(α^*) = **closure**(**regToFA** α);
- for all $\alpha, \beta \in$ **Reg**,
  regToFA$(\alpha + \beta)$ = **union**(**regToFA** α, **regToFA** β);

Conversion Algorithm

- for all $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in$ **Sym**, **regToFA**$(a_1 \cdots a_n) =$
- for all $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in$ **Sym** and $\alpha \in$ **Reg**, if α doesn't consist of a single symbol, and doesn't have the form $b\,\beta$ for some $b \in$ **Sym** and $\beta \in$ **Reg**, then **regToFA**$(a_1 \cdots a_n\,\alpha) =$
- for all $\alpha, \beta \in$ **Reg**, if α doesn't consist of a single symbol, then **regToFA**$(\alpha\beta) =$

For example, we have that
regToFA$(0101^*) =$

Conversion Algorithm

- for all $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in$ **Sym**,
  regToFA$(a_1 \cdots a_n) =$ **strToFA**$(a_1 \cdots a_n)$;
- for all $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in$ **Sym** and $\alpha \in$ **Reg**, if α doesn't consist of a single symbol, and doesn't have the form $b\,\beta$ for some $b \in$ **Sym** and $\beta \in$ **Reg**, then
  regToFA$(a_1 \cdots a_n\,\alpha) =$
- for all $\alpha, \beta \in$ **Reg**, if α doesn't consist of a single symbol, then **regToFA**$(\alpha\beta) =$

For example, we have that
regToFA$(0101^*) =$

Conversion Algorithm

- for all $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in \textbf{Sym}$, $\textbf{regToFA}(a_1 \cdots a_n) = \textbf{strToFA}(a_1 \cdots a_n)$;
- for all $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in \textbf{Sym}$ and $\alpha \in \textbf{Reg}$, if α doesn't consist of a single symbol, and doesn't have the form $b\,\beta$ for some $b \in \textbf{Sym}$ and $\beta \in \textbf{Reg}$, then $\textbf{regToFA}(a_1 \cdots a_n\,\alpha) = \textbf{concat}(\textbf{strToFA}(a_1 \cdots a_n), \textbf{regToFA}\,\alpha)$; and
- for all $\alpha, \beta \in \textbf{Reg}$, if α doesn't consist of a single symbol, then $\textbf{regToFA}(\alpha\beta) =$

For example, we have that
$\textbf{regToFA}(0101^*) =$

Conversion Algorithm

- for all $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in$ **Sym**, **regToFA**$(a_1 \cdots a_n) =$ **strToFA**$(a_1 \cdots a_n)$;
- for all $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in$ **Sym** and $\alpha \in$ **Reg**, if α doesn't consist of a single symbol, and doesn't have the form $b\,\beta$ for some $b \in$ **Sym** and $\beta \in$ **Reg**, then **regToFA**$(a_1 \cdots a_n\,\alpha) =$ **concat**(**strToFA**$(a_1 \cdots a_n)$, **regToFA** α); and
- for all $\alpha, \beta \in$ **Reg**, if α doesn't consist of a single symbol, then **regToFA**$(\alpha\beta) =$ **concat**(**regToFA** α, **regToFA** β).

For example, we have that
regToFA$(0101^*) =$

Conversion Algorithm

- for all $n \in \mathbb{N} - \{0\}$ and $a_1, \ldots, a_n \in \textbf{Sym}$, $\textbf{regToFA}(a_1 \cdots a_n) = \textbf{strToFA}(a_1 \cdots a_n)$;
- for all $n \in \mathbb{N} - \{0\}$, $a_1, \ldots, a_n \in \textbf{Sym}$ and $\alpha \in \textbf{Reg}$, if α doesn't consist of a single symbol, and doesn't have the form $b\,\beta$ for some $b \in \textbf{Sym}$ and $\beta \in \textbf{Reg}$, then $\textbf{regToFA}(a_1 \cdots a_n\,\alpha) = \textbf{concat}(\textbf{strToFA}(a_1 \cdots a_n), \textbf{regToFA}\,\alpha)$; and
- for all $\alpha, \beta \in \textbf{Reg}$, if α doesn't consist of a single symbol, then $\textbf{regToFA}(\alpha\beta) = \textbf{concat}(\textbf{regToFA}\,\alpha, \textbf{regToFA}\,\beta)$.

For example, we have that $\textbf{regToFA}(0101^*) = \textbf{concat}(\textbf{strToFA}(010), \textbf{regToFA}(1^*))$.

Specification of **regToFA**

Theorem 3.12.7
For all $\alpha \in \mathbf{Reg}$:

- $L(\mathbf{regToFA}\,\alpha) = L(\alpha)$; *and*
- $\mathbf{alphabet}(\mathbf{regToFA}\,\alpha) = \mathbf{alphabet}\,\alpha$.

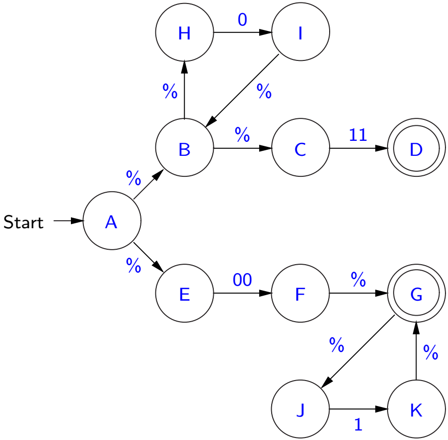
Proof. Because of the form of recursion used, the proof uses well-founded induction on the height of α. $\square$

Example Conversion

For example, **regToFA**$(0^*11 + 001^*)$ is isomorphic to the FA

Example Conversion

For example, **regToFA**$(0^*11 + 001^*)$ is isomorphic to the FA



Building FAs in Forlan

The Forlan module `FA` includes these constants and functions for building finite automata and converting regular expressions to finite automata:

```
val emptyStr : fa
val emptySet : fa
val fromStr  : str -> fa
val fromSym  : sym -> fa
val union    : fa * fa -> fa
val concat   : fa * fa -> fa
val closure  : fa -> fa
val fromReg  : reg -> fa
```

The functions `fromStr` and `fromSym` correspond to **strToFA** and **symToNFA**, and are also available in the top-level environment with the names

```
val strToFA : str -> fa
val symToFA : sym -> fa
```

Building FAs in Forlan

The function `fromReg` corresponds to **regToFA** and is available in the top-level environment with that name:

```
val regToFA : reg -> fa
```

The constants `emptyStr` and `emptySet` are inherited by the modules `DFA`, `NFA` and `EFA`.

The function `fromSym` is inherited by the modules `NFA` and `EFA`, and is available in the top-level environment with the names

```
val symToNFA : sym -> nfa
val symToEFA : sym -> efa
```

The functions `union`, `concat` and `closure` are inherited by the module `EFA`.

Forlan Example

Here is how the regular expression $0^*11 + 001^*$ can be converted to an FA in Forlan:

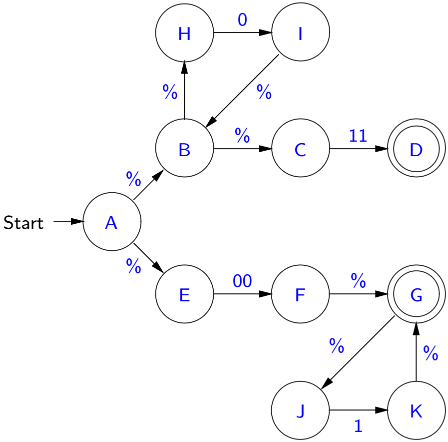
```
- val reg = Reg.input "";
@ 0*11 + 001*
@ .
val reg = - : reg
- val fa = regToFA reg;
val fa = - : fa
- val fa' = FA.renameStatesCanonically fa;
val fa' = - : fa
```

Forlan Example

```
- FA.output("", fa');
{states} A, B, C, D, E, F, G, H, I, J, K
{start state} A {accepting states} D, G
{transitions}
A, % -> B | E; B, % -> C | H; C, 11 -> D; E, 00 -> F;
F, % -> G; G, % -> J; H, 0 -> I; I, % -> B; J, 1 -> K;
K, % -> G
val it = () : unit
```

Forlan Example

Thus `fa'` is the finite automaton



Converting FAs to Regular Expressions

Our algorithm for converting FAs to regular expressions makes use of a more general kind of finite automata that we call regular expression finite automata.

Regular Expression Finite Automata

A *regular expression finite automaton* (RFA) M consists of:

- a finite set Q_M of symbols;
- an element s_M of Q_M;
- a subset A_M of Q_M; and
- a finite subset T_M of $\{\, (q, \quad, r) \mid q, r \in Q_M$ and $\quad \in \qquad \}$

We write **RFA** for the set of all RFAs, which is a countably infinite set.

RFAs are drawn analogously to FAs, and the Forlan syntax for RFAs is analogous to that of FAs.

Regular Expression Finite Automata

A *regular expression finite automaton* (RFA) M consists of:

- a finite set Q_M of symbols;
- an element s_M of Q_M;
- a subset A_M of Q_M; and
- a finite subset T_M of $\{\, (q, \alpha, r) \mid q, r \in Q_M \text{ and } \alpha \in \mathbf{Reg} \,\}$

We write **RFA** for the set of all RFAs, which is a countably infinite set.

RFAs are drawn analogously to FAs, and the Forlan syntax for RFAs is analogous to that of FAs.

Regular Expression Finite Automata

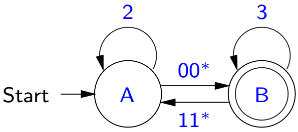
A *regular expression finite automaton* (RFA) M consists of:

- a finite set Q_M of symbols;
- an element s_M of Q_M;
- a subset A_M of Q_M; and
- a finite subset T_M of $\{ (q, \alpha, r) \mid q, r \in Q_M \text{ and } \alpha \in \mathbf{Reg} \}$ such that, for all $q, r \in Q_M$, there is at most one $\alpha \in \mathbf{Reg}$ such that $(q, \alpha, r) \in T_M$.

We write **RFA** for the set of all RFAs, which is a countably infinite set.

RFAs are drawn analogously to FAs, and the Forlan syntax for RFAs is analogous to that of FAs.

For example, the RFA M whose states are A and B, start state is A, only accepting state is B, and transitions are $(\mathsf{A}, 2, \mathsf{A})$, $(\mathsf{A}, 00^*, \mathsf{B})$, $(\mathsf{B}, 3, \mathsf{B})$ and $(\mathsf{B}, 11^*, \mathsf{A})$ can be drawn as



and expressed in Forlan as

```
{states} A, B {start state} A {accepting states} B
{transitions}
A, 2 -> A; A, 00* -> B; B, 3 -> B; B, 11* -> A
```

More on RFAs

The *alphabet of* an RFA M (**alphabet** M) is $\{\, a \in \textbf{Sym} \mid$ there are q, α, r such that $q, \alpha \to r \in T_M$ and $a \in \textbf{alphabet}\, \alpha \,\}$.

For example, the alphabet of our example FA M is $\{0, 1, 2, 3\}$.

More on RFAs

The *alphabet of* an RFA M (**alphabet** M) is $\{\, a \in \textbf{Sym} \mid$ there are q, α, r such that $q, \alpha \rightarrow r \in T_M$ and $a \in \textbf{alphabet}\, \alpha \,\}$.

For example, the alphabet of our example FA M is $\{0, 1, 2, 3\}$.

The Forlan module `RFA` defines an abstract type `rfa` (in the top-level environment) of regular expression finite automata, as well as some functions for processing RFAs including:

```
val input          : string -> rfa
val output         : string * rfa -> unit
val alphabet       : rfa -> sym set
val numStates      : rfa -> int
val numTransitions : rfa -> int
val equal          : rfa * rfa -> bool
```

Graphical Editor for RFAs

The Java program JForlan, can be used to view and edit regular expression finite automata. It can be invoked directly, or run via Forlan. See the Forlan website for more information.

Validity of Labeled Paths in RFAs

A labeled path

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots \; q_n \overset{x_n}{\Rightarrow} q_{n+1},$$

is *valid for* an RFA M iff, for all $i \in [1:n]$,

$$x_i \in L(\alpha), \text{ for some } \alpha \in \textbf{Reg} \text{ such that } q_i, \alpha \to q_{i+1},$$

and $q_{n+1} \in Q_M$.

Validity of Labeled Paths in RFAs

A labeled path

$$q_1 \overset{x_1}{\Rightarrow} q_2 \overset{x_2}{\Rightarrow} \cdots\ q_n \overset{x_n}{\Rightarrow} q_{n+1},$$

is *valid for* an RFA M iff, for all $i \in [1:n]$,

$$x_i \in L(\alpha), \text{ for some } \alpha \in \textbf{Reg} \text{ such that } q_i, \alpha \rightarrow q_{i+1},$$

and $q_{n+1} \in Q_M$.

For example, the labeled path

$$\mathsf{A} \overset{000}{\Rightarrow} \mathsf{B} \overset{3}{\Rightarrow} \mathsf{B}$$

is valid for our example FA M, because

- $000 \in L(00^*)$ and $\mathsf{A}, 00^* \rightarrow \mathsf{B} \in T$, and
- $3 \in L(3)$ and $\mathsf{B}, 3 \rightarrow \mathsf{B} \in T$.

The Meaning of RFAs

A string w is *accepted by* an RFA M iff there is a labeled path lp such that

- the label of lp is w;
- lp is valid for M;
- the start state of lp is the start state of M; and
- the end state of lp is an accepting state of M.

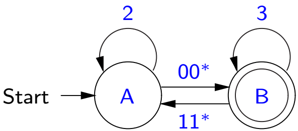
We have that, if w is accepted by M, then $\mathbf{alphabet}\, w \subseteq \mathbf{alphabet}\, M$.

The *language accepted by* an RFA M ($L(M)$) is

$$\{\, w \in \mathbf{Str} \mid w \text{ is accepted by } M \,\}.$$

RFA Meaning Example

Consider our example RFA M:



We have that 20 and 0000111103 are accepted by M, but that 23 and 122 are not accepted by M.

A Function for Combining Transitions

We define a function **combineTrans** that takes in a pair (simp, U) such that

- $\mathit{simp} \in \mathbf{Reg} \to \mathbf{Reg}$ and
- U is a finite subset of $\{\, p, \alpha \to q \mid p, q \in \mathbf{Sym}$ and $\alpha \in \mathbf{Reg} \,\}$,

and returns a finite subset V of $\{\, p, \alpha \to q \mid p, q \in \mathbf{Sym}$ and $\alpha \in \mathbf{Reg} \,\}$ with the property that, for all $p, q \in \mathbf{Sym}$, there is at most one β such that $p, \beta \to q \in V$.

A Function for Combining Transitions

We define a function **combineTrans** that takes in a pair (simp, U) such that

- $\mathit{simp} \in \mathbf{Reg} \to \mathbf{Reg}$ and
- U is a finite subset of $\{\, p, \alpha \to q \mid p, q \in \mathbf{Sym} \text{ and } \alpha \in \mathbf{Reg} \,\}$,

and returns a finite subset V of $\{\, p, \alpha \to q \mid p, q \in \mathbf{Sym} \text{ and } \alpha \in \mathbf{Reg} \,\}$ with the property that, for all $p, q \in \mathbf{Sym}$, there is at most one β such that $p, \beta \to q \in V$.

Given such a pair (simp, U), **combineTrans** returns the set of all transitions $p, \alpha \to q$ such that $\{\, \beta \mid p, \beta \to q \in U \,\}$ is nonempty, and $\alpha = \mathit{simp}(\beta_1 + \cdots + \beta_n)$, where $\beta_1, \ldots, \beta_n$ are all of the elements of this set, listed in increasing order and without repetition.

Converting FAs to RFAs

We define a function/algorithm

$$\textbf{faToRFA} \in (\textbf{Reg} \to \textbf{Reg}) \to \textbf{FA} \to \textbf{RFA}.$$

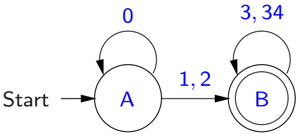
faToRFA takes in $simp \in \textbf{Reg} \to \textbf{Reg}$, and returns a function that takes in $M \in \textbf{FA}$, and returns the RFA N such that:

- $Q_N = Q_M$;
- $s_N = s_M$;
- $A_N = A_M$; and
- $T_N =$

$$\textbf{combineTrans}(simp, \{\, p, \textbf{strToReg}\, x \to q \mid p, x \to q \in T_M \,\}).$$

FA to RFA Example

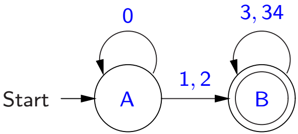
For example, if the FA *M* is



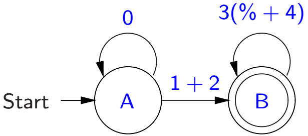
and the simplification function *simp* is **locallySimplify obviousSubset** then **faToRFA** *simp M* is the RFA

FA to RFA Example

For example, if the FA *M* is



and the simplification function *simp* is **locallySimplify obviousSubset** then **faToRFA** *simp M* is the RFA



Specification of **faToRFA**

Proposition 3.12.8

Suppose $\mathit{simp} \in \textbf{Reg} \to \textbf{Reg}$ *and* $M \in \textbf{FA}$*. If, for all* $\alpha \in \textbf{Reg}$*,* $L(\mathit{simp}\,\alpha) = L(\alpha)$ *and* $\textbf{alphabet}(\mathit{simp}\,\alpha) \subseteq \textbf{alphabet}\,\alpha$*, then*

(1) $L(\textbf{faToRFA}\,\mathit{simp}\,M) = L(M)$*, and*

(2) $\textbf{alphabet}(\textbf{faToRFA}\,\mathit{simp}\,M) = \textbf{alphabet}\,M$*.*

Converting FAs to RFAs in Forlan

The `RFA` module has a function

```
val fromRFA : (reg -> reg) -> fa -> rfa
```

that corresponds to **faToRFA**.

Converting FAs to RFAs in Forlan

Here is how our conversion example can be carried out in Forlan:

```
- val simp =
=       #2 o
=       Reg.locallySimplify(NONE, Reg.obviousSubset);
val simp = fn : reg -> reg
- val fa = FA.input "";
@ {states} A, B {start state} A {accepting states} B
@ {transitions}
@ A, 0 -> A; A, 1 -> B; A, 2 -> B;
@ B, 3 -> B; B, 34 -> B
@ .
val fa = - : fa
```

Converting FAs to RFAs in Forlan

```
- val rfa = RFA.fromFA simp fa;
val rfa = - : rfa
- RFA.output("", rfa);
{states} A, B {start state} A {accepting states} B
{transitions}
A, 0 -> A; A, 1 + 2 -> B; B, 3(% + 4) -> B
val it = () : unit
```

Standard FAs and RFAs

We say that an RFA M is *standard* iff

- M's start state is not an accepting state, and there are no transitions *into* M's start state (even from s_M to itself); and
- M has a single accepting state, and there are no transitions *from* that state (even from the accepting state to itself).

We say that an RFA M is *standard* iff

- M's start state is not an accepting state, and there are no transitions *into* M's start state (even from s_M to itself); and
- M has a single accepting state, and there are no transitions *from* that state (even from the accepting state to itself).

Proposition 3.12.9

Suppose M is a standard RFA with only two states, and that q is M's accepting state.

- *For all $\alpha \in \mathbf{Reg}$, if $s_M, \alpha \to q$, then $L(M) =$*
- *If there is no $\alpha \in \mathbf{Reg}$ such that $s_M, \alpha \to q$, then $L(M) =$*

Standard FAs and RFAs

We say that an RFA M is *standard* iff

- M's start state is not an accepting state, and there are no transitions *into* M's start state (even from s_M to itself); and
- M has a single accepting state, and there are no transitions *from* that state (even from the accepting state to itself).

Proposition 3.12.9

Suppose M is a standard RFA with only two states, and that q is M's accepting state.

- *For all $\alpha \in \mathbf{Reg}$, if $s_M, \alpha \to q$, then $L(M) = L(\alpha)$.*
- *If there is no $\alpha \in \mathbf{Reg}$ such that $s_M, \alpha \to q$, then $L(M) =$*

Standard FAs and RFAs

We say that an RFA M is *standard* iff

- M's start state is not an accepting state, and there are no transitions *into* M's start state (even from s_M to itself); and
- M has a single accepting state, and there are no transitions *from* that state (even from the accepting state to itself).

Proposition 3.12.9

Suppose M is a standard RFA with only two states, and that q is M's accepting state.

- *For all $\alpha \in \mathbf{Reg}$, if $s_M, \alpha \to q$, then $L(M) = L(\alpha)$.*
- *If there is no $\alpha \in \mathbf{Reg}$ such that $s_M, \alpha \to q$, then $L(M) = \emptyset$.*

Standardization

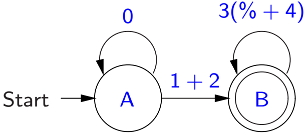
We define a function **standardize** $\in$ **RFA** $\rightarrow$ **RFA** that standardizes an RFA, as follows. Given an argument M, it returns the RFA N such that:

- $Q_N = \{ \langle q \rangle \mid q \in Q_M \} \cup \{\mathsf{A}, \mathsf{B}\}$;
- $s_N = \mathsf{A}$;
- $A_N = \{\mathsf{B}\}$; and
- T_N

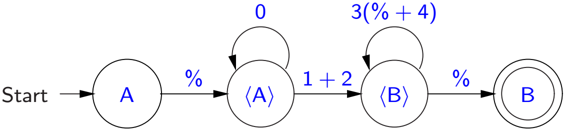
$$
\begin{aligned}
&= \{\mathsf{A}, \% \rightarrow \langle s_M \rangle\} \\
&\cup \ \{ \langle q \rangle, \% \rightarrow \mathsf{B} \mid q \in A_M \} \\
&\cup \ \{ \langle q \rangle, \alpha \rightarrow \langle r \rangle \mid q, \alpha \rightarrow r \in T_M \}.
\end{aligned}
$$

Standardization

For example, if M is the RFA



then **standardize** M is the RFA



Proposition 3.12.10

Suppose M is an RFA. Then:

- **standardize** M *is standard;*
- $L(\textbf{standardize}\ M) = L(M)$*; and*
- $\textbf{alphabet}(\textbf{standardize}\ M) = \textbf{alphabet}\ M$.

Eliminating a State of an RFA

Next, we define a function **eliminateState** that takes in a function $simp \in \mathbf{Reg} \to \mathbf{Reg}$, and returns a function that takes in a pair (M, q), where M is an RFA and $q \in Q_M - (\{s_M\} \cup A_M)$, and then returns an RFA. When called with such a $simp$ and (M, q), **eliminateState** returns the RFA N such that:

- $Q_N = Q_M - \{q\}$;
- $s_N = s_M$;
- $A_N = A_M$; and

Eliminating a State of an RFA

Next, we define a function **eliminateState** that takes in a function $simp \in \mathbf{Reg} \to \mathbf{Reg}$, and returns a function that takes in a pair (M, q), where M is an RFA and $q \in Q_M - (\{s_M\} \cup A_M)$, and then returns an RFA. When called with such a $simp$ and (M, q), **eliminateState** returns the RFA N such that:

- $Q_N = Q_M - \{q\}$;
- $s_N = s_M$;
- $A_N = A_M$; and
- $T_N = \mathbf{combineTrans}(simp, U \cup V)$, where
  - $U = \{\, p, \alpha \to r \in T_M \mid p \neq q$ and $r \neq q \,\}$,

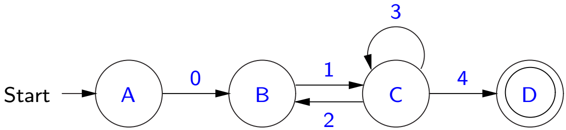
Eliminating a State of an RFA

Next, we define a function **eliminateState** that takes in a function $simp \in \mathbf{Reg} \rightarrow \mathbf{Reg}$, and returns a function that takes in a pair (M, q), where M is an RFA and $q \in Q_M - (\{s_M\} \cup A_M)$, and then returns an RFA. When called with such a $simp$ and (M, q), **eliminateState** returns the RFA N such that:

- $Q_N = Q_M - \{q\}$;
- $s_N = s_M$;
- $A_N = A_M$; and
- $T_N = \mathbf{combineTrans}(simp, U \cup V)$, where
  - $U = \{\, p, \alpha \rightarrow r \in T_M \mid p \neq q$ and $r \neq q \,\}$,
  - $V = \{\, p, simp(\alpha\beta^*\gamma) \rightarrow r \mid p \neq q, r \neq q, p, \alpha \rightarrow q \in T_M$ and $q, \gamma \rightarrow r \in T_M \,\}$, and
  - β is the unique $\alpha \in \mathbf{Reg}$ such that $q, \alpha \rightarrow q \in T_M$, if such an α exists, and is $\%$, otherwise.

Eliminating a State Example

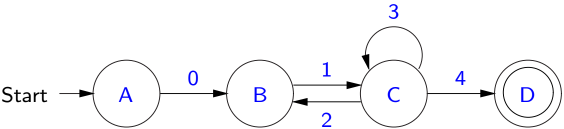
Suppose *simp* is **locallySimplify obviousSubset**



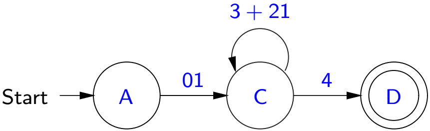
Then **eliminateState** *simp* (M, B) is

Eliminating a State Example

Suppose *simp* is **locallySimplify obviousSubset**



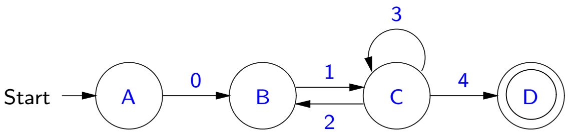
Then **eliminateState** *simp* (M, B) is



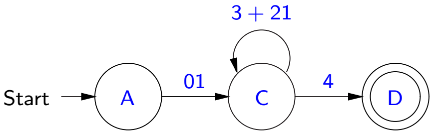
And, we can eliminate C from this RFA, yielding

Eliminating a State Example

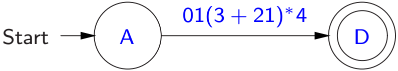
Suppose *simp* is **locallySimplify obviousSubset**



Then **eliminateState** *simp* (M, B) is

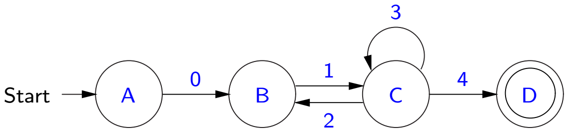


And, we can eliminate C from this RFA, yielding



Eliminating a State Example

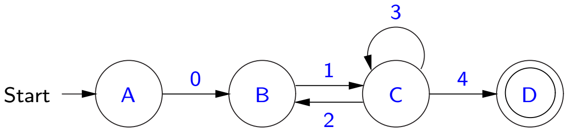
Alternatively, we could eliminate C from



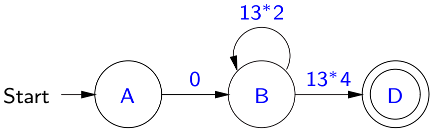
yielding

Eliminating a State Example

Alternatively, we could eliminate C from



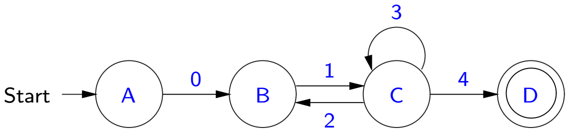
yielding



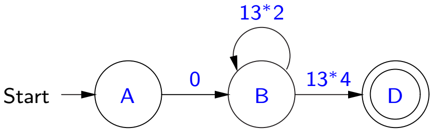
And could then eliminate B from this RFA, yielding

Eliminating a State Example

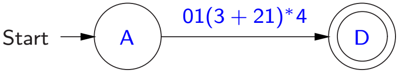
Alternatively, we could eliminate C from



yielding



And could then eliminate B from this RFA, yielding



($simp(0(13^*2)^*(13^*4)) = 01(3+21)^*4$.)

Eliminating a State Example

Instead of eliminating first C and then B, we could have renamed M's states using the bijection

$$\{(\mathsf{A},\mathsf{A}),(\mathsf{B},\mathsf{C}),(\mathsf{C},\mathsf{B}),(\mathsf{D},\mathsf{D})\}$$

and then have eliminated states in ascending order, according to our usual ordering on symbols: first B and then C.

This is the approach we'll use when looking for alternative answers.

Properties of **eliminateState**

Proposition 3.12.11

Suppose $simp \in \textbf{Reg} \to \textbf{Reg}$, M *is an RFA and* $q \in Q_M - (\{s_M\} \cup A_M)$.

Then:

(1) $\textbf{eliminateState}\, simp\, (M, q)$ *has one less state than* M.

(2) *If* M *is standard, then* $\textbf{eliminateState}\, simp\, (M, q)$ *is standard.*

(3) *If, for all* $\alpha \in \textbf{Reg}$, $L(simp\, \alpha) = L(\alpha)$, *then* $L(\textbf{eliminateState}\, simp\, (M, q)) = L(M)$.

(4) *If, for all* $\alpha \in \textbf{Reg}$, $\textbf{alphabet}(simp\, \alpha) \subseteq \textbf{alphabet}\, \alpha$, *then* $\textbf{alphabet}(\textbf{eliminateState}\, simp\, (M, q)) \subseteq \textbf{alphabet}\, M$.

Eliminating States in Forlan

The RFA module has a function

```
val eliminateState : (reg -> reg) -> rfa * sym -> rfa
```

that corresponds to **eliminateState**.

Eliminating States in Forlan

Here is how our state-elimination examples can be carried out in Forlan:

```
- val rfa = RFA.input "";
@ {states} A, B, C, D {start state} A
@ {accepting states} D
@ {transitions}
@ A, 0 -> B; B, 1 -> C; C, 2 -> B; C, 3 -> C;
@ C, 4 -> D
@ .
val rfa = - : rfa
- val simp =
=       #2 o
=       Reg.locallySimplify(NONE, Reg.obviousSubset);
val simp = fn : reg -> reg
- val eliminateState = RFA.eliminateState simp;
val eliminateState = fn : rfa * sym -> rfa
```

Eliminating States in Forlan

```
- val rfa' = eliminateState(rfa, Sym.fromString "B");
val rfa' = - : rfa
- RFA.output("", rfa');
{states} A, C, D {start state} A {accepting states} D
{transitions} A, 01 -> C; C, 4 -> D; C, 3 + 21 -> C
val it = () : unit
- val rfa'' =
=       eliminateState(rfa', Sym.fromString "C");
val rfa'' = - : rfa
- RFA.output("", rfa'');
{states} A, D {start state} A {accepting states} D
{transitions} A, 01(3 + 21)*4 -> D
val it = () : unit
```

Eliminating States in Forlan

```
- val rfa''' =
=       eliminateState(rfa, Sym.fromString "C");
val rfa''' = - : rfa
- RFA.output("", rfa''');
{states} A, B, D {start state} A {accepting states} D
{transitions} A, 0 -> B; B, 13*2 -> B; B, 13*4 -> D
val it = () : unit
- val rfa'''' =
=       eliminateState(rfa''', Sym.fromString "B");
val rfa'''' = - : rfa
- RFA.output("", rfa'''');
{states} A, D {start state} A {accepting states} D
{transitions} A, 01(3 + 21)*4 -> D
val it = () : unit
```

Eliminating States in Forlan

And `eliminateState` stops us from eliminating a start state or an accepting state:

```
- eliminateState(rfa, Sym.fromString "A");
cannot eliminate start state: "A"

uncaught exception Error
- eliminateState(rfa, Sym.fromString "D");
cannot eliminate accepting state: "D"

uncaught exception Error
```

Conversion Algorithm

Now, we use **eliminateState** to define a function/algorithm

$$\mathbf{rfaToReg} \in (\mathbf{Reg} \to \mathbf{Reg}) \to \mathbf{RFA} \to \mathbf{Reg}.$$

It takes elements $simp \in \mathbf{Reg} \to \mathbf{Reg}$ and $M \in \mathbf{RFA}$, and returns

$$f\,(\mathbf{standardize}\,M),$$

where f is the function from standard RFAs to regular expressions that is defined by well-founded recursion on the number of states of its input, M, as follows:

- If M has only two states, then f returns the label of the transition from s_M to M's accepting state, if such a transition exists, and returns $\$$, otherwise.
- Otherwise, f calls itself recursively on $\mathbf{eliminateState}\,simp\,(M, q)$, where q is the least element (in the standard ordering on symbols) of $Q_M - (\{s_M\} \cup A_M)$.

Proposition 3.12.12

Suppose M *is an RFA and* $simp \in \mathbf{Reg} \to \mathbf{Reg}$ *has the property that, for all* $\alpha \in \mathbf{Reg}$, $L(simp\,\alpha) = L(\alpha)$ *and* $\mathbf{alphabet}(simp\,\alpha) \subseteq \mathbf{alphabet}\,\alpha$. *Then:*

(1) $L(\mathbf{rfaToReg}\,simp\,M) = L(M)$*; and*

(2) $\mathbf{alphabet}(\mathbf{rfaToReg}\,simp\,M) \subseteq \mathbf{alphabet}\,M$.

Conversion Algorithm

Finally, we define our RFA to regular expression conversion algorithm/function:

$$\mathbf{faToReg} \in (\mathbf{Reg} \to \mathbf{Reg}) \to \mathbf{FA} \to \mathbf{Reg}.$$

$\mathbf{faToReg}$ takes in $simp \in \mathbf{Reg} \to \mathbf{Reg}$, and returns

$$\mathbf{rfaToReg}\, simp \circ \mathbf{faToRFA}\, simp.$$

Proposition 3.12.13

Suppose M is an FA and $simp \in \mathbf{Reg} \to \mathbf{Reg}$ has the property that, for all $\alpha \in \mathbf{Reg}$, $L(simp\,\alpha) = L(\alpha)$ and $\mathbf{alphabet}(simp\,\alpha) \subseteq \mathbf{alphabet}\,\alpha$. Then:

(1) $L(\mathbf{faToReg}\, simp\, M) = L(M)$; *and*

(2) $\mathbf{alphabet}(\mathbf{faToReg}\, simp\, M) \subseteq \mathbf{alphabet}\, M$.

Converting FAs to Regular Expressions in Forlan

The Forlan module RFA includes functions

```
val faToReg              : (reg -> reg) -> fa -> reg
val faToRegPerms         :
      int option * (reg -> reg) -> fa -> reg
val faToRegPermsTrace :
      int option * (reg -> reg) -> fa -> reg
```

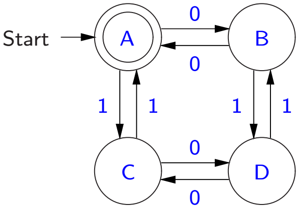
which are available in the top-level environment as

```
val faToReg              : (reg -> reg) -> fa -> reg
val faToRegPerms         :
      int option * (reg -> reg) -> fa -> reg
val faToRegPermsTrace :
      int option * (reg -> reg) -> fa -> reg
```

faToRegPerms tries faToReg on a specified number of permutations of the states of an FA, picking the simplest result.

Converting FAs to Regular Expressions in Forlan

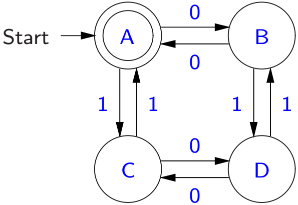
Suppose `fa` is the FA



which accepts

Converting FAs to Regular Expressions in Forlan

Suppose `fa` is the FA



which accepts $\{\, w \in \{0,1\}^* \mid w$ has an even number of 0 and 1's $\}$.

Converting `fa` into a regular expression using `faToReg` and **weaklySimplify** yields a fairly complicated answer:

```
- val reg = faToReg Reg.weaklySimplify fa;
val reg = - : reg
- Reg.output("", reg);
% + 00(00)* +
(1 + 00(00)*1)(11 + 100(00)*1)*(1 + 100(00)*) +
(0(00)*1 +
 (1 + 00(00)*1)(11 + 100(00)*1)*(0 + 10(00)*1))
(1(00)*1 +
 (0 + 10(00)*1)(11 + 100(00)*1)*(0 + 10(00)*1))*
(10(00)* +
 (0 + 10(00)*1)(11 + 100(00)*1)*(1 + 100(00)*))
val it = () : unit
```

Converting FAs to Regular Expressions in Forlan

But by using `faToRegPerms`, we can do much better:

```
- val reg' =
=       faToRegPerms (NONE, Reg.weaklySimplify) fa;
val reg' = - : reg
- Reg.output("", reg');
(00 + 11 + (01 + 10)(00 + 11)*(01 + 10))*
val it = () : unit
```

But by using faToRegPerms, we can do much better:

```
- val reg' =
=       faToRegPerms (NONE, Reg.weaklySimplify) fa;
val reg' = - : reg
- Reg.output("", reg');
(00 + 11 + (01 + 10)(00 + 11)*(01 + 10))*
val it = () : unit
```

By using faToRegPermsTrace, we can learn that this answer was found using the renaming

$$(\mathsf{A},\mathsf{D}),(\mathsf{B},\mathsf{A}),(\mathsf{C},\mathsf{B}),(\mathsf{D},\mathsf{C})$$

of M's states.

That is, it was found by making M into a standard RFA, with new start and accepting states, and then eliminating the states corresponding to B, C, D and A, in that order.

Regular Languages

Since we have algorithms for converting back and forth between regular expressions and finite automata, as well as algorithms for converting FAs to EFAs, EFAs to NFAs, and NFAs to DFAs, we have the following theorem:

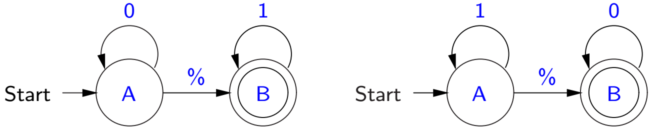
Theorem 3.12.14

Suppose L is a language. The following statements are equivalent:

- *L is regular;*
- *L is generated by a regular expression;*
- *L is accepted by a finite automaton;*
- *L is accepted by an EFA;*
- *L is accepted by an NFA; and*
- *L is accepted by a DFA.*

Intersections of EFAs

Consider the EFAs M_1 and M_2:

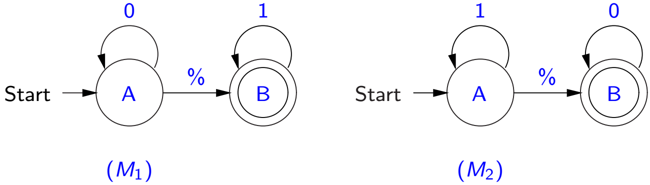


(M_1) (M_2)

How can we construct an EFA N such that
$L(N) = L(M_1) \cap L(M_2)$?

Intersections of EFAs

Consider the EFAs M_1 and M_2:



How can we construct an EFA N such that $L(N) = L(M_1) \cap L(M_2)$?

The idea is to make the states of N represent pairs of the form (q, r), where $q \in Q_{M_1}$ and $r \in Q_{M_2}$.

Auxiliary Functions for Intersection

In order to define our intersection operation on EFAs, we first need to define two auxiliary functions. Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextSym}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \times \mathbf{Sym} \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextSym}_{M_1,M_2}((q,r),a) =$

$$\{\,(q',r') \mid q,a \to q' \in T_{M_1} \text{ and } r,a \to r' \in T_{M_2}\,\}.$$

Auxiliary Functions for Intersection

In order to define our intersection operation on EFAs, we first need to define two auxiliary functions. Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextSym}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \times \mathbf{Sym} \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextSym}_{M_1,M_2}((q,r),a) =$

$$\{\,(q',r') \mid q,a \to q' \in T_{M_1} \text{ and } r,a \to r' \in T_{M_2}\,\}.$$

If M_1 and M_2 are our example EFAs, then

- $\mathbf{nextSym}((\mathrm{A},\mathrm{A}),0) =$
- $\mathbf{nextSym}((\mathrm{A},\mathrm{B}),0) =$

Auxiliary Functions for Intersection

In order to define our intersection operation on EFAs, we first need to define two auxiliary functions. Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextSym}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \times \mathbf{Sym} \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextSym}_{M_1,M_2}((q,r),a) =$

$$\{\,(q',r') \mid q,a \to q' \in T_{M_1} \text{ and } r,a \to r' \in T_{M_2}\,\}.$$

If M_1 and M_2 are our example EFAs, then

- $\mathbf{nextSym}((\mathsf{A},\mathsf{A}),0) = \emptyset$; and
- $\mathbf{nextSym}((\mathsf{A},\mathsf{B}),0) = \{(\mathsf{A},\mathsf{B})\}$.

Auxiliary Functions

Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextEmp}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextEmp}_{M_1,M_2}(q, r) =$

$$\{\, (q', r) \mid q, \% \to q' \in T_{M_1} \,\} \cup \{\, (q, r') \mid r, \% \to r' \in T_{M_2} \,\}.$$

Auxiliary Functions

Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextEmp}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \rightarrow \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextEmp}_{M_1,M_2}(q, r) =$

$$\{\, (q', r) \mid q, \% \rightarrow q' \in T_{M_1} \,\} \cup \{\, (q, r') \mid r, \% \rightarrow r' \in T_{M_2} \,\}.$$

If M_1 and M_2 are our example EFAs, then

- $\mathbf{nextEmp}(\mathsf{A}, \mathsf{A}) =$
- $\mathbf{nextEmp}(\mathsf{A}, \mathsf{B}) =$
- $\mathbf{nextEmp}(\mathsf{B}, \mathsf{A}) =$
- $\mathbf{nextEmp}(\mathsf{B}, \mathsf{B}) =$

Auxiliary Functions

Suppose M_1 and M_2 are EFAs. We define a function

$$\textbf{nextEmp}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\textbf{nextEmp}_{M_1,M_2}(q, r) =$

$$\{\, (q', r) \mid q, \% \to q' \in T_{M_1} \,\} \cup \{\, (q, r') \mid r, \% \to r' \in T_{M_2} \,\}.$$

If M_1 and M_2 are our example EFAs, then

- $\textbf{nextEmp}(\mathsf{A}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{A}), (\mathsf{A}, \mathsf{B})\}$;
- $\textbf{nextEmp}(\mathsf{A}, \mathsf{B}) =$
- $\textbf{nextEmp}(\mathsf{B}, \mathsf{A}) =$
- $\textbf{nextEmp}(\mathsf{B}, \mathsf{B}) =$

Auxiliary Functions

Suppose M_1 and M_2 are EFAs. We define a function

$$\textbf{nextEmp}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\textbf{nextEmp}_{M_1,M_2}(q, r) =$

$$\{\, (q', r) \mid q, \% \to q' \in T_{M_1} \,\} \cup \{\, (q, r') \mid r, \% \to r' \in T_{M_2} \,\}.$$

If M_1 and M_2 are our example EFAs, then

- $\textbf{nextEmp}(\mathsf{A}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{A}), (\mathsf{A}, \mathsf{B})\}$;
- $\textbf{nextEmp}(\mathsf{A}, \mathsf{B}) = \{(\mathsf{B}, \mathsf{B})\}$;
- $\textbf{nextEmp}(\mathsf{B}, \mathsf{A}) =$
- $\textbf{nextEmp}(\mathsf{B}, \mathsf{B}) =$

Auxiliary Functions

Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextEmp}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextEmp}_{M_1,M_2}(q, r) =$

$$\{\,(q', r) \mid q, \% \to q' \in T_{M_1}\,\} \cup \{\,(q, r') \mid r, \% \to r' \in T_{M_2}\,\}.$$

If M_1 and M_2 are our example EFAs, then

- $\mathbf{nextEmp}(\mathsf{A}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{A}), (\mathsf{A}, \mathsf{B})\}$;
- $\mathbf{nextEmp}(\mathsf{A}, \mathsf{B}) = \{(\mathsf{B}, \mathsf{B})\}$;
- $\mathbf{nextEmp}(\mathsf{B}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{B})\}$; and
- $\mathbf{nextEmp}(\mathsf{B}, \mathsf{B}) =$

Auxiliary Functions

Suppose M_1 and M_2 are EFAs. We define a function

$$\mathbf{nextEmp}_{M_1,M_2} \in (Q_{M_1} \times Q_{M_2}) \to \mathcal{P}(Q_{M_1} \times Q_{M_2})$$

by $\mathbf{nextEmp}_{M_1,M_2}(q, r) =$

$$\{\,(q', r) \mid q, \% \to q' \in T_{M_1}\,\} \cup \{\,(q, r') \mid r, \% \to r' \in T_{M_2}\,\}.$$

If M_1 and M_2 are our example EFAs, then

- $\mathbf{nextEmp}(\mathsf{A}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{A}), (\mathsf{A}, \mathsf{B})\}$;
- $\mathbf{nextEmp}(\mathsf{A}, \mathsf{B}) = \{(\mathsf{B}, \mathsf{B})\}$;
- $\mathbf{nextEmp}(\mathsf{B}, \mathsf{A}) = \{(\mathsf{B}, \mathsf{B})\}$; and
- $\mathbf{nextEmp}(\mathsf{B}, \mathsf{B}) = \emptyset$.

Intersection Algorithm

Now, we define a function/algorithm $\mathbf{inter} \in \mathbf{EFA} \times \mathbf{EFA} \to \mathbf{EFA}$ such that $L(\mathbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$, for all $M_1, M_2 \in \mathbf{EFA}$. Given EFAs M_1 and M_2, $\mathbf{inter}(M_1, M_2)$ is the EFA N that is constructed as follows.

Intersection Algorithm

Now, we define a function/algorithm $\textbf{inter} \in \textbf{EFA} \times \textbf{EFA} \to \textbf{EFA}$ such that $L(\textbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$, for all $M_1, M_2 \in \textbf{EFA}$. Given EFAs M_1 and M_2, $\textbf{inter}(M_1, M_2)$ is the EFA N that is constructed as follows.

First, we let $\Sigma = \textbf{alphabet}\ M_1 \cap \textbf{alphabet}\ M_2$.

Intersection Algorithm

Now, we define a function/algorithm $\textbf{inter} \in \textbf{EFA} \times \textbf{EFA} \to \textbf{EFA}$ such that $L(\textbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$, for all $M_1, M_2 \in \textbf{EFA}$. Given EFAs M_1 and M_2, $\textbf{inter}(M_1, M_2)$ is the EFA N that is constructed as follows.

First, we let $\Sigma = \textbf{alphabet}\, M_1 \cap \textbf{alphabet}\, M_2$.

Next, we generate the least subset X of $Q_{M_1} \times Q_{M_2}$ such that

- $(s_{M_1}, s_{M_2}) \in X$;
- for all $q \in Q_{M_1}$, $r \in Q_{M_2}$ and $a \in \Sigma$, if $(q, r) \in X$, then $\textbf{nextSym}((q, r), a) \subseteq X$; and
- for all $q \in Q_{M_1}$ and $r \in Q_{M_2}$, if $(q, r) \in X$, then $\textbf{nextEmp}(q, r) \subseteq X$.

Intersection Algorithm

Then, the EFA N is defined by:

- $Q_N =$
- $s_N =$
- $A_N =$
- $T_N =$

Intersection Algorithm

Then, the EFA N is defined by:

- $Q_N = \{ \langle q, r \rangle \mid (q, r) \in X \}$;
- $s_N =$
- $A_N =$
- $T_N =$

Intersection Algorithm

Then, the EFA N is defined by:

- $Q_N = \{\, \langle q, r \rangle \mid (q, r) \in X \,\}$;
- $s_N = \langle s_{M_1}, s_{M_2} \rangle$;
- $A_N =$
- $T_N =$

Intersection Algorithm

Then, the EFA N is defined by:

- $Q_N = \{\, \langle q, r\rangle \mid (q, r) \in X \,\}$;
- $s_N = \langle s_{M_1}, s_{M_2}\rangle$;
- $A_N = \{\, \langle q, r\rangle \mid (q, r) \in X$ and $q \in A_{M_1}$ and $r \in A_{M_2} \,\}$; and
- $T_N =$

Intersection Algorithm

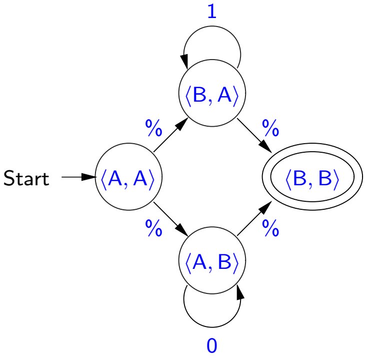
Then, the EFA N is defined by:

- $Q_N = \{ \langle q, r \rangle \mid (q, r) \in X \}$;
- $s_N = \langle s_{M_1}, s_{M_2} \rangle$;
- $A_N = \{ \langle q, r \rangle \mid (q, r) \in X$ and $q \in A_{M_1}$ and $r \in A_{M_2} \}$; and
- $T_N =$

$$
\begin{aligned}
&\{ \langle q, r \rangle, a \rightarrow \langle q', r' \rangle \mid (q, r) \in X \text{ and } a \in \Sigma \text{ and } \\
&\qquad\qquad (q', r') \in \mathbf{nextSym}((q, r), a) \} \\
\cup\ &\{ \langle q, r \rangle, \% \rightarrow \langle q', r' \rangle \mid (q, r) \in X \text{ and } \\
&\qquad\qquad (q', r') \in \mathbf{nextEmp}(q, r) \}.
\end{aligned}
$$

Intersection Example

Suppose M_1 and M_2 are our example EFAs. Then **inter**(M_1, M_2) is



Theorem 3.12.15

For all $M_1, M_2 \in \mathbf{EFA}$*:*

- $L(\mathbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$*; and*
- $\mathbf{alphabet}(\mathbf{inter}(M_1, M_2)) \subseteq \mathbf{alphabet}\, M_1 \cap \mathbf{alphabet}\, M_2$.

Intersection Algorithm

Theorem 3.12.15
For all $M_1, M_2 \in \mathbf{EFA}$:

- $L(\mathbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$; *and*
- $\mathbf{alphabet}(\mathbf{inter}(M_1, M_2)) \subseteq \mathbf{alphabet}\, M_1 \cap \mathbf{alphabet}\, M_2$.

Proposition 3.12.16
For all $M_1, M_2 \in \mathbf{NFA}$, $\mathbf{inter}(M_1, M_2) \in \mathbf{NFA}$.

Intersection Algorithm

Theorem 3.12.15
For all $M_1, M_2 \in \textbf{EFA}$*:*

- $L(\textbf{inter}(M_1, M_2)) = L(M_1) \cap L(M_2)$*; and*
- $\textbf{alphabet}(\textbf{inter}(M_1, M_2)) \subseteq \textbf{alphabet}\, M_1 \cap \textbf{alphabet}\, M_2$.

Proposition 3.12.16
For all $M_1, M_2 \in \textbf{NFA}$, $\textbf{inter}(M_1, M_2) \in \textbf{NFA}$.

Proposition 3.12.17
For all $M_1, M_2 \in \textbf{DFA}$*:*

(1) $\textbf{inter}(M_1, M_2) \in \textbf{DFA}$*; and*

(2) $\textbf{alphabet}(\textbf{inter}(M_1, M_2)) = \textbf{alphabet}\, M_1 \cap \textbf{alphabet}\, M_2$.

Complementation of DFAs

Next, we define a function **complement** $\in$ **DFA** $\times$ **Alp** $\rightarrow$ **DFA** such that, for all $M \in$ **DFA** and $\Sigma \in$ **Alp**,

$$L(\mathbf{complement}(M, \Sigma)) = (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M).$$

Complementation of DFAs

Next, we define a function $\mathbf{complement} \in \mathbf{DFA} \times \mathbf{Alp} \to \mathbf{DFA}$ such that, for all $M \in \mathbf{DFA}$ and $\Sigma \in \mathbf{Alp}$,

$$L(\mathbf{complement}(M, \Sigma)) = (\mathbf{alphabet}(L(M)) \cup \Sigma)^* - L(M).$$

In the common case when $L(M) \subseteq \Sigma^*$, we will have that $\mathbf{alphabet}(L(M)) \subseteq \Sigma$, and thus that $(\mathbf{alphabet}(L(M)) \cup \Sigma)^* = \Sigma^*$. Hence, it will be the case that

$$L(\mathbf{complement}(M, \Sigma)) = \Sigma^* - L(M).$$

Complementation of DFAs

Given a DFA M and an alphabet Σ, **complement**(M, Σ) is the DFA N that is produced as follows. First, we let the DFA $M' = \textbf{determSimplify}(M, \Sigma)$. Thus:

- M' is equivalent to M; and
- **alphabet** $M' = \textbf{alphabet}(L(M)) \cup \Sigma$.

Then, we define N by:

- $Q_N =$
- $s_N =$
- $A_N =$
- $T_N =$

Complementation of DFAs

Given a DFA M and an alphabet Σ, **complement**(M, Σ) is the DFA N that is produced as follows. First, we let the DFA $M' =$ **determSimplify**(M, Σ). Thus:

- M' is equivalent to M; and
- **alphabet** $M' =$ **alphabet**$(L(M)) \cup \Sigma$.

Then, we define N by:

- $Q_N = Q_{M'}$;
- $s_N = s_{M'}$;
- $A_N =$
- $T_N = T_{M'}$.

Complementation of DFAs

Given a DFA M and an alphabet Σ, **complement**(M, Σ) is the DFA N that is produced as follows. First, we let the DFA $M' = \textbf{determSimplify}(M, \Sigma)$. Thus:

- M' is equivalent to M; and
- **alphabet** $M' = \textbf{alphabet}(L(M)) \cup \Sigma$.

Then, we define N by:

- $Q_N = Q_{M'}$;
- $s_N = s_{M'}$;
- $A_N = Q_{M'} - A_{M'}$; and
- $T_N = T_{M'}$.

Complementation of DFAs

Then, for all
$w \in (\textbf{alphabet}\, M')^* = (\textbf{alphabet}\, N)^* = (\textbf{alphabet}(L(M)) \cup \Sigma)^*$,

$$\begin{aligned} w \in L(N) \quad &\text{iff} \quad \delta_N(s_N, w) \in A_N \\ &\text{iff} \quad \delta_N(s_N, w) \in Q_{M'} - A_{M'} \\ &\text{iff} \quad \delta_{M'}(s_{M'}, w) \notin A_{M'} \\ &\text{iff} \quad w \notin L(M') \\ &\text{iff} \quad w \notin L(M). \end{aligned}$$

Then, for all
$w \in (\textbf{alphabet}\, M')^* = (\textbf{alphabet}\, N)^* = (\textbf{alphabet}(L(M)) \cup \Sigma)^*$,

$$\begin{aligned} w \in L(N) \quad &\text{iff} \quad \delta_N(s_N, w) \in A_N \\ &\text{iff} \quad \delta_N(s_N, w) \in Q_{M'} - A_{M'} \\ &\text{iff} \quad \delta_{M'}(s_{M'}, w) \notin A_{M'} \\ &\text{iff} \quad w \notin L(M') \\ &\text{iff} \quad w \notin L(M). \end{aligned}$$

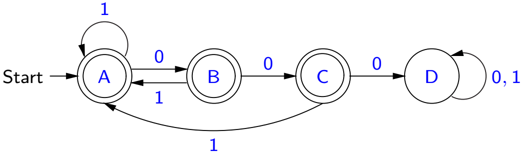
Hence:

Theorem 3.12.18
For all $M \in \textbf{DFA}$ and $\Sigma \in \textbf{Alp}$:

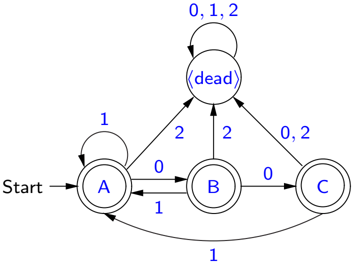
- $L(\textbf{complement}(M, \Sigma)) = (\textbf{alphabet}(L(M)) \cup \Sigma)^* - L(M)$; *and*
- $\textbf{alphabet}(\textbf{complement}(M, \Sigma)) = \textbf{alphabet}(L(M)) \cup \Sigma$.

Complementation Example

For example, suppose the DFA M is

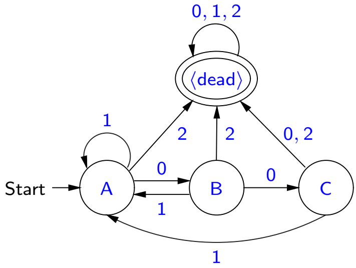


Then **determSimplify**$(M, \{2\})$ is the DFA



Complementation Example

Thus **complement**$(M, \{2\})$ is



Complementation Example

Let $X = \{ w \in \{0,1\}^* \mid 000$ is not a substring of $w \}$. Then $L(\mathbf{complement}(M, \{2\}))$ is

$$
\begin{aligned}
&(\mathbf{alphabet}(L(M)) \cup \{2\})^* - L(M) \\
&= (\{0,1\} \cup \{2\})^* - X \\
&= \{ w \in \{0,1,2\}^* \mid w \notin X \} \\
&= \{ w \in \{0,1,2\}^* \mid
\end{aligned}
$$

Complementation Example

Let $X = \{\, w \in \{0,1\}^* \mid 000 \text{ is not a substring of } w \,\}$. Then $L(\mathbf{complement}(M, \{2\}))$ is

$$
\begin{aligned}
&(\mathbf{alphabet}(L(M)) \cup \{2\})^* - L(M) \\
&= (\{0,1\} \cup \{2\})^* - X \\
&= \{\, w \in \{0,1,2\}^* \mid w \notin X \,\} \\
&= \{\, w \in \{0,1,2\}^* \mid 2 \in \mathbf{alphabet}\, w \text{ or } 000 \text{ is a substring of } w \,\}.
\end{aligned}
$$

Set Difference of DFAs

We define a function/algorithm **minus** $\in$ **DFA** $\times$ **DFA** $\rightarrow$ **DFA** by:

minus$(M_1, M_2) =$

Set Difference of DFAs

We define a function/algorithm **minus** $\in$ **DFA** $\times$ **DFA** $\to$ **DFA** by:

$$\mathbf{minus}(M_1, M_2) = \mathbf{inter}(M_1, \mathbf{complement}(M_2, \qquad\qquad)).$$

Set Difference of DFAs

We define a function/algorithm **minus** $\in$ **DFA** $\times$ **DFA** $\rightarrow$ **DFA** by:

$$\mathbf{minus}(M_1, M_2) = \mathbf{inter}(M_1, \mathbf{complement}(M_2, \mathbf{alphabet}\ M_1)).$$

Theorem 3.12.19

For all $M_1, M_2 \in \mathbf{DFA}$*:*

- $L(\mathbf{minus}(M_1, M_2)) = L(M_1) - L(M_2)$*; and*
- $\mathbf{alphabet}(\mathbf{minus}(M_1, M_2)) = \mathbf{alphabet}\ M_1$.

Theorem 3.12.19

For all $M_1, M_2 \in \mathbf{DFA}$:

- $L(\mathbf{minus}(M_1, M_2)) = L(M_1) - L(M_2)$*; and*
- $\mathbf{alphabet}(\mathbf{minus}(M_1, M_2)) = \mathbf{alphabet}\ M_1$*.*

Proof.

$$
\begin{aligned}
& w \in L(\mathbf{minus}(M_1, M_2)) \\
\text{iff} \quad & w \in L(\mathbf{inter}(M_1, \mathbf{complement}(M_2, \mathbf{alphabet}\ M_1))) \\
\text{iff} \quad & w \in L(M_1) \text{ and } w \in L(\mathbf{complement}(M_2, \mathbf{alphabet}\ M_1)) \\
\text{iff} \quad & w \in L(M_1) \text{ and } w \in (\mathbf{alphabet}(L(M_2)) \cup \mathbf{alphabet}\ M_1)^* \text{ and} \\
& w \notin L(M_2) \\
\text{iff} \quad & w \in L(M_1) \text{ and } w \notin L(M_2) \\
\text{iff} \quad & w \in L(M_1) - L(M_2).
\end{aligned}
$$

□

Theorem 3.12.29
*Suppose $L, L_1, L_2 \in$ **RegLan**. Then:*

- $L_1 \cup L_2 \in$ **RegLan** *(because of the operation* **union** *on FAs);*
- $L_1 L_2 \in$ **RegLan** *(because of the operation* **concat** *on FAs);*
- $L^* \in$ **RegLan** *(because of the operation* **closure** *on FAs);*
- $L_1 \cap L_2 \in$ **RegLan** *(because of the operation* **inter** *on EFAs); and*
- $L_1 - L_2 \in$ **RegLan** *(because of the operation* **minus** *on DFAs).*

The book shows several additional closure properties of regular languages.

Intersections, Complementations and Differences in Forlan

The Forlan module `EFA` defines the function/algorithm

```
val inter : efa * efa -> efa
```

which corresponds to **inter**. It is also inherited by the modules `DFA` and `NFA`.

The Forlan module DFA defines the functions

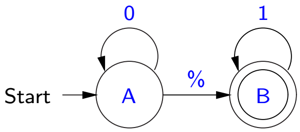
```
val complement : dfa * sym set -> dfa
val minus      : dfa * dfa -> dfa
```

which correspond to **complement** and **minus**.

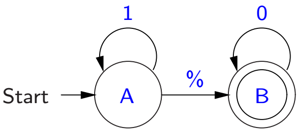
The book shows how several other operations on automata and regular expressions can be used in Forlan.

Forlan Examples

Suppose the identifiers `efa1` and `efa2` of type `efa` are bound to our example EFAs M_1 and M_2:



(M_1)



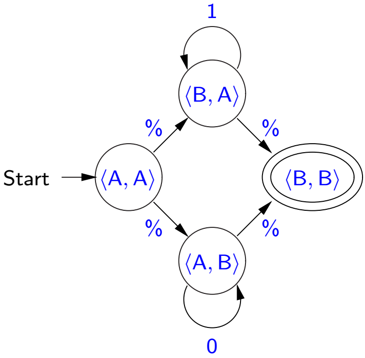
(M_2)

Then, we can construct **inter**(M_1, M_2) as follows:

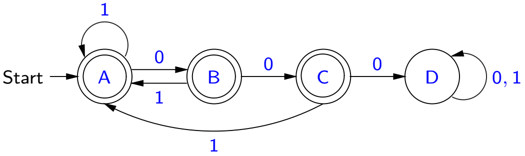
```
- val efa = EFA.inter(efa1, efa2);
val efa = - : efa
- EFA.output("", efa);
{states} <A,A>, <A,B>, <B,A>, <B,B>
{start state} <A,A> {accepting states} <B,B>
{transitions}
<A,A>, % -> <A,B> | <B,A>; <A,B>, % -> <B,B>;
<A,B>, 0 -> <A,B>; <B,A>, % -> <B,B>;
<B,A>, 1 -> <B,A>
val it = () : unit
```

Forlan Examples

Thus `efa` is bound to the EFA



Suppose `dfa` is bound to our example DFA M



Then we can construct the DFA **complement**$(M, \{2\})$ as follows:

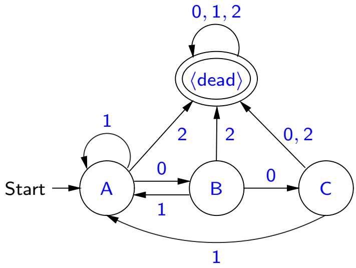
```
- val dfa' = DFA.complement(dfa, SymSet.input "");
@ 2
@ .
val dfa' = - : dfa
```

Forlan Examples

```
- DFA.output("", dfa');
{states} A, B, C, <dead> {start state} A
{accepting states} <dead>
{transitions}
A, 0 -> B; A, 1 -> A; A, 2 -> <dead>; B, 0 -> C;
B, 1 -> A; B, 2 -> <dead>; C, 0 -> <dead>; C, 1 -> A;
C, 2 -> <dead>; <dead>, 0 -> <dead>;
<dead>, 1 -> <dead>; <dead>, 2 -> <dead>
val it = () : unit
```

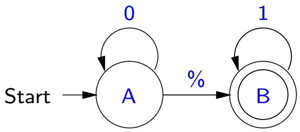
Forlan Examples

Thus `dfa'` is bound to the DFA

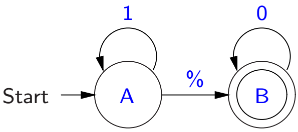


Forlan Examples

Suppose the identifiers `efa1` and `efa2` of type `efa` are bound to our example EFAs M_1 and M_2:



(M_1)

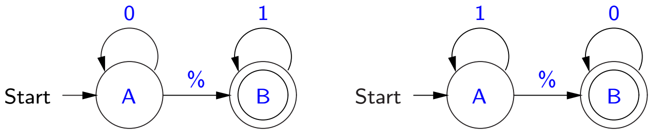


(M_2)

We can construct an EFA that accepts $L(M_1) - L(M_2)$ as follows:

Forlan Examples

Suppose the identifiers `efa1` and `efa2` of type `efa` are bound to our example EFAs M_1 and M_2:



(M_1) (M_2)

We can construct an EFA that accepts $L(M_1) - L(M_2)$ as follows:

```
- val dfa1 = nfaToDFA(efaToNFA efa1);
val dfa1 = - : dfa
- val dfa2 = nfaToDFA(efaToNFA efa2);
val dfa2 = - : dfa
- val dfa = DFA.minus(dfa1, dfa2);
val dfa = - : dfa
```

```
- val efa = injDFAToEFA dfa;
val efa = - : efa
- EFA.accepted efa (Str.input "");
@ 01
@ .
val it = true : bool
- EFA.accepted efa (Str.input "");
@ 0
@ .
val it = false : bool
```