Building Finite-State Machines
todo

- Why is it called closure?
  - What is a closed set?
  - Examples: closure of \{1\} or \{2\} or \{2,5\} under addition
- \(E^* = \text{eps} | E | EE | EEE | \ldots\)
- Show construction of complementation
- Non-final states have weight semizero
- Say what semiring is used in the examples (and maybe change to prob semiring)
- Show construction of final states
- Complain about Perl; explain why union/concat/closure is the standard subset (theory, efficiency); consider \((A&B)^*\) for why boolean tests aren’t enough
- Mention that \? or . is the standard way to write Sigma
- Explain that \texttt{a:b} and \texttt{(e,f)} and \texttt{E .o. F} all have upper on left, lower on right (we need a 2-dimensional parser!)
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)

- Concatenation: \( EF \)
- Iteration: \( E^*, E^+ \)
- Union: \( E \cup F \)
- Intersection: \( E \cap F \)
- Complementation, minus: \( \sim E, \setminus x, F-E \)
- 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)

concatenation $EF$

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

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

$E^* = \{e_1e_2 \ldots e_n : n \geq 0, e_1 \in E, \ldots, e_n \in E\}$

- 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, $e_1e_2 \ldots e_n$, all of which are in $E$.

$E^+ = \{e_1e_2 \ldots e_n : n > 0, e_1 \in E, \ldots, e_n \in E\} = EE^*$
Common Regular Expression Operators (in XFST notation)

- **concatenation**: $EF$
- **iteration**: $E^*, E^+$
- **union**: $E | F$

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

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

- concatenation: \( EF \)
- iteration: \( E^*, E^+ \)
- union: \( E \mid F \)
- 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 \( w \in F \).
Common Regular Expression Operators  
(in XFST notation)

- **concatenation**: \( EF \)
- **iteration**: \( E^*, E^+ \)
- **union**: \( E | F \)
- **intersection**: \( E \& F \)
- **complementation, minus**: \( \sim E, \backslash x, F-E \)

\[
\sim E = \{ e : e \notin E \} = \Sigma^* - E \\
E - F = \{ e : e \in E \text{ and } e \notin F \} = E \& \sim F \\
\backslash E = \Sigma - E \quad \text{(any single character not in E)}
\]

\( \Sigma \) 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:**

$$E \quad F^* \quad | \quad (F \quad &\quad G)^+$$

**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 | 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>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$</td>
</tr>
<tr>
<td>Complementation, minus</td>
<td>$\sim E$, \x, F-E</td>
</tr>
<tr>
<td>Crossproduct</td>
<td>$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** \( E \ | \ F \)
- **intersection** \( E \ & \ F \)
- **complementation, minus** \(~E, \ \setminus x, F - E\)
- **crossproduct** \( E \ .x. \ F \)
- **composition** \( E \ .o. \ F \)

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

- 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: $E .o. F$
- Upper (input) language: $E.u$ “domain”

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

- Concatenation: $EF$
- Iteration: $E^*, E^+$
- Union: $E \mid F$
- Intersection: $E \& F$
- Complementation, minus: $\sim E, \backslash x, F-E$
- Crossproduct: $E .x. F$
- Composition: $E .o. F$
- Upper (input) language: $E.u$ “domain”
- Lower (output) language: $E.l$ “range”
Function from strings to ...

Acceptors (FSAs)

Unweighted

\begin{align*}
&\text{a: } x \\
&\varepsilon: y
\end{align*}

Weighted

\begin{align*}
&\text{a: } x/0.5 \\
&\varepsilon: y/0.5
\end{align*}

Transducers (FSTs)

\begin{align*}
&\text{c: } z \\
&\text{strings: } x \\
&\varepsilon: y
\end{align*}

\begin{align*}
&\text{c: } z/0.7 \\
&\text{numbers: } 0.7 \\
&\varepsilon/0.5
\end{align*}

\begin{align*}
&\text{(string, num) pairs: } x/0.5 \\
&\varepsilon: y/0.5
\end{align*}
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 \( \oplus \).
  - 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
  $E \cdot F$
- iteration
  $E^*, E^+$
- union
  $E \cup F$
- complementation, minus
  $\sim E, \setminus x, E - F$
- intersection
  $E \cap F$
- 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 | $E \mid F$

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

- 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?

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Semiring Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>*   +</td>
<td>concatenation</td>
<td>EF</td>
</tr>
<tr>
<td>*   +</td>
<td>iteration</td>
<td>E*, E+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~E, (\backslash x), E-F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E &amp; F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E .(\times). F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E .(\circ). F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E.u “domain”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E.l “range”</td>
</tr>
</tbody>
</table>
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:

\[ (EF)(w) = \bigoplus (E(e) \otimes F(f)) \]

need both \( \oplus \) and \( \otimes \)

\( E^*, E+ \)
Which Semiring Operators are Needed?

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>EF</td>
<td></td>
</tr>
<tr>
<td>* + iteration</td>
<td>need both $\oplus$ and $\otimes$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E*, E+</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>union</td>
<td>to sum over 2 choices</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>~ \ in complementation, minus</td>
<td>$\sim$E, \x, E-F</td>
</tr>
<tr>
<td></td>
<td>\ in intersection</td>
<td>$\otimes$ to combine the matches against E and F</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>.x. in crossproduct</td>
<td>E .x. F</td>
</tr>
<tr>
<td></td>
<td>.o. in composition</td>
<td>both $\oplus$ and $\otimes$ (why?)</td>
</tr>
<tr>
<td></td>
<td>E .o. F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>u in upper (input) language</td>
<td>$\oplus$ E.u “domain”</td>
</tr>
<tr>
<td></td>
<td>l in lower (output) language</td>
<td>$\oplus$ E.l “range”</td>
</tr>
</tbody>
</table>
**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>Syntax</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>Complementation, minus</td>
<td>~E, \x, E-F</td>
</tr>
<tr>
<td>Intersection</td>
<td>E &amp; F</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>
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)

example courtesy of M. Mohri

similarly construct lower language .l also called input & output languages
Reversal

$\varepsilon \varepsilon /0.7$

$=\quad \quad \quad \quad \quad \quad 3/0.5$

example courtesy of M. Mohri
Inversion

example courtesy of M. Mohri
Intersection

- fat/0.5
- pig/0.3
- eats/0
- sleeps/0.6

- fat/0.2
- pig/0.4
- sleeps/1.3

- fat/0.7
- pig/0.7
- eats/0.6
- sleeps/1.9

Example adapted from M. Mohri
Intersection

Paths 0012 and 0110 both accept fat pig eats
So must the new machine: along path 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

\[ \begin{align*}
\text{fat/0.5} & \quad \rightarrow \quad \text{pig/0.3} \quad \rightarrow \quad 1 \quad \rightarrow \quad \text{eats/0} \quad \rightarrow \quad 2/0.8 \\
& \quad \downarrow \text{sleeps/0.6} \\
\text{pig/0.4} & \quad \rightarrow \quad \text{fat/0.2} \quad \rightarrow \quad 1 \quad \rightarrow \quad \text{sleeps/1.3} \quad \rightarrow \quad 2/0.5 \\
& \quad \downarrow \text{eats/0.6} \\
\& \quad \rightarrow \quad \text{0,0} \quad \rightarrow \quad 0,1 \quad \rightarrow \quad 1,1 \quad \rightarrow \quad 2,0/0.8 \quad \rightarrow \quad 2,2/1.3 \\
& \quad \downarrow \text{fat/0.7} \quad \rightarrow \quad \text{pig/0.7} \quad \rightarrow \quad \text{sleeps/1.9} \end{align*} \]
What Composition Means

\[
\begin{align*}
\text{ab?d} & \quad \rightarrow \quad \text{abcd} \quad \rightarrow \quad \alpha \beta \chi \delta \\
\text{abed} & \quad \rightarrow \quad \alpha \beta \epsilon \delta \\
\text{abjd} & \quad \rightarrow \quad \alpha \beta \delta \\
\end{align*}
\]
What Composition Means

Relation composition: $f \circ g$

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

$2\times2 \alpha + \beta + \epsilon + \delta$

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

...
Relation = set of pairs

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

\[
\begin{align*}
abcd &\mapsto \alpha \beta \chi \delta \\
abed &\mapsto \alpha \beta \varepsilon \delta \\
abed &\mapsto \alpha \beta \delta \\
\ldots
\end{align*}
\]

does not contain any pair of the form \(abjd \mapsto \ldots\)
Relation = set of pairs

\[
f \circ g = \{ e \mapsto f : \exists m ( e \mapsto m \in f \text{ and } m \mapsto f \in g ) \}
\]

where e, m, f are strings
Intersection vs. Composition

**Intersection**

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

**Composition**

\[
\begin{align*}
\text{Wilbur: pig/0.3} & \xrightarrow{\text{pig/0.3}} 1 \\
\& \\
\text{pig: pink/0.4} & \xrightarrow{\text{pig: pink/0.4}} 1 \\
= \\
0,1 & \xrightarrow{\text{Wilbur: pink/0.7}} 1,1
\end{align*}
\]
Intersection vs. Composition

Intersection mismatch

Wilbur: pig/0.3

\[ 0 \xrightarrow{\text{pig/0.3}} 1 \]

&

\[ 1 \xrightarrow{\text{pig/0.3}} 1 \]

= 

\[ 0,1 \xrightarrow{\text{pig/0.7}} 1,1 \]

Composition mismatch

\[ \text{elephant: gray/0.4} \]

\[ 0 \xrightarrow{\text{Wilbur: pig/0.3}} 1 \]

.0.

\[ 1 \xrightarrow{\text{Wilbur: gray/0.4}} 1 \]

= 

\[ 0,1 \xrightarrow{\text{Wilbur: gray/0.5}} 1,1 \]
Composition
d example courtesy of M. Mohri
Composition

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

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

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

b:b .o. b:a = b:a
Composition

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

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

b:b .o. a:b = nothing
(since intermediate symbol doesn’t match)
Composition

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

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

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

\[
\{ \begin{align*}
\text{abcd} & \mapsto \alpha \beta \chi \delta \\
\text{abed} & \mapsto \alpha \beta \epsilon \delta \\
\text{abed} & \mapsto \alpha \beta \delta \\
\ldots
\end{align*} \}
\]

\[
f \circ g = \{ e \mapsto f : \exists m (e \mapsto m \in f \text{ and } m \mapsto f \in g) \}
\]

where e, m, f are strings
Composition with Sets

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

- Really just treats a set as identity relation on set
  \{abc, pqr, ...\} = \{abc\mapsto abc, pqr\mapsto pqr, ...\}
- Two relations (FSTs):
  \( A \circ B = \{e\mapsto f: \exists m (e\mapsto m \in A \text{ and } m\mapsto f \in B)\} \)
- Set and relation is now special case (if \( \exists m \text{ then } y=x \)):
  \( A \circ B = \{e\mapsto f: \ e\mapsto e \in A \text{ and } e\mapsto f \in B \} \)
- Relation and set is now special case (if \( \exists m \text{ then } y=z \)):
  \( A \circ B = \{e\mapsto f: \ e\mapsto f \in A \text{ and } f\mapsto f \in B \} \)
- Two sets (acceptors) is now special case:
  \( A \circ B = \{e\mapsto e: \ e\mapsto e \in A \text{ and } e\mapsto e \in B \} \)
3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - \(\{\text{abed} \mapsto \text{abed}\} \circ \text{Greek} = \{\text{abed} \mapsto \alpha \ \beta, \epsilon\text{abed} \mapsto \alpha \ \beta \delta\}\)
  - \(\{\text{abed}\} \circ \text{Greek} = \{\text{abed} \mapsto \alpha \ \beta, \epsilon\text{abed} \mapsto \alpha \ \beta \delta\}\)
  - \([\{\text{abed}\} \circ \text{Greek}] \cdot l = \{\alpha \ \beta, \epsilon\alpha \delta \beta \delta\}\)

- **Feed several strings in parallel:**
  - \(\{\text{abcd, abed}\} \circ \text{Greek} = \{\text{abcd} \mapsto \alpha \ \beta \ \chi \text{abed} \mapsto \alpha \ \beta, \epsilon\text{abed} \mapsto \alpha \ \beta \delta\}\)
  - \([\{\text{abcd, abed}\} \circ \text{Greek}] \cdot l = \{\alpha \ \beta, \chi\alpha \delta \beta, \epsilon\alpha \delta \beta \delta\}\)

- **Filter result via \(\text{No} \epsilon = \{\alpha \ \beta, \chi \delta, \alpha \ \beta \delta, \ldots\}\)**
  - \(\{\text{abcd, abed}\} \circ \text{Greek} \circ \text{No} \epsilon = \{\text{abcd} \mapsto \alpha \ \beta \ \chi \text{abed} \mapsto \alpha \ \beta \delta\}\)
Complementation

- Given a machine $M$, represent all strings 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?)
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 $\varepsilon: \varepsilon$?