**Finite-state automata**

Automata are models of computation: they compute languages.

A finite-state automaton is a five-tuple \( \langle Q, q_0, \Sigma, \delta, F \rangle \), where \( \Sigma \) is a finite set of alphabet symbols, \( Q \) is a finite set of states, \( q_0 \in Q \) is the initial state, \( F \subseteq Q \) is a set of final (accepting) states and \( \delta : Q \times \Sigma \times Q \) is a relation from states and alphabet symbols to states.

**Example: Finite-state automaton**

- \( Q = \{ q_0, q_1, q_2, q_3 \} \)
- \( \Sigma = \{ c, a, t, r \} \)
- \( F = \{ q_3 \} \)
- \( \delta = \{ (q_0, c, q_1), (q_1, a, q_2), (q_2, t, q_3), (q_2, r, q_3) \} \)

\[
\begin{array}{c}
q_0 \xrightarrow{c} q_1 \xrightarrow{a} q_2 \xrightarrow{t} q_3
\end{array}
\]

**Finite-state automata**

The reflexive transitive extension of the transition relation \( \delta \) is a new relation, \( \delta' \), defined as follows:

- for every state \( q \in Q \), \( (q, \varepsilon, q) \in \delta' \)
- for every string \( w \in \Sigma^* \) and letter \( a \in \Sigma \), if \( (q, w, q'') \in \delta \) and \( (q', a, q''') \in \delta \) then \( (q, w, a, q''') \in \delta' \).

**Example: Paths**

For the finite-state automaton:

\[
\begin{array}{c}
q_0 \xrightarrow{c} q_1 \xrightarrow{a} q_2 \xrightarrow{t} q_3
\end{array}
\]

\( \delta \) is the following set of triples:

\[
\{ (q_0, \varepsilon, q_0), (q_1, \varepsilon, q_1), (q_2, \varepsilon, q_2), (q_3, \varepsilon, q_3), (q_0, c, q_1), (q_1, a, q_2), (q_2, t, q_3), (q_2, r, q_3), (q_3, a, q_2), (q_1, a, q_3), (q_1, a, q_3), (q_0, c, q_2), (q_0, c, q_3), (q_0, c, q_3) \}
\]
Finite-state automata

A string $w$ is accepted by the automaton $A = (Q, q_0, \Sigma, \delta, F)$ if and only if there exists a state $q_f \in F$ such that $(q_0, w, q_f) \in \delta$.

The language accepted by a finite-state automaton is the set of all string it accepts.

Example: Language
The language of the finite-state automaton:

$$
\begin{array}{c}
q_0 \\
\downarrow a \\
q_1 \\
\downarrow t \\
q_2 \\
\downarrow r \\
q_3
\end{array}
$$

is $\{cat, car\}$.

Finite-state automata

Example: Some finite-state automata

$$
\begin{array}{c}
q_0 \\
\downarrow a \\
q_1
\end{array}
$$

$\{a\}$

Finite-state automata

Example: Some finite-state automata

$$
\begin{array}{c}
q_0
\end{array}
$$

$\{\epsilon\}$
Finite-state automata

Example: Some finite-state automata

\[ a \xrightarrow{} \{ a, aa, aaaa, \ldots \} \]

Finite-state automata

Example: Some finite-state automata

\[ a \xrightarrow{} a^* \]

Finite-state automata

Example: Some finite-state automata

An extension: \( \epsilon \)-moves.

The transition relation \( \delta \) is extended to: \( \delta \subseteq Q \times ( \Sigma \cup \{ \epsilon \} ) \times Q \)

Example: Automata with \( \epsilon \)-moves

The language accepted by the following automaton is \{ \textit{do, undo, done, undone} \}:

\[ Q_6 \xrightarrow{u} Q_1 \xrightarrow{n} Q_2 \xrightarrow{d} Q_3 \xrightarrow{o} Q_4 \xrightarrow{n} Q_5 \xrightarrow{e} Q_6 \]
Finite-state automata

Theorem (Kleene, 1956): The class of languages recognized by finite-state automata is the class of regular languages.

Operations on finite-state automata

- Concatenation
- Union
- Intersection
- Minimization
- Determinization

Finite-state automata

Example: Finite-state automata and regular expressions

\[
\emptyset
\]

\[
a
\]

\[
((c \cdot a) \cdot t)
\]

\[
(((m \cdot e) \cdot (o)^* \cdot w)
\]

\[
((a + (e + (i + (o + u))))^*)
\]

Minimization and determinization

Example: Equivalent automata

\[
A_1
\]

\[
A_2
\]

\[
A_3
\]
Applications of finite-state automata in language processing

Lexicon:

\[\text{go, gone, going} : g \quad o \quad i \quad n \quad g \]

This automaton can then be determinized and minimized:

\[\text{go, gone, going} : g \quad o \quad i \quad n \quad e \]

Regular relations

While regular expressions are sufficiently expressive for some natural language applications, it is sometimes useful to define relations over two sets of strings.

Part-of-speech tagging:

\[
\begin{align*}
\text{I} & \quad \text{know} & \quad \text{some} & \quad \text{new} & \quad \text{tricks} \\
\text{PRON} & \quad \text{V} & \quad \text{DET} & \quad \text{ADJ} & \quad \text{N} \\
\text{said} & \quad \text{the} & \quad \text{Cat} & \quad \text{in} & \quad \text{the} & \quad \text{Hat} \\
\text{V} & \quad \text{DET} & \quad \text{N} & \quad \text{P} & \quad \text{DET} & \quad \text{N}
\end{align*}
\]
Regular relations

Morphological analysis:

<table>
<thead>
<tr>
<th>I-PRON-1-sg</th>
<th>I-P</th>
<th>know</th>
<th>some</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>-PRON-1-sg</td>
<td>-PRON-1-sg</td>
<td>know-V-pres</td>
<td>some-DET-def</td>
<td>new-ADJ</td>
</tr>
<tr>
<td>-PRON-1-sg</td>
<td>-PRON-1-sg</td>
<td>say-V-past</td>
<td>the-DET-def</td>
<td>cat-N-sg</td>
</tr>
<tr>
<td>in</td>
<td>in</td>
<td>the</td>
<td>the-DET-def</td>
<td>hat-N-sg</td>
</tr>
</tbody>
</table>

Finite-state transducers

A finite-state transducer is a six-tuple $\langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle$. Similarly to automata, $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final (or accepting) states, $\Sigma_1$ and $\Sigma_2$ are alphabets: finite sets of symbols, not necessarily disjoint (or different). $\delta : Q \times \Sigma_1 \times \Sigma_2 \times Q$ is a relation from states and pairs of alphabet symbols to states.

Regular relations

Singular-to-plural mapping:

cat hat ox child mouse sheep goose
cats hats oxen children mice sheep geese

Finite-state transducers

Shorthand notation:

Adding $\epsilon$-moves:
Finite-state transducers

The language of a finite-state transducer is a language of pairs: a binary relation over \( \Sigma_1 \times \Sigma_2 \). The language is defined analogously to how the language of an automaton is defined.

\[ T(w) = \{ u \mid (q_0, w, u, q_f) \in \delta \} \text{ for some } f \in F \].

Properties of finite-state transducers

Given a transducer \( \langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle \).

- its underlying automaton is \( \langle Q, q_0, \Sigma_1 \times \Sigma_2, \delta', F \rangle \), where \( (q_1, (a, b), q_2) \in \delta' \) iff \( (q_1, a, b, q_2) \in \delta \)
- its upper automaton is \( \langle Q, q_0, \Sigma_1, \delta_1, F \rangle \), where \( (q_1, a, q_2) \in \delta_1 \) iff for some \( b \in \Sigma_2 \), \( (q_1, a, b, q_2) \in \delta \)
- its lower automaton is \( \langle Q, q_0, \Sigma_2, \delta_2, F \rangle \), where \( (q_1, b, q_2) \in \delta_2 \) iff for some \( a \in \Sigma_1 \), \( (q_1, a, b, q_2) \in \delta \)

Finite-state transducers

Example: The uppercase transducer

\[ a : A, b : B, c : C, \ldots \]

Example: English-to-French

\[ c : c \quad o : h \quad i : ? \quad e : e \quad n : n \]

Properties of finite-state transducers

A transducer \( T \) is functional if for every \( w \in \Sigma_1 \), \( T(w) \) is either empty or a singleton.

Transducers are closed under union: if \( T_1 \) and \( T_2 \) are transducers, there exists a transducer \( T \) such that for every \( w \in \Sigma_1 \), \( T(w) = T_1(w) \cup T_2(w) \).

Transducers are closed under inversion: if \( T \) is a transducer, there exists a transducer \( T^{-1} \) such that for every \( w \in \Sigma_1 \), \( T^{-1}(w) = \{ u \in \Sigma_2 \mid w \in T(u) \} \).

The inverse transducer is \( \langle Q, q_0, \Sigma_2, \Sigma_1, \delta^{-1}, F \rangle \), where \( (q_1, a, b, q_2) \in \delta^{-1} \) iff \( (q_1, b, a, q_2) \in \delta \).
Properties of finite-state transducers

Transducers are closed under composition: if $T_1$ and $T_2$ are transducers, there exists a transducer $T$ such that for every $w \in \Sigma^*$, $T(w) = T_1(T_2(w))$.

The number of states in the composition transducer might be $|Q_1 \times Q_2|$.

Properties of finite-state transducers

- Computationally efficient
- Denote regular relations
- Closed under concatenation, Kleene-star, union
- Not closed under intersection (and hence complementation)
- Closed under composition
- Weights

Introduction to XFST

- XFST is an interface giving access to finite-state operations (algorithms such as union, concatenation, iteration, intersection, composition etc.)
- XFST includes a regular expression compiler
- The interface of XFST includes a lookup operation (apply up) and a generation operation (apply down)
- The regular expression language employed by XFST is an extended version of standard regular expressions
Introduction to XFST

a  a simple symbol
b  a concatenation of three symbols
[c a t]  grouping brackets
?  denotes any single symbol
‘+Noun’ single symbol with multicharacter print name
%+Noun single symbol with multicharacter print name
cat  a single multicharacter symbol
{cat}  equivalent to [c a t]

Introduction to XFST

A*  Kleene-star
A+  one or more iterations
?*  the universal language
~A  the complement of A; equivalent to [?* ~ A]
~[?]  the empty language
%+  the literal plus-sign symbol
%*  the literal asterisk symbol (and similarly for %?,
%(), %] etc.

Introduction to XFST

[ ] the empty string
0  the empty string
[A] bracketing; equivalent to A
A | B  union
(A)  optionality; equivalent to [A|0]
A&B  intersection
A B  concatenation
A-B  set difference

Introduction to XFST – denoting relations

A .X. B  Cartesian product; relates every string in A to every
string in B
a:b  shorthand for [a .x. b]
%+p1:s  shorthand for [%+p1 .x. s]
%+past:ed  shorthand for [%+Past .x. ed]
%+prog:ing  shorthand for [%+Prog .x. ing]
Introduction to XFST – useful abbreviations

$A$ the language of all the strings that contain $A$; equivalent to [?* $A$ ?*]

$A/B$ the language of all the strings in $A$, ignoring any strings from $B$, e.g.,

$a*/b$ includes strings such as $a$, $aa$, $aaa$, $ba$, $ab$, $aba$ etc.

\( \backslash A \) any single symbol, minus strings in $A$, Equivalent to $[? \ A]$, e.g.,

\( \backslash b \) any single symbol, except 'b'. Compare to:

$\sim A$ the complement of $A$, i.e., $[?^* \ A]$

Introduction to XFST – example

[ leave %+VBZ .x. leaves ]
[ leave %+VB .x. leave ]
[ leave %+VBG .x. leaving ]
[ leave %+VBD .x. left ]
[ leave %+NN .x. leave ]
[ leave %+NNS .x. leaves ]
[ leaf %+NNS .x. leaves ]
[ left %+JJ .x. left ]

Introduction to XFST – user interface

prompt% H:\class\data\shuly\xfst

xfst> help
xfst> help union net
xfst> exit
xfst> read regex [d o g | c a t];
xfst> read regex < myfile.regex
xfst> apply up dog
xfst> apply down dog
xfst> pop stack
xfst> clear stack
xfst> save stack myfile.fsm

Introduction to XFST – example of lookup and generation

APPLY DOWN> leave+VBD
left
APPLY UP> leaves
leave+NNS
leave+VBZ
leaf+NNS
Introduction to XFST – variables

xfst> define Myvar;  
xfst> define Myvar2 [dog | cat];  
xfst> undef Myvar;  

xfst> define var1 [bird | frog | dog];  
xfst> define var2 [dog | cat];  
xfst> define var3 var1 | var2;  
xfst> define var4 var1 var2;  
xfst> define var5 var1 & var2;  
xfst> define var6 var1 - var2;  

Introduction to XFST – replace rules

Replace rules are an extremely powerful extension of the regular expression metalanguage.

The simplest replace rule is of the form

\[ \text{upper} \rightarrow \text{lower} \parallel \text{leftcontext} - \\text{rightcontext} \]

Its denotation is the relation which maps string to themselves, with the exception that an occurrence of upper in the input string, preceded by leftcontext and followed by rightcontext, is replaced in the output by lower.

Introduction to XFST – replace rules

The language Bambona has an underspecified nasal morpheme \( N \) that is realized as a labial \( m \) or as a dental \( n \) depending on its environment: \( N \) is realized as \( m \) before \( p \) and as \( n \) elsewhere.

The language also has an assimilation rule which changes \( p \) to \( m \) when the \( p \) is followed by \( m \).

xfst> clear stack;  
xfst> define Rule1 N -> m || - p;  
xfst> define Rule2 N -> n;  
xfst> define Rule3 p -> m || m -;  
xfst> read regex Rule1 .o. Rule2 .o. Rule3 ;
Introduction to XFST – replace rules

Word boundaries can be explicitly referred to:

```
xfst> define Vowel [a|e|i|o|u];
xfst> e -> , | | [.#.] [c | d | l | s] _ [% Vowel];
```

Introduction to XFST – replace rules

Contexts can be omitted:

```
xfst> define Rule1 N -> m || _ p ;
xfst> define Rule2 N -> n ;
xfst> define Rule3 p -> m || m _ ;
```

This can be used to clear unnecessary symbols introduced for “bookkeeping”:

```
xfst> define Rule1 %MorphemeBoundary -> 0;
```

Introduction to XFST – replace rules

Rules can define multiple replacements:

```
[ A -> B, B -> A ]
```

or multiple replacements that share the same context:

```
[ A -> B, B -> A || L _ R ]
```

or multiple contexts:

```
[ A -> B || L1 _ R1, L2 _ R2 ]
```

or multiple replacements and multiple contexts:

```
[ A -> B, B -> A || L1 _ R1, L2 _ R2 ]
```

Introduction to XFST – replace rules

Rules can apply in parallel:

```
xfst> clear stack
xfst> read regex a -> b .o. b -> a ;
xfst> apply down abba
aaaa
xfst> clear stack
xfst> read regex b -> a .o. a -> b ;
xfst> apply down abba
bbbb
xfst> clear stack
xfst> read regex a -> b , b -> a ;
xfst> apply down abba
baab
```


**Introduction to XFST – replace rules**

When rules that have contexts apply in parallel, the rule separator is a double comma:

```
xfst> clear stack
xfst> read regex
b -> a || .#. s ?* _ , a -> b || _ ?* e .#. ;
xfst> apply down sabbе
sbaає
```

**Introduction to XFST – marking**

The special symbol “…” in the right-hand side of a replace rule stands for whatever was matched in the left-hand side of the rule.

```
xfst> clear stack;
xfst> read regex [a|e|i|o|u] -> %[ ... %];
xfst> apply down unnecessarily
[u]nn[e]c[e] ss[a]r[i]ly
```

**Introduction to XFST – marking**

Assume that text is represented as strings of part-of-speech tags, using ‘d’ for determiner, ‘a’ for adjective, ‘n’ for noun, and ‘v’ verb, etc. In other words, in this example the regular expression symbols represent whole words rather than single letters in a text.

Assume that a noun phrase consists of an optional determiner, any number of adjectives, and one or more nouns:

```
[(d) a* n+]
```

This expression denotes an infinite set of strings, such as “n” (cats), “aan” (discriminating aristocratic cats), “nn” (cat food), “dn” (many cats), “dann” (that expensive cat food) etc.
Introduction to XFST – shallow parsing

A simple noun phrase parser can be thought of as a transducer that inserts markers, say, a pair of braces { }, around noun phrases in a text. The task is not as trivial as it seems at first glance. Consider the expression

\[(d) \ a\*\ n\+ \rightarrow \ %\{ \ldots \%\}\]

Applied to the input “danvn” (many small cats like milk) this transducer yields three alternative bracketings:

xfst> apply down danvn
da{n}v{n}
da{n}v{n}
d{an}v{n}
{dan}v{n}

Introduction to XFST – longest match

For certain applications it may be desirable to produce a unique parse, marking the maximal expansion of each NP: “{dan\}v{n}”. Using the left-to-right, longest-match replace operator \(\rightarrow\) instead of the simple replace operator \(\rightarrow\) yields the desired result:

\[(d) \ a\*\ n\+ \rightarrow \ %\{ \ldots \%\}\]

xfst> apply down danvn
{dan}v{n}

Introduction to XFST – the coke machine

A vending machine dispenses drinks for 65 cents a can. It accepts any sequence of the following coins: 5 cents (represented as ‘n’), 10 cents (‘d’) or 25 cents (‘q’). Construct a regular expression that compiles into a finite-state automaton that implements the behavior of the soft drink machine, pairing “PLONK” with a legal sequence that amounts to 65 cents.

The construction \(\mathbf{\ast}^n\) denotes the concatenation of \(\mathbf{\ast}\) with itself \(n\) times.

Thus the expression \([n .x. c^\ast 5]\) expresses the fact that a nickel is worth 5 cents.

A mapping from all possible sequences of the three symbols to the corresponding value:

\([[n .x. c^\ast 5] \mid [d .x. c^\ast 10] \mid [q .x. c^\ast 25]]\star\]

The solution:

\([[n .x. c^\ast 5] \mid [d .x. c^\ast 10] \mid [q .x. c^\ast 25]]\star\]
\(\cdot\)
\(c^\ast 65 .x. PLONK\)
Introduction to XFST – the coke machine

clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define BuyCoke
SixtyFiveCents .o. [c^65 .x. PLONK] ;

Introduction to XFST – the coke machine

In order to ensure that extra money is paid back, we need to modify the lower language of BuyCoke to make it a subset of
[PLONK* q* d* n*].
To ensure that the extra change is paid out only once, we need to make sure that quarters get paid before dimes and dimes before nickels.
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define ReturnChange SixtyFiveCents .o.
[[c^65 .x. PLONK]* [c^25 .x. q]*
[c^10 .x. d]* [c^5 .x. n]*] ;

Introduction to XFST – the coke machine

The next refinement is to ensure that as much money as possible is converted into soft drinks and to remove any ambiguity in how the extra change is to be reimbursed.
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define ReturnChange SixtyFiveCents .o.
[[c^65 .x. PLONK]* [c^25 .x. q]*
[c^10 .x. d]* [c^5 .x. n]*] ;
define IgnoreGarbage
[ [ ExactChange ]/[\[q | d | n]] ] ;

Introduction to XFST – the coke machine

To make the machine completely foolproof, we need one final improvement. Some clients may insert unwanted items into the machine (subway tokens, foreign coins, etc.). These objects should not be accepted; they should pass right back to the client. This goal can be achieved easily by wrapping the entire expression inside an ignore operator.
define IgnoreGarbage
[ [ ExactChange ]/[\[q | d | n]] ] ;