Building Finite-State Machines
Xerox Finite-State Tool

- You’ll use it for homework ...
- Commercial (we have license; open-source clone is Foma)
  - One of several finite-state toolkits available
  - This one is easiest to use but doesn’t have probabilities
- Usage:
  - Enter a regular expression; it builds FSA or FST
  - Now type in input string
    - FSA: It tells you whether it’s accepted
    - FST: It tells you all the output strings (if any)
    - Can also invert FST to let you map outputs to inputs
  - Could hook it up to other NLP tools that need finite-state processing of their input or output
# Common Regular Expression Operators (in XFST notation)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>EF</td>
</tr>
<tr>
<td>iteration</td>
<td>E*, E+</td>
</tr>
<tr>
<td>union</td>
<td>E</td>
</tr>
<tr>
<td>intersection</td>
<td>E &amp; F</td>
</tr>
<tr>
<td>complementation, minus</td>
<td>~E, \x, F-E</td>
</tr>
<tr>
<td>crossproduct</td>
<td>E .x. F</td>
</tr>
<tr>
<td>composition</td>
<td>E .o. F</td>
</tr>
<tr>
<td>upper (input) language</td>
<td>E.u “domain”</td>
</tr>
<tr>
<td>lower (output) language</td>
<td>E.l “range”</td>
</tr>
</tbody>
</table>
Common Regular Expression Operators (in XFST notation)

concatenation

\[ EF = \{ ef: e \in E, f \in F \} \]

- \( ef \) denotes the concatenation of 2 strings.
- \( EF \) denotes the concatenation of 2 languages.
  - To pick a string in \( EF \), pick \( e \in E \) and \( f \in F \) and concatenate them.
  - To find out whether \( w \in EF \), look for at least one way to split \( w \) into two “halves,” \( w = ef \), such that \( e \in E \) and \( f \in F \).

A language is a set of strings.
It is a regular language if there exists an FSA that accepts all the strings in the language, and no other strings.
If \( E \) and \( F \) denote regular languages, than so does \( EF \).
(We will have to prove this by finding the FSA for \( EF \!\!\!\)!)
Common Regular Expression Operators (in XFST notation)

- **concatenation**: EF
- **iteration**: E*, E+

\[
E^* = \{ e_1 e_2 \ldots e_n : n \geq 0, \ e_1 \in E, \ldots \ e_n \in E \}
\]

- To pick a string in \( E^* \), pick any number of strings in \( E \) and concatenate them.
- To find out whether \( w \in E^* \), look for *at least one* way to split \( w \) into 0 or more sections, \( e_1 e_2 \ldots e_n \), all of which are in \( E \).

\[
E^+ = \{ e_1 e_2 \ldots e_n : n > 0, \ e_1 \in E, \ldots \ e_n \in E \} = E E^*
\]
Common Regular Expression Operators (in XFST notation)

- **concatenation** \( EF \)
- **iteration** \( E^*, E^+ \)
- **union** \( E \mid F \)

\[ E \mid F = \{ w : w \in E \text{ or } w \in F \} = E \cup F \]

- To pick a string in \( E \mid F \), pick a string from either \( E \) or \( F \).
- To find out whether \( w \in E \mid F \), check whether \( w \in E \) or \( w \in F \).
Common Regular Expression Operators (in XFST notation)

- concatenation \( EF \)
- iteration \( E^*, E^+ \)
- union \( E \cup F \)
- intersection \( E \cap F \)

\[ E \cap F = \{ w : w \in E \text{ and } w \in F \} = E \cap F \]

- To pick a string in \( E \cap F \), pick a string from \( E \) that is also in \( F \).
- To find out whether \( w \in E \cap F \), check whether \( w \in E \) and \( w \in F \).

600.465 - Intro to NLP - J. Eisner
Common Regular Expression Operators (in XFST notation)

concatenation                        $EF$
* +  iteration                       $E^*, E^+$
|  union                                $E \mid F$
&  intersection                        $E \& F$
~ \ \ complementation, minus         $\sim E, \setminus x, F-E$

$\sim E = \{e: e \notin E\} = \Sigma^* - E$

$E - F = \{e: e \in E \text{ and } e \notin F\} = E \& \sim F$

$\setminus E = \Sigma - E$ (any single character not in E)

$\Sigma$ is set of all letters; so $\Sigma^*$ is set of all strings
Regular Expressions

A **language** is a set of strings.

It is a **regular language** if there exists an FSA that accepts all the strings in the language, and no other strings.

If E and F denote regular languages, than so do EF, etc.

Regular expression: \( EF^*|(F & G)^+ \)

Syntax:

![Syntax Tree]

Semantics:

Denotes a regular language. As usual, can build semantics compositionally bottom-up.

\( E, F, G \) must be regular languages. As a base case, \( e \) denotes \{e\} (a language containing a single string), so \( ef^*|(f&g)^+ \) is regular.
Regular Expressions for Regular Relations

A **language** is a set of strings.
It is a **regular language** if there exists an FSA that accepts all the strings in the language, and no other strings.
If E and F denote regular languages, than so do EF, etc.

A **relation** is a set of pairs – here, pairs of strings.
It is a **regular relation** if there exists an FST that accepts all the pairs in the language, and no other pairs.
If E and F denote regular relations, then so do EF, etc.

\[ EF = \{(ef, e'f') : (e, e') \in E, (f, f') \in F\} \]

Can you guess the definitions for \( E^* \), \( E^+ \), \( E \mid F \), \( E \& F \) when E and F are regular relations?

Surprise: \( E \& F \) isn’t necessarily regular in the case of relations; so not supported.
# Common Regular Expression Operators (in XFST notation)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>concatenation</strong></td>
<td>EF</td>
</tr>
<tr>
<td>*</td>
<td>E*</td>
</tr>
<tr>
<td>+</td>
<td>E+</td>
</tr>
<tr>
<td><strong>iteration</strong></td>
<td>E*, E+</td>
</tr>
<tr>
<td><strong>union</strong></td>
<td>E</td>
</tr>
<tr>
<td><strong>intersection</strong></td>
<td>E &amp; F</td>
</tr>
<tr>
<td>~ \ -</td>
<td>~E, \x, F-E</td>
</tr>
<tr>
<td><strong>complementation, minus</strong></td>
<td>E \ F</td>
</tr>
<tr>
<td>.x.</td>
<td>E .x. F</td>
</tr>
</tbody>
</table>

\[
E .x. F = \{(e,f): e \in E, f \in F\}
\]

- Combines two regular languages into a regular relation.
Common Regular Expression Operators (in XFST notation)

- **concatenation** $EF$
- **iteration** $E^*, E^+$
- **union** $E \mid F$
- **intersection** $E \& F$
- **complementation, minus** $\sim E, \setminus x, F-E$
- **crossproduct** $E \cdot x \cdot F$
- **composition** $E \cdot o \cdot F$

$E \cdot o \cdot F = \{(e,f): \exists m. (e,m) \in E, (m,f) \in F\}$

- Composes two regular relations into a regular relation.
- As we’ve seen, this generalizes ordinary function composition.
Common Regular Expression Operators (in XFST notation)

- **concatenation** \( EF \)
- **iteration** \( E^*, E^+ \)
- **union** \( E | F \)
- **intersection** \( E \& F \)
- **complementation, minus** \( \sim E, \setminus x, F-E \)
- **crossproduct** \( E \times F \)
- **composition** \( E \circ F \)
- **upper (input) language** \( E.u \) “domain”

\[
E.u = \{ e : \exists m. (e,m) \in E \}
\]
Common Regular Expression Operators (in XFST notation)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>EF</td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td>E*, E+</td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>Intersection</td>
<td>E &amp; F</td>
<td></td>
</tr>
<tr>
<td>Complementation, minus</td>
<td>~E, \x, F-E</td>
<td></td>
</tr>
<tr>
<td>Crossproduct</td>
<td>E .x. F</td>
<td></td>
</tr>
<tr>
<td>Composition</td>
<td>E .o. F</td>
<td></td>
</tr>
<tr>
<td>Upper (input) language</td>
<td>E.u</td>
<td>“domain”</td>
</tr>
<tr>
<td>Lower (output) language</td>
<td>E.l</td>
<td>“range”</td>
</tr>
</tbody>
</table>
Function from strings to ...

**Acceptors (FSAs)**

Unweighted

- a
- c
- ε

- numbers
- a/.5
- c/.7
- ε/.5

Weighted

- {false, true}

- strings
- a:x
- c:z
- ε:γ

- (string, num) pairs
- a:x/.5
- c:z/.7
- ε:γ/.5

**Transducers (FSTs)**
Weighted Relations

- If we have a language [or relation], we can ask it: Do you contain this string [or string pair]?

- If we have a weighted language [or relation], we ask: What weight do you assign to this string [or string pair]?

- Pick a semiring: all our weights will be in that semiring.
  - Just as for parsing, this makes our formalism & algorithms general.
  - The unweighted case is the boolean semiring {true, false}.
  - If a string is not in the language, it has weight $\circ$.
  - If an FST or regular expression can choose among multiple ways to match, use $\oplus$ to combine the weights of the different choices.
  - If an FST or regular expression matches by matching multiple substrings, use $\otimes$ to combine those different matches.
  - Remember, $\oplus$ is like “or” and $\otimes$ is like “and”!
Which Semiring Operators are Needed?

- Concatenation: \( EF \)
- Iteration: \( E^*, E^+ \)
- Union: \( E \mid F \)
- Complementation, minus: \( \sim E, \setminus x, E-F \)
- Intersection: \( E \& F \)
- Crossproduct: \( E \times F \)
- Composition: \( E \circ F \)
- Upper (input) language: \( E.u \) “domain”
- Lower (output) language: \( E.l \) “range”
Common Regular Expression Operators (in XFST notation)

| union          | $\oplus$ to sum over 2 choices | $E | F$

$E | F = \{w: w \in E \text{ or } w \in F\} = E \cup F$

- Weighted case: Let’s write $E(w)$ to denote the weight of $w$ in the weighted language $E$.

$$(E|F)(w) = E(w) \oplus F(w)$$
Which Semiring Operators are Needed?

- **concatenation** need both $\oplus$ and $\otimes$
- **iteration** $E^*, E^+$
- **union** $\oplus$ to sum over 2 choices $E \mid F$
- **complementation, minus** $\sim E, \setminus x, E-F$
- **intersection** $\otimes$ to combine the matches against $E$ and $F$ $E \& F$
- **crossproduct** $E \times F$
- **composition** $E .o. F$
- **upper (input) language** $E.u$ “domain”
- **lower (output) language** $E.l$ “range”
Which Semiring Operators are Needed?

- concatenation
- + iteration

\[ EF = \{ ef: e \in E, f \in F \} \]

- Weighted case must match two things (\( \otimes \)), but there's a choice (\( \oplus \)) about which two things:

\[ (EF)(w) = \bigoplus_{e,f} (E(e) \otimes F(f)) \]

\( e,f \) such that \( w=ef \)
Which Semiring Operators are Needed?

- **concatenation**: 
  - need both $\oplus$ and $\otimes$
  - $EF$

- **iteration**: 
  - $E^*, E^+$

- **union**: 
  - $E \mid F$

- **complementation, minus**: 
  - $\sim E, \setminus x, E-F$

- **intersection**: 
  - $E \& F$

- **crossproduct**: 
  - $E .x. F$

- **composition**: 
  - both $\oplus$ and $\otimes$ (why?)
  - $E .o. F$

- **upper (input) language**: 
  - $E.u$ “domain”

- **lower (output) language**: 
  - $E.l$ “range”
Definition of FSTs

[Red material shows differences from FSAs.]

Simple view:
- An FST is simply a finite directed graph, with some labels.
- It has a designated initial state and a set of final states.
- Each edge is labeled with an “upper string” (in $\Sigma^*$).
- Each edge is also labeled with a “lower string” (in $\Delta^*$).
- [Upper/lower are sometimes regarded as input/output.]
- Each edge and final state is also labeled with a semiring weight.

More traditional definition specifies an FST via these:
- a state set $Q$
- initial state $i$
- set of final states $F$
- input alphabet $\Sigma$ (also define $\Sigma^*$, $\Sigma^+$, $\Sigma?$)
- output alphabet $\Delta$
- transition function $d: Q \times \Sigma? \rightarrow 2^Q$
- output function $s: Q \times \Sigma? \times Q \rightarrow \Delta?$
## How to implement?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>EF</td>
</tr>
<tr>
<td>Iteration</td>
<td>$E^* \text{, } E_+$</td>
</tr>
<tr>
<td>Union</td>
<td>$E \mid F$</td>
</tr>
<tr>
<td>Complementation</td>
<td>$\sim E \text{, } \setminus x \text{, } E-F$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$E &amp; F$</td>
</tr>
<tr>
<td>Crossproduct</td>
<td>$E \times F$</td>
</tr>
<tr>
<td>Composition</td>
<td>$E \circ F$</td>
</tr>
<tr>
<td>Upper (input)</td>
<td>$E.u$ “domain”</td>
</tr>
<tr>
<td>Lower (output)</td>
<td>$E.l$ “range”</td>
</tr>
</tbody>
</table>

---

*slide courtesy of L. Karttunen (modified)*
Concatenation

example courtesy of M. Mohri
Union

example courtesy of M. Mohri
Closure (this example has outputs too)

why add new start state 4?
why not just make state 0 final?
Upper language (domain)

similarly construct lower language \( l \)
also called input & output languages

example courtesy of M. Mohri
Reversal

\[ \text{Example courtesy of M. Mohri} \]
Inversion

example courtesy of M. Mohri
Complementation

- Given a machine M, represent all strings \textit{not} accepted by M
- Just change final states to non-final and vice-versa
- Works only if machine has been determinized and completed first (why?)
Intersection

Example adapted from M. Mohri

\[
\begin{align*}
\text{fat}/0.5 \\
0 & \rightarrow \text{pig}/0.3 \\ & \rightarrow 1 \\ & \rightarrow \text{eats}/0 \\ & \rightarrow 2/0.8 \\
& \rightarrow \text{sleps}/0.6 \\
\text{pig}/0.4 & \rightarrow 0 \\
& \rightarrow \text{fat}/0.2 \\ & \rightarrow 1 \\ & \rightarrow \text{sleps}/1.3 \\ & \rightarrow 2/0.5 \\
& \rightarrow \text{eats}/0.6 \\
\& & \rightarrow \text{fat}/0.7 \\ & \rightarrow 0,0 \\ & \rightarrow 0,1 \\ & \rightarrow 1,1 \\ & \rightarrow 2,0/0.8 \\ & \rightarrow \text{eats}/0.6 \\ & \rightarrow 2,2/1.3 \\ & \rightarrow \text{sleps}/1.9
\end{align*}
\]
Intersection

Paths \texttt{0012} and \texttt{0110} both accept \texttt{fat pig eats}
So must the new machine: along path \texttt{0,0 0,1 1,1 2,0}
Intersection

Paths 00 and 01 both accept fat
So must the new machine: along path 0,0 0,1
Intersection

Paths 00 and 11 both accept pig
So must the new machine: along path 0,1 1,1
Intersection

Paths 12 and 12 both accept fat
So must the new machine: along path 1,1 2,2
Intersection

fat/0.5

pig/0.3

1

eats/0

2/0.8

sleeps/0.6

&

pig/0.4

fat/0.2

1

sleeps/1.3

2/0.5

eats/0.6

=

0,0

fat/0.7

0,1

pig/0.7

1,1

2,0/0.8

eats/0.6

sleeps/1.9

2,2/1.3
What Composition Means

ab?d → f → 3 → abcd → g → 4 → αβγδ

2 → abed → 2 → αβεδ
6 → abjd → 8 → αβ∈δ

...
What Composition Means

Relation composition: $f \circ g$

$ab?d \rightarrow 3+4 \alpha\beta\gamma\delta$

$2+2 \alpha\beta\epsilon\delta$

$6+8 \alpha\beta\in\delta$

...
Relation = set of pairs

\{ ab?d \rightarrow abcd, ab?d \rightarrow abed, ab?d \rightarrow abjd \}

\{ abcd \rightarrow \alpha\beta\gamma\delta, abed \rightarrow \alpha\beta\varepsilon\delta, abed \rightarrow \alpha\beta\in\delta \}

\text{does not contain any pair of the form } abjd \rightarrow ...
Relation = set of pairs

Relation = set of pairs

\[ \begin{align*} 
\text{Relation} &= \text{set of pairs} \\
\{ \begin{align*} 
\text{ab?d} &\rightarrow \text{abcd} \\
\text{ab?d} &\rightarrow \text{abed} \\
\text{ab?d} &\rightarrow \text{abjd} \\
\end{align*} \right\} \\
\{ \begin{align*} 
\text{abcd} &\rightarrow \alpha\beta\gamma\delta \\
\text{abed} &\rightarrow \alpha\beta\varepsilon\delta \\
\text{abed} &\rightarrow \alpha\beta\in\delta \\
\end{align*} \right\} \\
\end{align*} \]

\[ f \circ g = \{ x \rightarrow z : \exists y \ (x \rightarrow y \in f \text{ and } y \rightarrow z \in g) \} \]

where \( x, y, z \) are strings
Intersection vs. Composition

Intersection

\[ \begin{align*}
0 \xrightarrow{\text{pig}/0.3} 1 \quad \& \quad 1 \xrightarrow{\text{pig}/0.4} 1 \, = \, 0,1 \xrightarrow{\text{pig}/0.7} 1,1
\end{align*} \]

Composition

\[ \begin{align*}
\text{Wilbur:} \, \text{pig}/0.3 \quad \& \quad \text{pig:} \, \text{pink}/0.4 \quad = \quad \text{Wilbur:} \, \text{pink}/0.7
\end{align*} \]
Intersection vs. Composition

Intersection mismatch

[Diagram showing Intersection mismatch with examples involving "pig" and "elephant"]

Composition mismatch

[Diagram showing Composition mismatch with examples involving "Wilbur: pig" and "Wilbur: gray"]
Composition

Example courtesy of M. Mohri
Composition

\[ \text{graph} \]

\[ a:b \cdot o \cdot b:b = a:b \]
Composition

\[ \text{a:b \cdot o \cdot b:a} = \text{a:a} \]
Composition

\[ \text{Composition} \]

\[ a:b \odot b:a = a:a \]
Composition

\[ b:b \cdot o \cdot b:a = b:a \]
Composition

\[ \text{a:b .o. b:a} = \text{a:a} \]
Composition

\[ a:a \cdot o \cdot a:b = a:b \]
Composition

\[ \text{b:b .o. a:b} = \text{nothing} \]
(since intermediate symbol doesn’t match)
Composition

\[ b:b \circ b:a = b:a \]
Composition

\[ a:b \cdot o \cdot a:b = a:b \]
**Relation = set of pairs**

\[
\begin{align*}
\text{ab?d} & \rightarrow \text{abcd} \\
\text{ab?d} & \rightarrow \text{abed} \\
\text{ab?d} & \rightarrow \text{abjd} \\
\end{align*}
\]

\[
\begin{align*}
\text{abcd} & \rightarrow \alpha\beta\gamma\delta \\
\text{abed} & \rightarrow \alpha\beta\varepsilon\delta \\
\text{abed} & \rightarrow \alpha\beta \in \delta \\
\end{align*}
\]

\[
f \circ g = \{ x \rightarrow z : \exists y \ (x \rightarrow y \in f \text{ and } y \rightarrow z \in g) \}
\]

where \(x, y, z\) are strings.
Composition with Sets

- We’ve defined $A \cdot o \cdot B$ where both are FSTs
- Now extend definition to allow one to be a FSA
- Two relations (FSTs):
  $$A \circ B = \{x \rightarrow z: \exists y (x \rightarrow y \in A \text{ and } y \rightarrow z \in B)\}$$
- Set and relation:
  $$A \circ B = \{x \rightarrow z: x \in A \text{ and } x \rightarrow z \in B \}$$
- Relation and set:
  $$A \circ B = \{x \rightarrow z: x \rightarrow z \in A \text{ and } z \in B \}$$
- Two sets (acceptors) – same as intersection:
  $$A \circ B = \{x: x \in A \text{ and } x \in B \}$$
Composition and Coercion

- Really just treats a set as identity relation on set
  \{abc, pqr, ...\} = \{abc\rightarrow abc, pqr\rightarrow pqr, ...\}

- Two relations (FSTs):
  \[ A \circ B = \{x\rightarrow z: \exists y (x\rightarrow y \in A \text{ and } y\rightarrow z \in B)\} \]

- Set and relation is now special case (if \(\exists y\) then \(y=x\)):
  \[ A \circ B = \{x\rightarrow z: \quad x\rightarrow x \in A \text{ and } x\rightarrow z \in B \} \]

- Relation and set is now special case (if \(\exists y\) then \(y=z\)):
  \[ A \circ B = \{x\rightarrow z: \quad x\rightarrow z \in A \text{ and } z\rightarrow z \in B \} \]

- Two sets (acceptors) is now special case:
  \[ A \circ B = \{x\rightarrow x: \quad x\rightarrow x \in A \text{ and } x\rightarrow x \in B \} \]
3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - \{abed \rightarrow abed\} \cdot \text{Greek} = \{abed \rightarrow \alpha \beta \varepsilon \delta, abed \rightarrow \alpha \beta \varepsilon \delta\}
  - \{abed\} \cdot \text{Greek} = \{abed \rightarrow \alpha \beta \varepsilon \delta, abed \rightarrow \alpha \beta \varepsilon \delta\}
  - \{\{abed\} \cdot \text{Greek}\}.l = \{\alpha \beta \varepsilon \delta, \alpha \beta \varepsilon \delta\}

- **Feed several strings in parallel:**
  - \{abcd, abed\} \cdot \text{Greek} = \{abcd \rightarrow \alpha \beta \gamma \delta, abed \rightarrow \alpha \beta \varepsilon \delta, abed \rightarrow \alpha \beta \varepsilon \delta\}
  - \{\{abcd, abed\} \cdot \text{Greek}\}.l = \{\alpha \beta \gamma \delta, \alpha \beta \varepsilon \delta, \alpha \beta \varepsilon \delta\}

- **Filter result via Noε = \{\alpha \beta \gamma \delta, \alpha \beta \varepsilon \delta, \alpha \beta \varepsilon \delta, \ldots\}**
  - \{abcd, abed\} \cdot \text{Greek} \cdot \text{Noε} = \{abcd \rightarrow \alpha \beta \gamma \delta, abed \rightarrow \alpha \beta \varepsilon \delta\}
What are the “basic” transducers?

- The operations on the previous slides combine transducers into bigger ones.
- But where do we start?

- $a:\varepsilon$ for $a \in \Sigma$
- $\varepsilon:x$ for $x \in \Delta$

Q: Do we also need $a:x$? How about $\varepsilon:\varepsilon$?