Regular Languages

Natural Language Processing
CS 6120—Spring 2014
Northeastern University

David Smith
with material from Jason Eisner, Andrew McCallum, and Lari Karttunen
Brief History: 1950s

- Early NLP on machines less powerful than pocket calculators
  - E.g., how to compress a word list into memory
- Foundational work on automata, formal languages, information theory
- First speech systems (Bell Labs)
- Machine translation heavily funded by military, basically just word substitution
- Little formalization of syntax, semantics, pragmatics
Brief History: 1960s

- ALPAC report (Alvey, 1966) ends funding for MT in U.S.
  - Lack of practical results, recommends basic research
- ELIZA and other early AI dialogue systems
  - Risibly easy Turing tests
- Early corpora: Brown Corpus (Kučera & Francis)
Brief History: 1970s

- Winograd’s SHRDLU (1971): existence proof of NLP (in tangled Lisp code)
  - Interpreted language about “blocks world”
    - Which cube is sitting on the table?
    - *The large green one which supports the red pyramid.*
    - Is there a large block behind the pyramid?
    - *Yes, three of them. A large red one, a large green cube, and the blue one.*
    - Put a small one onto the green cube which supports a pyramid.
    - *OK.*

- Hidden Markov models for speech recognition
Brief History: 1980s

- Procedural $\rightarrow$ declarative
  - Grammars, logic programming
  - Separation of processing (parser) from description of linguistic knowledge
- Representations of meaning: procedural semantics (SHRDLU), semantic nets (Schank), logic (starting in 1970s, Montague, Partee)
- Knowledge representation (Lenat: Cyc, still going!)
- MT in limited domains (METEO)
- HMMs for part-of-speech tagging (independently, Church & DeRose)
Brief History: 1990s

• Probabilistic paradigm shift
  ✤ Speech recognition methods take over the world

• IR-style evaluations take over the world

• Finite-state methods in speech and beyond

• Large amounts of monolingual and multilingual text become available, esp. on WWW

• Classification problems and *ambiguity resolution* (in syntax, lexical semantics, translation, etc.)
Brief History: Now

• Even more machine learning
  ✤ Successful unsupervised systems

• Even more data, and tasks

• Widely usable—and used—speech recognition and machine translation

• Widely usable syntactic parsing

• Some usable dialog systems

• Convergence with IR, question answering, probabilistic knowledge representation
Noam Chomsky 1928–

Formal languages (Chomsky hierarchy)

Generative grammar

Anarcho-Socialist
A Language

• Some sentences in the language
  ✤ The man took the book.
  ✤ Colorless green ideas sleep furiously.
  ✤ This sentence is false.

• Some sentences not in the language
  ✤ *The girl, the sidewalk, the chalk, drew.
  ✤ *Backwards is sentence this.
  ✤ *Je parle anglais.
Languages as Rewriting Systems

• Start with some “non-terminal” symbol S
• Expand that symbol, using a rewrite rule.
• Keep applying rules until all non-terminals are expanded to terminals.
• The string of terminals is a sentence of the language.
Chomsky Hierarchy

- Let Caps = nonterminals; lower = terminals; Greek = strings of terms/nonterms
- Recursively enumerable (Turing equivalent)
  - Rules: $\alpha \rightarrow \beta$
- Context-sensitive
  - Rules: $\alpha A \beta \rightarrow \alpha \gamma \beta$
- Context-free
  - Rules: $A \rightarrow \alpha$
- Regular (finite-state)
  - Rules: $A \rightarrow aB$ ; $A \rightarrow a$
Regular Language Example

- Nonterminals: S, X
- Terminals: m, o
- Rules:
  - S → mX
  - X → oX
  - X → o
- Start symbol: S

One expansion

S
mX
moX
mooX
mooo
Another Regular Language

- Strings in and not in this language
  - In the language:
    - “ba!”,”baa!”,”baaaaaaa!”
  - Not in the language:
    - “ba”,“b!”,“ab!”,“bbaaa!”,”alibaba!”

- Regular expression: baa*!

- Finite state automaton
Regular Languages

Regular Languages
the accepted strings

Finite-state Automata
machinery for accepting

Regular Expressions
a way to type the automata
Finite-State Automata

- A (deterministic) finite-state automaton is a 5-tuple \((Q, \Sigma, q_0, F, \delta(q,i))\)
  - \(Q\): finite set of states \(q_0, q_1, q_2, ..., q_N\)
  - \(\Sigma\): finite set of terminals
  - \(\delta(q,i)\): transition function (relation if non-deterministic)
  - \(q_0\): start state
  - \(F\): set of final states

The FSA

State marker

Input tape
# Transition Table

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$b$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Regular Expressions

• Two types of characters

• Literal
  ✤ Every “normal” alphanumeric character is an RE, and matches itself

• Meta-characters
  ✤ Special characters that allow you to combine REs in various ways

• Example:
  ✤ a matches a
  ✤ a* matches ε or a or aa or aaa or ...
### Regular Expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>abc</td>
<td>abc</td>
</tr>
<tr>
<td>Disjunction</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>(a</td>
<td>bb)</td>
</tr>
<tr>
<td>Kleene star</td>
<td>a*</td>
<td>ε a aa aaa ...</td>
</tr>
<tr>
<td></td>
<td>c (a</td>
<td>bb)*</td>
</tr>
</tbody>
</table>

Regular expressions / FSAs are closed under these operations.
Practical Applications

• Word processing find & replace
• Validate fields in database (dates, email, ...)
• Searching for linguistic patterns
• Finite-state machines
  ✤ Language modeling in speech recognition (where things need to be real-time or better)
  ✤ Information extraction
  ✤ Morphology
# Syntactic Sugar

<table>
<thead>
<tr>
<th>Character Concat</th>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>went</td>
<td>went</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternatives</th>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>disjunct. negation</td>
<td>(go</td>
<td>went)</td>
</tr>
<tr>
<td>wildcard char</td>
<td>[aeiou]</td>
<td>a o u</td>
</tr>
<tr>
<td></td>
<td>[^aeiou]</td>
<td>b c d f g</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>a z &amp;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loops &amp; skips</th>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>one or more</td>
<td>a*</td>
<td>ε a aa aaa ...</td>
</tr>
<tr>
<td>zero or one</td>
<td>a+</td>
<td>a aa aaa</td>
</tr>
<tr>
<td></td>
<td>colour?r</td>
<td>color colour</td>
</tr>
</tbody>
</table>
Syntactic Sugar

- **Special characters**
  - \t tab \v vertical tab
  - \n newline \r carriage return

- **Aliases (shorthand)**
  - \d digits [0-9]
  - \D non-digits [^0-9]
  - \w alphabetic [a-zA-Z]
  - \W non-alphabetic [^a-zA-Z]
  - \s whitespace [\t\n\r\f\v]
  - \w alphabetic [a-zA-Z]

- **Examples**
  - \d+ dollars 3 dollars, 50 dollars, 982 dollars
  - \w*oo\w* food, boo, oodles

- **Escape character**
  - \ is the general escape character; e.g. \. is not a wildcard, but matches a period .
  - if you want to use \ in a string it has to be escaped \\
Syntactic Sugar

• **Anchors.** AKA, “zero width characters”.
  • They match positions in the text.
    – `^` beginning of line
    – `$` end of line
    – `\b` word boundary, i.e. location with `\w` on one side but not on the other.
    – `\B` negated word boundary, i.e. any location that would not match `\b`

• **Examples:**
  – `\bthe\b` the together

• **Counters** `{1}`, `{1,2}`, `{3,}`
Syntactic Sugar

- **Grouping**
  - a (good|bad) movie
  - He said it (again and )*again.

- Parenthesis also indicate **Registers** (saved contents)
  - b(\w+)h\l
    matches boohoo and baha, but not boohaa
    The digit after the \ indicates which of multiple paren groups, as ordered by when then were opened.

- Grouping without the cost of register saving
  - He went (?:this|that) way.
Syntactic Sugar

• Non-consuming tests
  - (?=...) - Positive lookAHEAD
  - (?!...) - Negative lookAHEAD
  - (?<=...) - Positive lookBEHIND
  - (?<!...) - Negative lookBEHIND

• Example
  - (?=^[Hh]e) \w+ed (?=\w+ly)
ELIZA (Weizenbaum 1966)

User: Men are all alike.
ELIZA: IN WHAT WAY?
User: They’re always bugging us about something or other.
ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE?
User: Well, my boyfriend made me come here.
ELIZA: YOUR BOYFRIEND MADE YOU COME HERE.
User: He say’s I’m depressed much of the time.
ELIZA: I’M SORRY TO HEAR THAT YOU ARE DEPRESSED.

Implemented with regular expression substitution!
s/.* I’m (depressed|sad) .*/I AM SORRY TO HEAR THAT YOU ARE \1/
s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/?/
Reading

  

- RE/FSA background: Jurafsky & Martin, c.2
Finite-State Machines: Acceptors and Transducers
Finite state acceptors (FSAs)
Finite state acceptors (FSAs)

- Regexps
Finite state acceptors (FSAs)

- Regexps
- Union, Kleene *, concat, intersect, complement, reversal
Finite state acceptors (FSAs)

- Regexps
- Union, Kleene *, concat, intersect, complement, reversal
- Determinization, minimization
Finite state acceptors (FSAs)

- Regexps
- Union, Kleene *, concat, intersect, complement, reversal
- Determinization, minimization
- Pumping, Myhill-Nerode
A useful FSA ...

Wordlist

clear
clever
ear
ever
fat
father

/usr/dict/words
25K words
206K chars

0.6 sec

FSM
17728 states,
37100 arcs
A useful FSA ...

Wordlist

clear
clever
ear
ever
fat
father

compile

Network

FSM

17728 states,
37100 arcs

/usr/dict/words
25K words
206K chars

0.6 sec
Weights are useful here too!

Wordlist

<table>
<thead>
<tr>
<th>clear</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>clever</td>
<td>1</td>
</tr>
<tr>
<td>ear</td>
<td>2</td>
</tr>
<tr>
<td>ever</td>
<td>3</td>
</tr>
<tr>
<td>fat</td>
<td>4</td>
</tr>
<tr>
<td>father</td>
<td>5</td>
</tr>
</tbody>
</table>

Network

Computes a perfect hash!
Example: Weighted acceptor

- Compute **number of paths** from each state *(Q: how?)*
  - Successor states partition the path set
  - Use offsets of successor states as arc weights
- **Q:** Would this work for an arbitrary numbering of the words?
Example: Unweighted transducer

VP [head=vouloir,...]

V [head=vouloir, ...
    tense=Present,
    num=SG, person=P3]

veut
Example: Unweighted transducer

the problem of morphology ("word shape") - an area of linguistics
Example: Unweighted transducer

\[
\text{V}[\text{head}=\text{vouloir},\ldots]
\]

\[
\text{tense}=\text{Present},\quad \text{num}=\text{SG},\quad \text{person}=\text{P3}
\]

\[
\text{veut}
\]
Example: Unweighted transducer

vouloir +Pres +Sing + P3

Finite-state transducer

veut

VP [head=vouloir,...]

V [head=vouloir, ...]

tense=Present,
num=SG, person=P3]

veut
Example: Unweighted transducer

\[ \text{V}[\text{head}=\text{vouloir}, \ldots] \]
\[ \text{tense}=\text{Present}, \text{num}=\text{SG}, \text{person}=\text{P3} \]

\[ \text{veut} \]

\[ \text{V}[\text{head}=\text{vouloir}, \ldots] \]
\[ \text{tense}=\text{Present}, \text{num}=\text{SG}, \text{person}=\text{P3} \]

\[ \text{veut} \]
Example: Unweighted transducer

- Bidirectional: generation or analysis
- Compact and fast
- Xerox sells for about 20 languages including English, German, Dutch, French, Italian, Spanish, Portuguese, Finnish, Russian, Turkish, Japanese, ...
- Research systems for many other languages, including Arabic, Malay

Finite-state transducer

canonical form

inflection codes

the relevant path

inflected form
Regular Relation (of strings)
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs

- $b \rightarrow ?$  $a \rightarrow ?$

![Diagram of a regular relation](image)
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs

- $b \rightarrow \{b\}$  
  $a \rightarrow ?$

```
\begin{align*}
  \text{b} & \rightarrow \{\text{b}\} & \text{a} & \rightarrow ? \\
  \text{b} & \rightarrow \text{b} & \text{b} & \rightarrow \text{a} \\
  \text{a} & \rightarrow \varepsilon & \text{b} & \rightarrow \varepsilon \\
  \text{a} & \rightarrow \text{c} & \text{?} & \rightarrow \text{c} \\
  \text{b} & \rightarrow \text{b} & \text{?} & \rightarrow \text{b} \\
  \text{b} & \rightarrow \varepsilon & \text{?} & \rightarrow \varepsilon \\
  \end{align*}
```
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs

- $b \rightarrow \{b\}$  $a \rightarrow \{\}$
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs

- $b \rightarrow \{b\}$  
  $a \rightarrow \{\}$

  \{ac, aca, acab, acabc\}
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs

- $b \rightarrow \{b\}$  $a \rightarrow \{\}$
  $
\{ac, aca, acab, acabc\}$

- Invertible?
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs

- $b \rightarrow \{b\}$  $a \rightarrow \{\}$
  \[
  \{ac, aca, acab, acabc\}
  \]

- Invertible?
- Closed under composition?
Regular Relation (of strings)
Regular Relation (of strings)

- Can weight the arcs: \( \rightarrow \) vs. \( \rightarrow \)
Can weight the arcs: $\rightarrow$ vs. $\rightarrow$

- $b \rightarrow \{b\}$  $a \rightarrow \{\}$

Regular Relation (of strings)
Regular Relation (of strings)

- Can weight the arcs: $\rightarrow$ vs. $\rightarrow$
- $b \rightarrow \{b\}$  $a \rightarrow \{\}$
- $aaaaa \rightarrow \{ac, aca, acab, acabc\}$
Regular Relation (of strings)

- Can weight the arcs: $\rightarrow$ vs. $\rightarrow$
- $b \rightarrow \{b\}$, $a \rightarrow \{\}$
- $aaaaa \rightarrow \{ac, aca, acab, acabc\}$

- How to find best outputs?
Function from strings to ...

Acceptors (FSAs)

Transducers (FSTs)

Unweighted

Weighted
Function from strings to ...

Acceptors (FSAs)

Unweighted

- a
- ε

Transducers (FSTs)

- a:
- c:
- ε:

Weighted

- a/: .5
- ε/: .5
- c/: .7

{false, true}
Function from strings to ...

Acceptors (FSAs)

Unweighted

\[ \{\text{false, true}\} \]

\[ a \rightarrow c \]

\[ \varepsilon \rightarrow .5 \]

Weighted

\[ a/.5 \rightarrow \]

\[ \varepsilon/.5 \rightarrow .3 \]

\[ c/.7 \rightarrow \]

Transducers (FSTs)

Strings

\[ a:x \rightarrow c:z \]

\[ \varepsilon:y \rightarrow .5 \]

\[ \varepsilon:y/.5 \rightarrow .3 \]

\[ c:z/.7 \rightarrow \]
Function from strings to ...

Acceptors (FSAs)

Unweighted

Transducers (FSTs)

Weighted

{false, true}

strings

numbers

\( a: \times \)

\( a: x \)

\( a: x/\cdot 5 \)

\( a: x/\cdot 5 \)

\( \varepsilon: y \)

\( \varepsilon: y/\cdot 5 \)

\( c: z \)

\( c: z/\cdot 7 \)
Function from strings to ... 

Acceptors (FSAs)

Unweighted

{false, true}

Transducers (FSTs)

Weighted

Numbers

[(string, num) pairs]
Sample functions

Acceptors (FSAs)
- \{false, true\}

Transducers (FSTs)
- strings
- (string, num) pairs

Unweighted

Weighted
Sample functions

Acceptors (FSAs)

Unweighted

{false, true}

Grammatical? (false, true)

Weighted

numbers

Transducers (FSTs)

strings

(string, num) pairs
Sample functions

Acceptors (FSAs)
- Unweighted: Grammatical?
- Weighted: How grammatical? Better, how likely?

Transducers (FSTs)
- Unweighted: Strings
- Weighted: Numbers, (string, num) pairs
Sample functions

Acceptors (FSAs)

- Unweighted
  - \{false, true\}
  - Grammatical?

- Weighted
  - numbers
  - How grammatical?
  - Better, how likely?

Transducers (FSTs)

- Weighted
  - (string, num) pairs
  - Markup
  - Correction
  - Translation
Sample functions

Acceptors (FSAs)

- Unweighted: {false, true}
- Weighted: Grammatical?

Transducers (FSTs)

- Unweighted: strings
- Weighted:
  - Markup
  - Correction
  - Translation

Weighted:

- How grammatical?
- Better, how likely?

Good markups
Good corrections
Good translations
Terminology (acceptors)

- **Regexp**
  - matches
  - compiles into
  - matches

- **String**
  - accepts
  - matches

- **Regular language**
  - defines
  - compiles into
  - implements
  - recognizes

- **FSA**
  - implements
  - accepts
Terminology (acceptors)

- **Regexp** defines **Regular language** which compiles into **FSA** that implements **String** that matches **Regexp** that accepts (or generates) **Regular language** that recognizes **FSA**.
Terminology (transducers)

Regular relation
- defines
- compiles into
- implements
- matches
- matches
- accepts (or generates)

Regexp -> FST

String pair
Terminology (transducers)

Regular relation

Regexp

defines

matches

accepts
(or generates)

String pair

matches

compiles into

implements

FST

recognizes

(or, transduces one string of the pair into the other)
Terminology (transducers)

Regular relation

- Defines
- Recognizes
- Compiles into
- Implements

Regexp

String pair

- Matches
- Accepts
- (or generates)
- (or transduces one string of the pair into the other)
Perspectives on a Transducer

- Given 0 strings, **generate** a new string pair (by picking a path)
- Given **one** string (upper or lower), **transduce** it to the other kind
- Given **two** strings (upper & lower), **decide** whether to accept the pair

FST just defines the regular relation (mathematical object: set of pairs).
What’s “input” and “output” depends on what one **asks** about the relation.
The 0, 1, or 2 given string(s) constrain which paths you can use.
Functions

ab?d →
Functions

ab?d → f → abcd
Functions

\[ ab?d \rightarrow f \rightarrow abcd \rightarrow g \rightarrow \alpha\beta\chi\delta \]
Functions

Function composition: $f \circ g$
Functions

Function composition: \( f \circ g \)

[first f, then g – intuitive notation, but opposite of the traditional math notation]
From Functions to Relations

\[ f \]

\[ ab?d \rightarrow \text{abcd} \]
From Functions to Relations

ab?d \rightarrow f \rightarrow abcd \rightarrow g \rightarrow \alpha\beta\gamma\delta
From Functions to Relations

ab?d → f → abcd → g → αβγδ

abed → f

Diagram showing the transformation from functions to relations.
From Functions to Relations

ab?d → f → abcd → g → αβγδ
abed → g → αβεδ
From Functions to Relations

\[ \text{f} \rightarrow \text{abcd} \rightarrow \text{g} \rightarrow \alpha\beta\gamma\delta \]
\[ \text{ab?d} \rightarrow \text{abed} \rightarrow \alpha\beta\varepsilon\delta \]
\[ \text{...} \]
From Functions to Relations

\[ \text{ab?d} \rightarrow \text{abcd} \rightarrow \text{abed} \rightarrow \text{abjd} \rightarrow f \]

\[ \text{g} \rightarrow \alpha\beta\gamma\delta \rightarrow \alpha\beta\varepsilon\delta \rightarrow \alpha\beta\varepsilon\delta \rightarrow \ldots \]
From Functions to Relations

f

ab?d → abcd

bed

abjd

g

αβγδ

αβεδ

αβεδ

...
From Functions to Relations

\[ f : \text{ab?d} \rightarrow \text{abcd} \]
\[ g : \text{abcd} \rightarrow \alpha\beta\gamma\delta \]
\[ g : \text{abed} \rightarrow \alpha\beta\epsilon\delta \]
\[ g : \text{abjd} \rightarrow \alpha\beta\epsilon\delta \]
\[ g : \text{abjd} \rightarrow \ldots \]
From Functions to Relations

\[ ab?d \xrightarrow{3} abcd \xrightarrow{4} \alpha\beta\gamma\delta \]
\[ \xrightarrow{2} \text{abed} \xrightarrow{2} \alpha\beta\varepsilon\delta \]
\[ \xrightarrow{6} \text{abjd} \xrightarrow{8} \alpha\beta\varepsilon\delta \]
\[ \ldots \]
From Functions to Relations

Relation composition: $f \circ g$
From Functions to Relations

Relation composition: $f \circ g$
From Functions to Relations

Relation composition: $f \circ g$
From Functions to Relations

Pick min-cost or max-prob output
From Functions to Relations

Often in NLP, all of the functions or relations involved can be described as finite-state machines, and manipulated using standard algorithms.
Inverting Relations

\[ \text{ab?d} \rightarrow f \rightarrow 3 \text{abcd} \rightarrow 2 \text{abed} \rightarrow 6 \text{abjd} \rightarrow \text{g} \rightarrow 4 \alpha\beta\gamma\delta \rightarrow 2 \alpha\beta\varepsilon\delta \rightarrow 8 \alpha\beta\varepsilon\delta \rightarrow \ldots \]
Inverting Relations

\[ f^{-1} \quad g^{-1} \]

- \( ab?d \) to \( abcd \)
- \( abd \) to \( abed \)
- \( ab?d \) to \( abjd \)
- \( 3 \) to \( abcd \)
- \( 2 \) to \( abed \)
- \( 6 \) to \( abjd \)
- \( \alpha\beta\gamma\delta \) to \( 4 \)
- \( \alpha\beta\varepsilon\delta \) to \( 2 \)
- \( \alpha\beta\varepsilon\delta \) to \( 8 \)
- \( \ldots \)
Inverting Relations

\[(f \circ g)^{-1} = g^{-1} \circ f^{-1}\]
Building a lexical transducer

big | clear | clever | ear | fat | ...

Regular Expression Lexicon
Building a lexical transducer

- big
- clear
- clever
- ear
- fat
- ...

Regular Expression
Lexicon

Compiler

Lexicon
FSA
Building a lexical transducer

Regular Expressions
Lexicon

Compiler

Lexicon
FSA

Composed
Rule FSTs

big | clear | clever | ear | fat | ...

Regular Expressions for Rules

slide courtesy of L. Karttunen (modified)
Building a lexical transducer

Regular Expression Lexicon

Compiler

Composition

Lexicon FSA

Composed Rule FSTs

Lexical Transducer (a single FST)

big | clear | clever | ear | fat | ...

one path
Building a lexical transducer

big | clear | clever | ear | fat | ...

Regular Expression
Lexicon

Lexicon
FSA
Building a lexical transducer

- Actually, the lexicon must contain elements like **big +Adj +Comp**
Building a lexical transducer

- Actually, the lexicon must contain elements like `big +Adj +Comp`
- So write it as a more complicated expression:
  \[(\text{big } | \text{ clear } | \text{ clever } | \text{ ear } | \text{ fat } | \ldots) +\text{Adj} (\varepsilon | +\text{Comp} | +\text{Sup}) \quad \leftarrow \text{adjectives}\]
  \[| (\text{ear } | \text{ father } | \ldots) +\text{Noun} (+\text{Sing} | +\text{Pl}) \quad \leftarrow \text{nouns}\]
  \[| \ldots \]

Regular Expression

Lexicon

FSA
Actually, the lexicon must contain elements like big +Adj +Comp

So write it as a more complicated expression:

\[(\text{big } | \text{ clear } | \text{ clever } | \text{ fat } | \ldots) +\text{Adj } (\varepsilon | +\text{Comp} | +\text{Sup}) \quad \leftrightarrow \text{ adjectives}\]

\[| (\text{ear } | \text{ father } | \ldots) +\text{Noun } (+\text{Sing} | +\text{Pl}) \quad \leftrightarrow \text{ nouns}\]

\[| \ldots\]

Q: Why do we need a lexicon at all?
Weighted version of transducer: Assigns a weight to each string pair

Weighted French Transducer

"upper language"

être+IndP +SG + P1
suivre+IndP+SG+P1
suivre+IndP+SG+P2

"lower language"

suis
paie
paye

payer+IndP+SG+P1

.slide courtesy of L. Karttunen (modified)
Constructing Regular Languages
Xerox Regex Notation (Paper)

- Concatenation: EF
- Iteration: E*, E+
- Union: E | F
- Intersection: E & F
- Complementation, minus: ∼E, \x, F-E
- Crossproduct: E .x. F
- Composition: E .o. 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**: \( E^*, E^+ \)

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

- 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 \( E \).

\[
E^+ = \{ e_1 e_2 \ldots e_n : n > 0, e_1 \in E, \ldots e_n \in E \} = EE^*
\]
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)

<table>
<thead>
<tr>
<th>Operation</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>
</tbody>
</table>

**Example:**

$$E \& F = \{w: w \in E \text{ and } w \in F\} = E \cap F$$

- To pick a string in $E \& F$, pick a string from $E$ that is also in $F$.
- To find out whether $w \in E \& F$, check whether $w \in E$ and $w \in F$. 
<table>
<thead>
<tr>
<th>Operator</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>$EF$</td>
<td>Two sets concatenated</td>
</tr>
<tr>
<td>Iteration</td>
<td>$E^*$, $E^+$</td>
<td>Repeatedly combines elements of $E$</td>
</tr>
<tr>
<td>Union</td>
<td>$E</td>
<td>F$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$E \cap F$</td>
<td>Elements common to $E$ and $F$</td>
</tr>
<tr>
<td>Complementation, minus</td>
<td>$\sim E$, $\setminus x$, $F-E$</td>
<td>Set of elements not in $E$</td>
</tr>
</tbody>
</table>

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

$\Sigma$ is set of all letters; so $\Sigma^*$ is set of all strings; $?$* in XFST
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  F
    *  
      &
        +
  concat
```

**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 here 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)

- Concatenation: $EF$
- Iteration: $E^*, E^+$
- Union: $E \cup F$
- Intersection: $E \cap F$
- Complementation, minus: $\sim E$, $\backslash x$, $F - E$
- Crossproduct: $E \times F$

$E \times F = \{(e, f) : e \in E, f \in F\}$

- 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: $\sim E, \setminus x, F-E$
- Crossproduct: $E \cdot x. F$
- Composition: $E \cdot o. F$

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

- 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: $\sim E, \backslash x, F-E$
- 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>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>$EF$</td>
<td></td>
</tr>
<tr>
<td>* +</td>
<td>iteration</td>
<td>$E^*, E^+$</td>
</tr>
<tr>
<td></td>
<td>union</td>
<td>$E</td>
</tr>
<tr>
<td>&amp;</td>
<td>intersection</td>
<td>$E &amp; F$</td>
</tr>
<tr>
<td>~ \ -</td>
<td>complementation, minus</td>
<td>$\sim E, \backslash x, F-E$</td>
</tr>
<tr>
<td>.x.</td>
<td>crossproduct</td>
<td>$E.x.F$</td>
</tr>
<tr>
<td>.o.</td>
<td>composition</td>
<td>$E.o.F$</td>
</tr>
<tr>
<td>.u</td>
<td>upper (input) language</td>
<td>$E.u$ “domain”</td>
</tr>
<tr>
<td>.l</td>
<td>lower (output) language</td>
<td>$E.l$ “range”</td>
</tr>
</tbody>
</table>
Finite-State Programming
## Finite-state “programming”

<table>
<thead>
<tr>
<th>Function</th>
<th>Function on (set of) strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>Regular expression</td>
</tr>
<tr>
<td>Object code</td>
<td>Finite state machine</td>
</tr>
<tr>
<td>Compiler</td>
<td>Regexp compiler</td>
</tr>
<tr>
<td>Optimization of object code</td>
<td>Determinization, minimization, pruning</td>
</tr>
</tbody>
</table>
## Finite-state “programming”

<table>
<thead>
<tr>
<th>Function composition</th>
<th>(Weighted) composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-order function Operator</td>
<td></td>
</tr>
<tr>
<td>Function inversion (available in Prolog) Function inversion</td>
<td></td>
</tr>
<tr>
<td>Structured programming</td>
<td>Ops + small regexps</td>
</tr>
</tbody>
</table>
Finite-state "programming"

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Apply to set of strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nondeterminism</td>
<td>Nondeterminism</td>
</tr>
<tr>
<td>Stochasticity</td>
<td>Prob.-weighted arcs</td>
</tr>
</tbody>
</table>
Some Xerox Extensions

\[
\begin{align*}
\$ & \quad \text{containment} \\
\Rightarrow & \quad \text{restriction} \\
\rightarrow @\rightarrow & \quad \text{replacement}
\end{align*}
\]

Make it easier to describe complex languages and relations without extending the formal power of finite-state systems.
Containment
Containment

$[ab*c]$

“Must contain a substring that matches ab*c.”

Accepts xxxacyy
Rejects bcba
Containment

$[ab* c]$

“Must contain a substring that matches $ab* c$.”

Accepts xxxacyy
Rejects bcba
Containment

$\{ab*c\}$

“Must contain a substring that matches $ab*c$.”

Accepts $xxxacyy$
Rejects $bcba$

$?^* \{ab*c\} \ ?^*$

Equivalent expression
Containment

\[
\$[ab^*c]\$

“Must contain a substring that matches \textit{ab}^*\textit{c}.”

Accepts \textbf{xxxacyy}

Rejects \textbf{bcba}

\[?^* \text{[ab}^*\text{c}] ?^*\]

Equivalent expression

Warning: \textit{?} in regexps means “any character at all.”
But \textit{?} in machines means “any character not explicitly mentioned anywhere in the machine.”
Restriction
Restriction

\[ a \Rightarrow b \_ c \]

“Any \textbf{a} must be preceded by \textbf{b} and followed by \textbf{c}.”

Accepts \textbf{bacbbacde}

Rejects \textbf{baca}
Restriction

\[ a \Rightarrow b \_ c \]

“Any \( a \) must be preceded by \( b \) and followed by \( c \).”

Accepts \( \text{bacbbacde} \)

Rejects \( \text{baca} \)
Restriction

\[ a \implies b \_ c \]

“Any \( a \) must be preceded by \( b \) and followed by \( c \).”

Accepts \textbf{bacbbacde}
Rejects \textbf{baca}

\[ \sim \left[ \sim[?* \ b] \ a \ ?* \right] \ & \sim \left[ ?* \ a \ \sim[ c \ ?*] \right] \]

Equivalent expression
Restriction

\[ a \Rightarrow b \_ c \]

“Any \textit{a} must be preceded by \textit{b} and followed by \textit{c}.”

Accepts \textit{bacbbacde}
Rejects \textit{baca}

\[ \sim [\sim [\text{?* b}] \ _ \ a \ _ \ ?*] \ \& \ \sim [\text{?* a} \ _ \ [c \ _ \ ?*]] \]

Equivalent expression

slide courtesy of L. Karttunen (modified)
Restriction

\[ a \Rightarrow b \_ c \]

"Any \ a \ must be preceded by \ b \ and followed by \ c."

Accepts \textbf{bacbbacdde}

Rejects \textbf{baca}

\[ \sim [ \sim [ ?* b ] a ?* ] \land \sim [ ?* a \sim [ c ?* ] ] \]

Equivalent expression
Replacement
Replacement

$$a \ b \rightarrow b \ a$$

“Replace ‘ab’ by ‘ba’.”

Transduces \underline{abcdbaba} to \underline{bacdbbbaa}
Replacement

a b -> b a

“Replace ‘ab’ by ‘ba’.”

Transduces \textbf{abcdbaba} to \underline{bacdbbaa}

slide courtesy of L. Karttunen (modified)
Replacement

\[ a \ b \rightarrow b \ a \]

“Replace ‘ab’ by ‘ba’.”

Transduces \textbf{abcdbaba} to \textbf{bacdbbbaa}

\[
\left[ \sim[ab]\ [ab.x.\ [ba]] \right]^* \sim[ab]
\]

Equivalent expression
Replacement is Nondeterministic
Replacement is Nondeterministic

\[ a\ b \rightarrow \ b\ a \mid x \]

“Replace ‘ab’ by ‘ba’ or ‘x’, nondeterministically.”

Transduces \underline{abcdbaba} to \{\underline{bacdbbaa}, \underline{bacdbxa}, \underline{xcdbbaa}, \underline{xcdbxa}\}
Replacement is Nondeterministic

\[ \text{Replacement is Nondeterministic} \]

\[[ a b \rightarrow b a | x ] . o. [ x \Rightarrow _c ]\]

“Replace ‘ab’ by ‘ba’ or ‘x’, nondeterministically.”

Transduces \textbf{abcdbaba} to \{\textbf{bacdbbaa}, \textbf{bacdbxa}, \textbf{xcdbbaa}, \textbf{xcdbxa}\}
Replacement is Nondeterministic

\[ \begin{array}{ll}
ab & \rightarrow \ ba \mid x \\
\end{array} \quad \circ \quad \begin{array}{ll}
x & \Rightarrow _c \\
\end{array} \]

“Replace ‘ab’ by ‘ba’ or ‘x’, nondeterministically.”

Transduces \textcolor{red}{\underline{abcdbaba}}

to \{\textcolor{red}{\underline{bacdbbaa}}, \textcolor{red}{\underline{bacdbxa}}, \textcolor{red}{\underline{xcdbbbaa}}, \textcolor{red}{\underline{xcdbxa}}\}
Replacement is Nondeterministic

\[ a \ b \ | \ b \ | \ b \ a \ | \ a \ b \ a \ \rightarrow \ x \]

applied to “aba”

Four overlapping substrings match; we haven’t told it which one to replace so it chooses nondeterministically

\[ \begin{align*}
  & a \ b \ a \\
  & a \ b \ a \\
  & a \ x \ a
\end{align*} \]

slide courtesy of L. Karttunen (modified)
More Replace Operators

- Optional replacement: \[ a \ b \ (-\rightarrow) \ b \ a \]

- Directed replacement
  - guarantees a unique result by constraining the factorization of the input string by
    - Direction of the match (rightward or leftward)
    - Length (longest or shortest)
Left-to-right, Longest-match Replacement

@-> a b | b | b a | a b a @-> x

applied to “aba”

@-> left-to-right, longest match (cf. perl s///)
@-> left-to-right, shortest match
->@ right-to-left, longest match
>@@ right-to-left, shortest match
Using “...” for marking

\(a|e|i|o|u \rightarrow [\ldots]\)
Using “…” for marking

a|e|i|o|u -> [ ... ]

p o t a t o
p[o]t[a]t[o]
Using “…” for marking

\[ a|e|i|o|u \rightarrow [ \ldots ] \]

potato
\[ p[o]t[a]t[o] \]

Note: actually have to write as \( \rightarrow [%[ \ldots %] \) or \( \rightarrow "[" \ldots "]" \) since \([\] \) are parens in the regexp language
Using “…” for marking

\[ a|e|i|o|u \rightarrow [ \ldots ] \]

\texttt{p o t a t o}
\texttt{p[o]t[a]t[o]}

Which way does the FST transduce potatoe?

\texttt{p o t a t o e}
\texttt{p[o]t[a]t[o][e]} \quad \text{vs.} \quad \texttt{p o t a t o e}
\texttt{p[o]t[a]t[o][e]}
Using “…” for marking

\[ a|e|i|o|u \rightarrow [ \ldots ] \]

\textit{p o t a t o\, p[o]t[a]t[o]} vs. \textit{p o t a t o\, p[o]t[a]t[o]e} vs. \textit{p o t a t o\, p[o]t[a]t[o]e}

Which way does the FST transduce potatoe?

How would you change it to get the other answer?
Example: Finnish Syllabification
Example: Finnish Syllabification

define C [ b | c | d | f ... 
define V [ a | e | i | o | u | y | ä | ...
Example: Finnish Syllabification

\[\text{define } C \[ b | c | d | f \ldots \]
\[\text{define } V \[ a | e | i | o | u | y | ä | \ldots \]

\[[C^* V+ C^*] \rightarrow \ldots "-" \mid \_ [C V]\)

“Insert a hyphen after the longest instance of the \(C^* V+ C^*\) pattern in front of a \(C V\) pattern.”
Example: Finnish Syllabification

\[
\text{define } C [ b \mid c \mid d \mid f \ldots \\
\text{define } V [ a \mid e \mid i \mid o \mid u \mid y \mid ä \mid \ldots}
\]

\[
[C^* \; V^+ \; C^*] \rightarrow \ldots \\ \\
"\text{"Insert a hyphen after the longest instance of the } C^* \; V^+ \; C^* \text{ pattern in front of a } C \; V \text{ pattern.}\"
\]

\[
\text{struktuuralismi}
\]

\[
\text{strukt-ту-ra-lis-mi}
\]
Example: Finnish Syllabification

```
define C [ b | c | d | f ... 
define V [ a | e | i | o | u | y | ä | ... 

[C* V+ C*] @-> ... "-" || _ [C V]
```

“Insert a hyphen after the longest instance of the C* V+ C* pattern in front of a C V pattern.”

strukturaalismi
strukt-tu-raa-lis-mi
Conditional Replacement
Conditional Replacement

A -> B

Replacement

L _ R

Context

The relation that replaces A by B between L and R leaving everything else unchanged.
Conditional Replacement

A -> B

Replacement

L _ R

Context

The relation that replaces A by B between L and R leaving everything else unchanged.

Sources of complexity:

- Replacements and contexts may overlap
- Alternative ways of interpreting “between left and right.”
Hand-Coded Example: Parsing Dates

Today is [Tuesday, July 25, 2000].
Hand-Coded Example: Parsing Dates

Today is [Tuesday, July 25, 2000].

Today is Tuesday, [July 25, 2000].
Today is [Tuesday, July 25], 2000.
Today is Tuesday, [July 25], 2000.
Today is [Tuesday], July 25, 2000.

Best result
Bad results
Hand-Coded Example: Parsing Dates

Today is [Tuesday, July 25, 2000].

Today is Tuesday, [July 25, 2000].
Today is [Tuesday, July 25], 2000.
Today is Tuesday, [July 25], 2000.
Today is [Tuesday], July 25, 2000.

Need left-to-right, longest-match constraints.
Source code: Language of Dates
Source code: Language of Dates

Day = Monday | Tuesday | ... | Sunday
Month = January | February | ... | December
Date = 1 | 2 | 3 | ... | 31
Year = %0To9 (%0To9 (%0To9 (%0To9))) - %0?*

from 1 to 9999
Source code: Language of Dates

Day = Monday | Tuesday | ... | Sunday
Month = January | February | ... | December
Date = 1 | 2 | 3 | ... | 31
Year = %0To9 (%0To9 (%0To9 (%0To9))) - %0?* from 1 to 9999

AllDates = Day | (Day "", ") Month " " Date ("", ") Year)
Object code:
All Dates from 1/1/1 to 12/31/9999

13 states, 96 arcs
29 760 007 date expressions
Object code:
All Dates from 1/1/1 to 12/31/9999

actually represents 7 arcs, each labeled by a string

13 states, 96 arcs
29 760 007 date expressions
Parser for Dates
Parser for Dates

AllDates @-> "[DT " ... "]"
Parser for Dates

\texttt{AllDates \@-> \"[DT \ldots \"]\}"

Compiles into an unambiguous transducer (23 states, 332 arcs).
Parser for Dates

AllDates @ -> "[DT " ... " ]"

Compiles into an unambiguous transducer (23 states, 332 arcs).

Today is [DT Tuesday, July 25, 2000] because yesterday was [DT Monday] and it was [DT July 24] so tomorrow must be [DT Wednesday, July 26] and not [DT July 27] as it says on the program.
Parser for Dates

AllDates @-> "[DT " ... "]"

Compiles into an unambiguous transducer (23 states, 332 arcs).

Today is [DT Tuesday, July 25, 2000] because yesterday was [DT Monday] and it was [DT July 24] so tomorrow must be [DT Wednesday, July 26] and not [DT July 27] as it says on the program.
Problem of Reference

Valid dates
- Tuesday, July 25, 2000
- Tuesday, February 29, 2000
- Monday, September 16, 1996

Invalid dates
- Wednesday, April 31, 1996
- Thursday, February 29, 1900
- Tuesday, July 26, 2000
Refinement by Intersection
Refinement by Intersection

AllDates
Refinement by Intersection

MaxDays In Month

```
" 31" => Jan|Mar|May|...

" 30" => Jan|Mar|Apr|...
```
Refinement by Intersection

MaxDays In Month

" 31" => Jan|Mar|May|... _
" 30" => Jan|Mar|Apr|... _

Xerox contextual restriction operator
Refinement by Intersection

MaxDays In Month

" 31" => Jan|Mar|May|...
" 30" => Jan|Mar|Apr|...

Xerox contextual restriction operator

Q: Why do these rules start with spaces? (And is it enough?)
Refinement by Intersection

MaxDays In Month

" 31" => Jan|Mar|May|... _
" 30" => Jan|Mar|Apr|... _

LeapYears
Feb 29, => _ ...

Xerox contextual restriction operator

Q: Why do these rules start with spaces? (And is it enough?)
Refinement by Intersection

MaxDays In Month

" 31" => Jan|Mar|May|...
" 30" => Jan|Mar|Apr|...

LeapYears

Feb 29, => ... 

Q: Why does this rule end with a comma?
Q: Can we write the whole rule?

Xerox contextual restriction operator

Q: Why do these rules start with spaces? (And is it enough?)
Refinement by Intersection

MaxDays In Month

"31" => Jan|Mar|May|... 
"30" => Jan|Mar|Apr|... 

LeapYears
Feb 29, => _ ...

Q: Why do these rules start with spaces? (And is it enough?)

Q: Why does this rule end with a comma?
Q: Can we write the whole rule?
Refinement by Intersection

MaxDays In Month

```
" 31" => Jan|Mar|May|... _
" 30" => Jan|Mar|Apr|... _
```

Valid Dates

WeekdayDate

LeapYears

Feb 29, => _ ...

Q: Why does this rule end with a comma?
Q: Can we write the whole rule?

Xerox contextual restriction operator

Q: Why do these rules start with spaces? (And is it enough?)
Defining Valid Dates

\[ \text{AllDates} \& \text{MaxDaysInMonth} \& \text{LeapYears} \& \text{WeekdayDates} = \text{ValidDates} \]

- **AllDates**: 13 states, 96 arcs
  - 29,760,007 date expressions

- **ValidDates**: 805 states, 6472 arcs
  - 7,307,053 date expressions

Slide courtesy of L. Karttunen
Parser for Valid and Invalid Dates

[AllDates - ValidDates] @-> "[ID " ... "]"

ValidDates @-> "[VD " ... "]"

2688 states,
20439 arcs

Today is [VD Tuesday, July 25, 2000],
valid date
not [ID Tuesday, July 26, 2000].
invalid date
Parser for Valid and Invalid Dates

\[
\text{[AllDates} - \text{ValidDates}] \rightarrow \text{``[ID } \ldots \text{``]''}
\]

\[
\text{ValidDates} \rightarrow \text{``[VD } \ldots \text{``]''}
\]

Comma creates a single FST that does left-to-right longest match against either pattern

Today is [VD Tuesday, July 25, 2000], valid date
not [ID Tuesday, July 26, 2000], invalid date
More Engineering Applications
More Engineering Applications

- Markup
More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)
More Engineering Applications

- **Markup**
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)

- **Translation**
More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)

- Translation
  - Spelling correction / edit distance
More Engineering Applications

- **Markup**
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)

- **Translation**
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
More Engineering Applications

- **Markup**
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)

- **Translation**
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
  - Speech
More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)

- Translation
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
  - Speech
  - Transliteration / back-transliteration
More Engineering Applications

- **Markup**
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)

- **Translation**
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
  - Speech
  - Transliteration / back-transliteration
  - Machine translation?
More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)

- Translation
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
  - Speech
  - Transliteration / back-transliteration
  - Machine translation?

- Learning …
Input: Bridgestone Sports Co. said Friday it has set up a joint venture in Taiwan with a local concern and a Japanese trading house to produce golf clubs to be shipped to Japan. The joint venture, Bridgestone Sports Taiwan Co., capitalized at 20 million new Taiwan dollars, will start production in January 1990 with ...

Output:

Relationship: TIE-UP
Entities: “Bridgestone Sports Co.”
“A local concern”
“A Japanese trading house”
Joint Venture Company: “Bridgestone Sports Taiwan Co.”
Amount: NT$20000000
FASTUS: Successive Markups
(details on subsequent slides)

Tokenization

Multiwords

Basic phrases (noun groups, verb groups ...)

Complex phrases

Semantic Patterns

Merging different references
FASTUS: Tokenization

- Spaces, hyphens, etc.
- wouldn’t → would not
- their → them ‘s
- company. → company .
  but
- Co. → Co.
FASTUS: Multiwords

- “set up”
- “joint venture”
- “San Francisco Symphony Orchestra,”
  “Canadian Opera Company”

... use a specialized regexp to match musical groups.
... what kind of regexp would match company names?
Company Name: Bridgestone Sports Co.
Verb Group: said
Noun Group: Friday
Noun Group: it
Verb Group: had set up
Noun Group: a joint venture
Preposition: in
Location: Taiwan
Preposition: with
Noun Group: a local concern
FASTUS: Noun Groups

Build FSA to recognize phrases like
  approximately 5 kg
  more than 30 people
  the newly elected president
  the largest leftist political force
  a government and commercial project

Use the FSA for left-to-right longest-match markup

What does FSA look like? See next slide ...
FASTUS: Noun Groups

Described with a kind of non-recursive CFG ...
(a regexp can include names that stand for other regexps)

NG → Pronoun | Time-NP | Date-NP
NG → (Det) (Adjs) HeadNouns
...
Adjs → sequence of adjectives maybe with commas, conjunctions, adverbs
...
Det → DetNP | DetNonNP
DetNP → detailed expression to match “the only five, another three, this, many, hers, all, the most ...”
...
FASTUS: Semantic patterns

BusinessRelationship =
  NounGroup(Company/ies) VerbGroup(Set-up) NounGroup(JointVenture) with NounGroup(Company/ies) | ...

ProductionActivity =
  VerbGroup(Produce) NounGroup(Product)

NounGroup(Company/ies) → NounGroup & ...
  is made easy by the processing done at a previous level

Use this for spotting references to put in the database.
Weighted FSMs
Function from strings to ...

Acceptors (FSAs)

Unweighted

Weighted

Transducers (FSTs)

Unweighted

Weighted

\( a : x \quad c : z \)

\( \varepsilon : y \quad \varepsilon : y / .5 \)

\( a / .5 \quad c / .7 \)

\( \varepsilon / .5 \quad .3 \)

\( \varepsilon / .5 \quad .3 \)

\( c : z / .7 \)

\( \varepsilon : y / .5 \)
Function from strings to ...

Acceptors (FSAs)  Transducers (FSTs)

Unweighted  Weighted

{false, true}  a

ε  a

ε  c

ε  a:x

ε:y  c:z

c:z/.7  c/.7

a/.5  a:.5

ε/.5  ε:.3

ε/.5  ε:y/.5

ε/.3

{false, true}
Function from strings to ...

Acceptors (FSAs)

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>a</td>
<td>ε</td>
</tr>
<tr>
<td>Start</td>
<td>c</td>
<td>ε</td>
</tr>
</tbody>
</table>

Transducers (FSTs)

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>a</td>
<td>ε</td>
</tr>
<tr>
<td>Start</td>
<td>c</td>
<td>ε</td>
</tr>
</tbody>
</table>

Unweighted

- a: x
- c: z

Weighted

- a: x
- c: z

{false, true}
Function from strings to ...

**Acceptors (FSAs)**

- **Unweighted**
  - a \(\xrightarrow{\varepsilon} \cdot \)
  - c

- **Weighted**
  - a \(\xrightarrow{\varepsilon/0.5} \cdot \cdot \cdot 0.3 \)
  - c \(\xrightarrow{\varepsilon/0.5} \cdot \cdot \cdot 0.3 \)

**Transducers (FSTs)**

- **Unweighted**
  - a \(\xrightarrow{x} \cdot \)
  - c

- **Weighted**
  - a \(\xrightarrow{x/0.5} \cdot \cdot \cdot 0.3 \)
  - c \(\xrightarrow{z/0.7} \cdot \cdot \cdot 0.3 \)

**Values:**
- a, c: x, z
- \(\varepsilon: y, \cdot \)
- 
- Numbers: true, false
- Weights: 0.5, 0.7
Function from strings to ...

Acceptors (FSAs)

Unweighted

- \( a \rightarrow \varepsilon \rightarrow c \)
- \( \{\text{false, true}\} \)

Weighted

- \( a \rightarrow \varepsilon \rightarrow c \)
- \( a/0.5 \rightarrow \varepsilon/0.5 \rightarrow 0.3 \)

Transducers (FSTs)

- \( a:x \rightarrow \varepsilon:y \rightarrow c:z \)
- \( \text{strings} \)

- \( a:x/0.5 \rightarrow \varepsilon:y/0.5 \rightarrow 0.3 \)
- \( \text{(string, num) pairs} \)

- \( c:z/0.7 \)
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 \textit{weighted} language [or relation], we ask: What \textit{weight} do you assign to this string [or string pair]?

- Pick a semiring: all our \textit{weights} will be in that semiring.
  - What?!
## Semirings

<table>
<thead>
<tr>
<th>Category</th>
<th>Set</th>
<th>⊕</th>
<th>⊗</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prob</td>
<td>( \mathbb{R} )</td>
<td>+</td>
<td>( \times )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Max</td>
<td>( \mathbb{R} )</td>
<td>max</td>
<td>( \times )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Log</td>
<td>( \mathbb{R} \cup { \pm \infty } )</td>
<td>( \log^+ )</td>
<td>+</td>
<td>(-\infty)</td>
<td>0</td>
</tr>
<tr>
<td>“Tropical”</td>
<td>( \mathbb{R} \cup { \pm \infty } )</td>
<td>max</td>
<td>+</td>
<td>(-\infty)</td>
<td>0</td>
</tr>
<tr>
<td>Shortest path</td>
<td>( \mathbb{R} \cup { \pm \infty } )</td>
<td>min</td>
<td>+</td>
<td>(\infty)</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>{0, 1}</td>
<td>( \lor )</td>
<td>( \land )</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>String</td>
<td>( \Sigma )</td>
<td>longest common prefix</td>
<td>concat</td>
<td>(\infty)</td>
<td>(\varepsilon)</td>
</tr>
</tbody>
</table>
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.
  - Don’t panic! We will cover this again when we get to HMMs and parsing.
  - The unweighted case is the boolean semiring \{true, false\}.
  - If a string is not in the language, it has weight $\bot$.
  - 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: $E \cdot x \cdot F$
- Composition: $E \cdot o \cdot F$
- Upper (input) language: $E.u$ “domain”
- Lower (output) language: $E.l$ “range”
### Which Semiring Operators are Needed?

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td></td>
<td>EF</td>
</tr>
<tr>
<td>* +</td>
<td>iteration</td>
<td>E*, E+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to sum over 2 choices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complementation, minus</td>
</tr>
<tr>
<td>~ \ -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>intersection</td>
<td>E &amp; F</td>
</tr>
<tr>
<td>.x.</td>
<td>crossproduct</td>
<td>E .x. F</td>
</tr>
<tr>
<td>.o.</td>
<td>composition</td>
<td>E .o. F</td>
</tr>
<tr>
<td>.u</td>
<td>upper (input) language</td>
<td>E.u</td>
</tr>
<tr>
<td>.l</td>
<td>lower (output) language</td>
<td>E.l</td>
</tr>
<tr>
<td></td>
<td>“domain”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“range”</td>
<td></td>
</tr>
</tbody>
</table>
### Which Semiring Operators are Needed?

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\oplus$</td>
<td>Sum over 2 choices</td>
<td>$E \oplus F$</td>
</tr>
<tr>
<td>$\otimes$</td>
<td>Combine the matches against $E$ and $F$</td>
<td>$E \otimes F$</td>
</tr>
<tr>
<td>$\cap$</td>
<td>Intersection</td>
<td>$E \cap F$</td>
</tr>
<tr>
<td>$\cup$</td>
<td>Union</td>
<td>$E \cup F$</td>
</tr>
<tr>
<td>$\sim \setminus -$</td>
<td>Complementation, minus</td>
<td>$\sim E, \setminus x, E \setminus F$</td>
</tr>
<tr>
<td>$\times$</td>
<td>Crossproduct</td>
<td>$E \times F$</td>
</tr>
<tr>
<td>$\circ$</td>
<td>Composition</td>
<td>$E \circ F$</td>
</tr>
<tr>
<td>$u$</td>
<td>Upper (input) language</td>
<td>$E.u$ “domain”</td>
</tr>
<tr>
<td>$l$</td>
<td>Lower (output) language</td>
<td>$E.l$ “range”</td>
</tr>
</tbody>
</table>
Common Regular Expression Operators (in XFST notation)

union $E | F$

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

- 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?

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\oplus$</td>
<td>Concatenation</td>
<td>EF</td>
</tr>
<tr>
<td>$\ast$</td>
<td>Iteration</td>
<td>$E^*, E^+$</td>
</tr>
<tr>
<td>$\mid$</td>
<td>Union</td>
<td>$E \mid F$</td>
</tr>
<tr>
<td>$\sim \backslash -$</td>
<td>Complementation, minus</td>
<td>$\sim E, \backslash x, E-F$</td>
</tr>
<tr>
<td>$&amp;$</td>
<td>Intersection</td>
<td>$E &amp; F$</td>
</tr>
<tr>
<td>$.x.$</td>
<td>Crossproduct</td>
<td>$E .x. F$</td>
</tr>
<tr>
<td>$.o.$</td>
<td>Composition</td>
<td>$E .o. F$</td>
</tr>
<tr>
<td>$.u$</td>
<td>Upper (input) language</td>
<td>$E.u$</td>
</tr>
<tr>
<td>$.l$</td>
<td>Lower (output) language</td>
<td>$E.l$</td>
</tr>
</tbody>
</table>
Which Semiring Operators are Needed?

concatenation

\[
\begin{align*}
\ast & \quad \text{iteration} \\
+ & \quad \text{E}^*, \; E^+ \\
| & \quad \text{E} \mid F \\
\sim \setminus - & \quad \text{E} \sim, \; \backslash x, \; \text{E-F} \\
\& & \quad \text{E} \& \; \text{F} \\
.x. & \quad \text{E} . \times . \; \text{F} \\
.o. & \quad \text{E} . \circ . \; \text{F} \\
.u & \quad \text{E} . \cdot u \quad \text{“domain”} \\
.l & \quad \text{E} . \cdot l \quad \text{“range”}
\end{align*}
\]

\[
\oplus \quad \text{to sum over 2 choices} \\
\otimes \quad \text{to combine the matches against E and F}
\]
Which Semiring Operators are Needed?

- Concatenation: $EF$
- Iteration: $E^*, E^+$
- Union: $E \mid F$
- Complementation, minus: $\sim E, \backslash x, E-F$
- Intersection: $E \& F$
- Crossproduct: $E \times F$
- Composition: $E \circ F$
- Upper (input) language: $E.u$ “domain”
- Lower (output) language: $E.l$ “range”
Which Semiring Operators are Needed?

- Concatenation
- Iteration

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

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

\[
(\text{EF})(w) = \bigoplus_{e,f \text{ such that } w=ef} (E(e) \otimes F(f))
\]
Which Semiring Operators are Needed?

- Concatenation
- Iteration
- Union
- Complementation, minus
- Intersection
- Crossproduct
- Composition
- Upper (input) language
- Lower (output) language

EF

- Need both $\oplus$ and $\otimes$
- $E^\ast$, $E^+$
- $E \mid F$
- $\sim E$, $\setminus x$, $E-F$
- $E \& F$
- $E .x. F$
- $E .o. F$
- $E.u$ “domain”
- $E.l$ “range”
Which Semiring Operators are Needed?

- Concatenation
- Iteration
- Union
- Complementation, minus
- Intersection
- Crossproduct
- Composition
- Upper (input) language
- Lower (output) language

EF
E*, E+
E | F
~E, \x, E-F
E & F
E .x. F
E .o. F
E.u “domain”
E.l “range”
Which Semiring Operators are Needed?

- Concatenation \( EF \) need both \( \oplus \) and \( \otimes \)
- Iteration \( E^*, E^+ \)
- Union \( E \mid F \)
- Complementation, minus \( \sim E, \setminus x, E-F \)
- Intersection \( E \& F \)
- Crossproduct \( E \cdot x \cdot F \)
- Composition \( E \cdot o \cdot F \)
- Upper (input) language \( + E.u \) “domain”
- Lower (output) language \( + E.l \) “range”
Definition of FSTs
Definition of FSTs

- [Red material shows differences from FSAs.]
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 $\Sigma^*$).
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.]
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.
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 $\Delta$
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 \( d: Q \times \Sigma? \to 2^Q \)
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?

concatenation: EF
iteration: E*, E+
union: E | F
complementation, minus: ~E, \x, E-F
intersection: E & F
crossproduct: E .x. F
composition: E .o. F
upper (input) language: E.u “domain”
lower (output) language: E.l “range”
Concatenation

dexample courtesy of M. Mohri
Concatenation

equivalent courtesy of M. Mohri
Union

example courtesy of M. Mohri
Union

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

\[
\begin{array}{c}
0 \xrightarrow{a:a/0.1, a:b/0.2} 1 \xrightarrow{b:b/0.3} 2 \xrightarrow{a:b/0.5} 3/0.7 \\
\end{array}
\]

\[
\begin{array}{c}
0 \xrightarrow{a:a/0.1, a:b/0.2} 1 \xrightarrow{b:b/0.3} 2 \xrightarrow{a:b/0.5} 3/0.7 \\
\end{array}
\]
Closure (this example has outputs too)

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)

\[ u = \]

- **0**
  - **a**: 0.1
  - **a**: 0.2

- **1**
  - **b**: 0.3
  - **b**: 0.4

- **2**
  - **a**: 0.5
  - Self-loop: **a**: 0.6

- **3**
  - Self-loop: **a**: 0.7

Example courtesy of M. Mohri
Upper language (domain)

Similarly construct lower language .l
Upper language (domain)

Similarly construct lower language. Also called input & output languages.
Reversal
Reversal
Reversal

example courtesy of M. Mohri
Inversion

example courtesy of M. Mohri
Inversion

\[
\begin{align*}
0 \xrightarrow{a:a/0.1} 1 & \xrightarrow{b:b/0.3} 2 & \xrightarrow{a:b/0.5} 3/0.7 \\
1 & \xrightarrow{b:b/0.4} 2 & \xrightarrow{a:b/0.6} 3/0.7 \\
\end{align*}
\]
Inversion

\[
\begin{array}{c}
\text{0} \\
\text{a:a/0.1} \\
\text{a:b/0.2}
\end{array}
\quad \Rightarrow
\quad
\begin{array}{c}
\text{1} \\
\text{b:b/0.3}
\end{array}
\quad \Rightarrow
\quad
\begin{array}{c}
\text{2} \\
\text{a:b/0.5}
\end{array}
\quad \Rightarrow
\quad
\begin{array}{c}
\text{3/0.7}
\end{array}
\]

\[
\begin{array}{c}
\text{0} \\
\text{a:a/0.1} \\
\text{b:a/0.2}
\end{array}
\quad \Rightarrow
\quad
\begin{array}{c}
\text{1} \\
\text{b:b/0.3}
\end{array}
\quad \Rightarrow
\quad
\begin{array}{c}
\text{2} \\
\text{b:a/0.5}
\end{array}
\quad \Rightarrow
\quad
\begin{array}{c}
\text{3/0.7}
\end{array}
\]

\text{example courtesy of M. Mohri}
Inversion

example courtesy of M. Mohri
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
Intersection

Example adapted from M. Mohri
Intersection

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

(example adapted from M. Mohri)
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 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 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
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
Paths 12 and 12 both accept fat
So must the new machine: along path 1,1 2,2
Intersection

fat/0.5

0 → pig/0.3 → 1 → eats/0 → 2/0.8

sleeps/0.6

pig/0.4

0 → 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 → sleeps/1.9 → 2,2/1.3
Intersection

fat/0.5

pig/0.3

0 ➔ 1 ➔ 2

eats/0

sleeps/0.6

pig/0.4

fat/0.2

0 ➔ 1 ➔ 2

eats/0.6

sleeps/1.3

pig/0.7

fat/0.7

0,0 ➔ 0,1 ➔ 1,1 ➔ 2,2

eats/0.6

sleeps/1.9

2,0/0.8

2,2/1.3
What Composition Means

\[ f : \text{ab?d} \rightarrow \text{abcd} \]
What Composition Means

ab?d → f → abcd → g → αβγδ
What Composition Means

\[
\begin{align*}
ab?d & \rightarrow f \\
& \rightarrow abcd \\
& \rightarrow g \\
& \rightarrow \alpha\beta\gamma\delta
\end{align*}
\]
What Composition Means

\[
\begin{align*}
ab?d & \rightarrow f \\
abcd & \rightarrow g \\
abed & \rightarrow \alpha\beta\gamma\delta \\
\end{align*}
\]
What Composition Means

ab?d → abcd → αβεδ → αβγδ

abed → αβεδ → αβ∈δ → ...

f → g
What Composition Means

\[ \text{ab?d} \xrightarrow{f} \text{abcd} \xrightarrow{g} \alpha\beta\gamma\delta \]
\[ \text{abed} \xrightarrow{g} \alpha\beta\epsilon\delta \]
\[ \text{abjd} \xrightarrow{g} \alpha\beta\epsilon\delta \]
\[ \ldots \]
What Composition Means

ab?d → f → abcd → g → αβγδ
   |     |   |     |   |   |   |
abed → |  abjd → | αβεδ → | αβ∈δ → | ...
What Composition Means

ab?d → f → 3 → abcd → g → αβγδ
2 → abed → αβεδ
6 → abjd → αβ∈δ
...
What Composition Means

\[ \text{ab?d} \to \text{f} \to 3 \to \text{abcd} \to \text{g} \to 4 \to \alpha \beta \gamma \delta \]

\[ \text{ab?d} \to \text{f} \to 2 \to \text{abed} \to \text{g} \to 2 \to \alpha \beta \epsilon \delta \]

\[ \text{ab?d} \to \text{f} \to 6 \to \text{abjd} \to \text{g} \to 8 \to \alpha \beta \epsilon \delta \]

\[ \text{ab?d} \to \text{f} \to \text{...} \]

\[ \alpha \beta \epsilon \delta \]
What Composition Means

Relation composition: $f \circ g$

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

$2+2 \alpha \beta \varepsilon \delta$

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

...
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 \}

\[ \begin{align*}
\text{f} \quad \text{ab?d} & \rightarrow 3 \rightarrow \text{abcd} \\
& \quad 2 \rightarrow \text{abed} \\
& \quad 6 \rightarrow \text{abjd} \\
\text{g} \quad \text{abcd} & \rightarrow 4 \rightarrow \alpha\beta\gamma\delta \\
& \quad 2 \rightarrow \alpha\beta\varepsilon\delta \\
& \quad 8 \rightarrow \alpha\beta\varepsilon\delta
\end{align*} \]
Relation = set of pairs

\[
\begin{align*}
\text{relation 1:} & \quad \{ab?d \to abcd, ab?d \to abed, ab?d \to abjd\} \\
\text{relation 2:} & \quad \{abcd \to \alpha\beta\gamma\delta, abed \to \alpha\beta\varepsilon\delta, abed \to \alpha\beta\in\delta\}
\end{align*}
\]

\( f \):

- \( ab?d \to abcd \) \rightarrow 3
- \( ab?d \to abed \) \rightarrow 2
- \( ab?d \to abjd \) \rightarrow 6

\( g \):

- \( abcd \to \alpha\beta\gamma\delta \) \rightarrow 4
- \( abed \to \alpha\beta\varepsilon\delta \) \rightarrow 2
- \( abed \to \alpha\beta\in\delta \) \rightarrow 8

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

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

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

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

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

Intersection

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

Composition

Wilbur: pig/0.3

\[ 0 \xrightarrow{\text{Wilbur: pig}/0.3} 1 \quad .0. \quad 1 \xrightarrow{\text{pig: pink}/0.4} 1 \quad = \quad 0,1 \xrightarrow{\text{Wilbur: pink}/0.7} 1,1 \]
Intersection vs. Composition

Intersection mismatch

\[ \text{pig/0.3} \rightarrow 0 \quad \& \quad 1 \quad = \quad 0,1 \]

Composition mismatch

\[ \text{Wilbur: pig/0.3} \rightarrow 0 \quad \& \quad \text{elephant: gray/0.4} \quad = \quad 0,1 \]
Composition

example courtesy of M. Mohri
Composition

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

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

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

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

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

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

\[
\begin{align*}
\text{b:b \ .o. a:b} &= \text{nothing} \\
&= \text{(since intermediate symbol doesn't match)}
\end{align*}
\]
Composition

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

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

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

where \( x, y, z \) are strings
Composition with Sets
We’ve defined $A \circ B$ where both are FSTs
Composition with Sets

- We’ve defined $A \circ B$ where both are FSTs
- Now extend definition to allow one to be a FSA
Composition with Sets

- We’ve defined A .o. B where both are FSTs
- Now extend definition to allow one to be a FSA
- Two relations (FSTs):
  \[ A \circ B = \{ x \rightarrow z : \exists y (x \rightarrow y \in A \text{ and } y \rightarrow z \in B) \} \]
Composition with Sets

- We’ve defined $A .o. B$ 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:
Composition with Sets

- We’ve defined $A .o. B$ 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:
  \[ A \circ B = \{ x \rightarrow z : x \in A \text{ and } x \rightarrow z \in B \} \]
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):
  \[ A \circ B = \{ x \rightarrow z : \exists y ( x \rightarrow y \in A \text{ and } y \rightarrow z \in B ) \} \]
- Set and relation:
  \[ A \circ B = \{ x \rightarrow z : x \in A \text{ and } x \rightarrow z \in B \} \]
- Relation and set:
  \[ A \circ B = \{ x \rightarrow z : x \rightarrow z \in A \text{ and } z \in B \} \]
Composition with Sets

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

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

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

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

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

- Feed string into Greek transducer:
3 Uses of Set Composition:

- Feed string into Greek transducer:
  - \{\text{abed} \rightarrow \text{abed}\} \cdot \text{Greek} = \{\text{abed} \rightarrow \alpha\beta\varepsilon\delta, \text{abed} \rightarrow \alpha\beta\varepsilon\delta\}
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\}
3 Uses of Set Composition:

- Feed string into Greek transducer:
  - \{\text{abed} \rightarrow \text{abed}\} .o. \text{Greek} = \{\text{abed} \rightarrow \alpha\beta\epsilon\delta, \text{abed} \rightarrow \alpha\beta\epsilon\delta\}
  - \{\text{abed}\} .o. \text{Greek} = \{\text{abed} \rightarrow \alpha\beta\epsilon\delta, \text{abed} \rightarrow \alpha\beta\epsilon\delta\}
  - \{\{\text{abed}\} .o. \text{Greek}\}.l = \{\alpha\beta\epsilon\delta, \alpha\beta\epsilon\delta\}
3 Uses of Set Composition:

- Feed string into Greek transducer:
  - \{abed\rightarrow abed\} \cdot \mathrm{o. \ Greek} = \{abed\rightarrow \alpha\beta\epsilon\delta, abed\rightarrow \alpha\beta\in\delta\}
  - \{abed\} \cdot \mathrm{o. \ Greek} = \{abed\rightarrow \alpha\beta\epsilon\delta, abed\rightarrow \alpha\beta\in\delta\}
  - [\{abed\} \cdot \mathrm{o. \ Greek}]\cdot l = \{\alpha\beta\epsilon\delta, \alpha\beta\in\delta\}

- Feed several strings in parallel:
3 Uses of Set Composition:

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

- Feed several strings in parallel:
  - \{abcd, abed\} \cdot \text{o. Greek}
    = \{abcd \rightarrow \alpha \beta \gamma \delta, abed \rightarrow \alpha \beta \varepsilon \delta, abed \rightarrow \alpha \beta \varepsilon \delta\}
3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - \{abed \rightarrow abed\} .o. Greek = \{abed \rightarrow \alpha\beta\epsilon\delta, abed \rightarrow \alpha\beta\in\delta\}
  - \{abed\} .o. Greek = \{abed \rightarrow \alpha\beta\epsilon\delta, abed \rightarrow \alpha\beta\in\delta\}
  - [[\{abed\} .o. Greek]].l = \{\alpha\beta\epsilon\delta, \alpha\beta\in\delta\}

- **Feed several strings in parallel:**
  - \{abcd, abed\} .o. Greek
    = \{abcd \rightarrow \alpha\beta\gamma\delta, abed \rightarrow \alpha\beta\epsilon\delta, abed \rightarrow \alpha\beta\in\delta\}
  - [[\{abcd, abed\} .o. Greek]].l = \{\alpha\beta\gamma\delta, \alpha\beta\epsilon\delta, \alpha\beta\in\delta\}
3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - \(\{\text{abed} \rightarrow \text{abed}\}.o. \text{Greek} = \{\text{abed} \rightarrow \alpha \beta \epsilon \delta, \text{abed} \rightarrow \alpha \beta \in \delta\}\)
  - \(\{\text{abed}\}.o. \text{Greek} = \{\text{abed} \rightarrow \alpha \beta \epsilon \delta, \text{abed} \rightarrow \alpha \beta \in \delta\}\)
  - \([\{\text{abed}\}.o. \text{Greek}]\).l = \{\alpha \beta \epsilon \delta, \alpha \beta \in \delta\}\)

- **Feed several strings in parallel:**
  - \(\{\text{abcd, abed}\}.o. \text{Greek}\)
    - \(\{\text{abcd, abed}\}.o. \text{Greek} = \{\text{abcd} \rightarrow \alpha \beta \gamma \delta, \text{abed} \rightarrow \alpha \beta \epsilon \delta, \text{abed} \rightarrow \alpha \beta \in \delta\}\)
  - \([\{\text{abcd, abed}\}.o. \text{Greek}]\).l = \{\alpha \beta \gamma \delta, \alpha \beta \epsilon \delta, \alpha \beta \in \delta\}\)

- **Filter result via Noε:** \(\{\alpha \beta \gamma \delta, \alpha \beta \in \delta, \ldots\}\)
3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - \( \{ \text{abed} \rightarrow \text{abed} \} \).o. Greek = \( \{ \text{abed} \rightarrow \alpha \beta \varepsilon \delta, \text{abed} \rightarrow \alpha \beta \varepsilon \delta \} \)
  - \( \{ \text{abed} \} \).o. Greek = \( \{ \text{abed} \rightarrow \alpha \beta \varepsilon \delta, \text{abed} \rightarrow \alpha \beta \varepsilon \delta \} \)
  - \( [\{ \text{abed} \} \).o. Greek].l = \{ \alpha \beta \varepsilon \delta, \alpha \beta \varepsilon \delta \} \)

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

- **Filter result via No\( \varepsilon \) = \{ \alpha \beta \gamma \delta, \alpha \beta \varepsilon \delta, \ldots \} \)
  - \( \{ \text{abcd}, \text{abed} \} \).o. Greek .o. No\( \varepsilon \) = \( \{ \text{abcd} \rightarrow \alpha \beta \gamma \delta, \text{abed} \rightarrow \alpha \beta \varepsilon \delta \} \)
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 $\varepsilon:\varepsilon$?

http://aclweb.org/anthology-new/N/N03/N03-1018.pdf