

Unless specified otherwise, r,s,t,u,v,w,x,y,z are strings over alphabet Σ ; while a, b, c, d are individual alphabet symbols.

DFSM notation: $M = (K, \Sigma, \delta, s, A)$, where:

K is a finite set of **states**, Σ is a finite **alphabet**

$s \in K$ is start state, $A \subseteq K$ is set of **accepting states**

$\delta: (K \times \Sigma) \rightarrow K$ is the **transition function**

Extend δ 's definition to $\delta: (K \times \Sigma^*) \rightarrow K$ by the recursive definition $\delta(q, \epsilon) = q$, $\delta(q, xa) = \delta(\delta(q, x), a)$

M accepts w iff $\delta(s, w) \in A$. $L(M) = \{w \in \Sigma^* : \delta(s, w) \in A\}$

Alternate notation:

(q, w) is a **configuration** of M . (current state, remaining input)

The **yields-in-one-step** relation: \vdash_M :

$(q, w) \vdash_M (q', w')$ iff $w = aw'$ for some symbol $a \in \Sigma$, and $\delta(q, a) = q'$

The **yields-in-zero-or-more-steps** relation: \vdash_M^* is the reflexive, transitive closure of \vdash_M .

A **computation** by M is a finite sequence of configurations C_0, C_1, \dots, C_n for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ϵ) , for some state $q \in K_M$,
- $\forall i \in \{0, 1, \dots, n-1\} (C_i \vdash_M C_{i+1})$

M **accepts** w iff the state that is part of the last step in w is in A .

A language L is **regular** if $L = L(M)$ for some DFSM M .

In an **NDFSM**, the function δ is replaced by the relation $\Delta: \Delta \subseteq (K \times (\Sigma \cup \{\epsilon\})) \times K$

$ndfsm\ to\ dfsm(M: NDFSM) =$

1. For each state q in K_M do:
 - 1.1 Compute $eps(q)$.
2. $s' = eps(s)$
3. Compute δ' :
 - 3.1 **active-states** = $\{s\}$.
 - 3.2 $\delta' = \emptyset$.
 - 3.3 While there exists some element Q of **active-states** for which δ' has not yet been computed do:
 - For each character c in Σ_M do:
 - $new_state = \emptyset$.
 - For each state q in Q do:
 - For each state p such that $(q, c, p) \in \Delta$ do:
 - $new_state = new_state \cup eps(p)$.
 - Add the transition (q, c, new_state) to δ' .
 - If $new_state \notin active_states$ then insert it.
4. $K' = active_states$.
5. $A' = \{Q \in K' : Q \cap A \neq \emptyset\}$.

Some functions over languages:

$maxstring(L) =$

$$\{w \in L : \forall z \in \Sigma^* (z \neq \epsilon \rightarrow wz \notin L)\}.$$

$chop(L) =$

$$\{w : \exists x \in L (x = x_1cx_2, x_1 \in \Sigma_L^*, x_2 \in \Sigma_L^*, c \in \Sigma_L, |x_1| = |x_2|, \text{ and } w = x_1x_2)\}.$$

$firstchars(L) =$

$$\{w : \exists y \in L (y = cx \wedge c \in \Sigma_L \wedge x \in \Sigma_L^* \wedge w \in \{c\}^*)\}.$$

Equivalent strings relative to a language: Given a language L , two strings w and x in Σ_L^* are **indistinguishable** with respect to L , written $w \approx_L x$, iff $\forall z \in \Sigma^* (xz \in L \text{ iff } yz \in L)$.

$[x]$ is a notation for "the equivalence class that contains the string x ".

The construction of a minimal-state DSFM based on \approx_L :

$M = (K, \Sigma, \delta, s, A)$, where K contains n states, one for each equivalence class of \approx_L .

$s = [\epsilon]$, the equivalence class containing ϵ under \approx_L ,

$A = \{[x] : x \in L\}$,

$\delta([x], a) = [xa]$.

Enumerator (generator) for a language: when it is asked, enumerator gives us the next element of the language. Any given element of the language will appear within a finite amount of time. It is allowed that some may appear multiple times.

Recognizer: Given a string s , recognizer halts and accepts s if s is in the language. If not, recognizer either halts and rejects s or keeps running forever. This is a **semidecision procedure**. If recognizer is guaranteed to always halt and (accept or reject) no matter what string it is given as input, it is a **decision procedure**.

The **regular expressions** over an alphabet Σ are the strings that can be obtained as follows:

1. \emptyset is a regular expression.
2. ε is a regular expression.
3. Every element of Σ is a regular expression.
4. If α, β are regular expressions, then so is $\alpha\beta$.
5. If α, β are regular expressions, then so is $\alpha \cup \beta$.
6. If α is a regular expression, then so is α^+ .
7. α is a regular expression, then so is α^* .
8. If α is a regular expression, then so is (α) .

Reg. exp. operator precedence (High to Low):
 parenthesized expressions, * and +, concatenation, union

Functions on languages:

$firstchars(L) = \{w : \exists y \in L (y = cx, c \in \Sigma_L, x \in \Sigma_L^*, \text{ and } w \in c^*)\}$
 $chop(L) = \{w : \exists x \in L (x = x_1cx_2, x_1 \in \Sigma_L^*, x_2 \in \Sigma_L^*, c \in \Sigma_L \mid |x_1| = |x_2|, \text{ and } w = x_1x_2)\}$
 $maxstring(L) = \{w : w \in L, \forall z \in \Sigma^* (z \neq \varepsilon \rightarrow wz \notin L)\}$
 $mix(L) = \{w : \exists x, y, z (x \in L, x = yz, |y| = |z|, w = yz^R)\}$

Recursive formula for constructing a regular expression from a DFSM: r_{ijk} is $r_{ij(k-1)} \cup r_{ik(k-1)}(r_{kk(k-1)})^*r_{kj(k-1)}$

The set of regular languages is closed under complement, intersection, union, set difference, concatenation, Kleene * and +, reverse

Pumping Theorem and its contrapositive:

Formally, if L is regular, then

$\exists k \geq 1$ such that
 (\forall strings $w \in L$,
 $(|w| \geq k \rightarrow$
 $(\exists x, y, z (w = xyz,$
 $|xy| \leq k,$
 $y \neq \varepsilon, \text{ and}$
 $\forall q \geq 0 (xy^qz \text{ is in } L))))))$

The contrapositive form:

$(\forall k \geq 1$
 $(\exists \text{ a string } w \in L$
 $(|w| \geq k \text{ and}$
 $(\forall x, y, z$
 $((w = xyz \wedge |xy| \leq k \wedge y \neq \varepsilon) \rightarrow$
 $\exists q \geq 0 (xy^qz \text{ is not in } L)$
 $))) \rightarrow L \text{ is not regular}$

Summary of Algorithms

The next few slides are here for reference. I do not expect to spend class time on them.

You should know how to do all of them, but during today's exercises you may simply "call" any of them as part of your decision procedures.

- Operate on FSMs without altering the language that is accepted:
 - $ndfsmto fsm(M: NDFSM)$
 - $ninDFSM(M:DFSM)$
 - $buildFSMcanonicalform(M:FSM)$

Summary of Algorithms

- Compute functions of languages defined as FSMs:
 - Given FSMs M_1 and M_2 , construct a FSM M_3 such that $L(M_3) = L(M_2) \cup L(M_1)$.
 - Given FSMs M_1 and M_2 , construct a new FSM M_3 such that $L(M_3) = L(M_2) L(M_1)$.
 - Given FSM M , construct an FSM M^* such that $L(M^*) = (L(M))^*$.
 - Given a DFSM M , construct an FSM M^* such that $L(M^*) = \sim L(M)$.
 - Given two FSMs M_1 and M_2 , construct an FSM M_3 such that $L(M_3) = L(M_2) \cap L(M_1)$.
 - Given two FSMs M_1 and M_2 , construct an FSM M_3 such that $L(M_3) = L(M_2) - L(M_1)$.
 - Given an FSM M , construct an FSM M^* such that $L(M^*) = (L(M))^R$.
 - Given an FSM M , construct an FSM M^* that accepts $letsub(L(M))$.

Algorithms, Continued

- Converting between FSMs and regular expressions:
 - Given a regular expression α , construct an FSM M such that:

$$L(\alpha) = L(M)$$
 - Given an FSM M , construct a regular expression α such that:

$$L(\alpha) = L(M)$$
- Algorithms that implement operations on languages defined by regular expressions: any operation that can be performed on languages defined by FSMs can be implemented by converting all regular expressions to equivalent FSMs and then executing the appropriate FSM algorithm.

Algorithms, Continued

- Converting between FSMs and regular grammars:
 - Given a regular grammar G , construct an FSM M such that:

$$L(G) = L(M)$$
 - Given an FSM M , construct a regular grammar G such that:

$$L(G) = L(M)$$
.