Finite-state automata

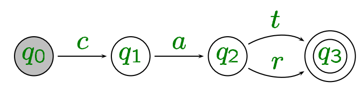
Automata are models of computation: they compute languages.

A finite-state automaton is a five-tuple $\langle Q, q_0, \Sigma, \delta, F\rangle$, where Σ 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.

Finite-state automata

Example: Finite-state automaton

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{c, a, t, r\}$
- $F = \{q_3\}$
- $\delta = \{\langle q_0, c, q_1\rangle, \langle q_1, a, q_2\rangle, \langle q_2, t, q_3\rangle, \langle q_2, r, q_3\rangle\}$



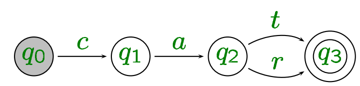
Finite-state automata

The reflexive transitive extension of the transition relation δ is a new relation, $\hat{\delta}$, defined as follows:

- for every state $q \in Q$, $(q, \epsilon, q) \in \hat{\delta}$
- for every string $w \in \Sigma^*$ and letter $a \in \Sigma$, if $(q, w, q') \in \hat{\delta}$ and $(q', a, q'') \in \delta$ then $(q, w \cdot a, q'') \in \hat{\delta}$.

Finite-state automata

Example: Paths
For the finite-state automaton:



$\hat{\delta}$ is the following set of triples:

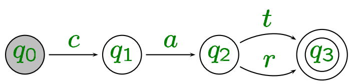
$$\langle q_0, \epsilon, q_0\rangle, \langle q_1, \epsilon, q_1\rangle, \langle q_2, \epsilon, q_2\rangle, \langle q_3, \epsilon, q_3\rangle,$$
$$\langle q_0, c, q_1\rangle, \langle q_1, a, q_2\rangle, \langle q_2, t, q_3\rangle, \langle q_2, r, q_3\rangle,$$
$$\langle q_0, ca, q_2\rangle, \langle q_1, at, q_3\rangle, \langle q_1, ar, q_3\rangle,$$
$$\langle q_0, cat, q_3\rangle, \langle q_0, car, q_3\rangle$$

Finite-state automata

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

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:



is $\{$*cat, car*$\}$.

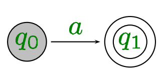
Finite-state automata

Example: Some finite-state automata

$\emptyset$

Finite-state automata

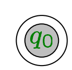
Example: Some finite-state automata



$\{a\}$

Finite-state automata

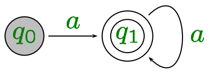
Example: Some finite-state automata



$\{\epsilon\}$

Finite-state automata

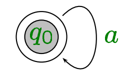
Example: Some finite-state automata



$\{a, aa, aaa, aaaa, \ldots\}$

Finite-state automata

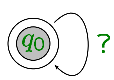
Example: Some finite-state automata



a^*

Finite-state automata

Example: Some finite-state automata



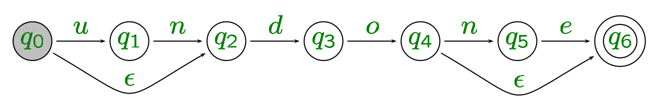
Σ^*

Finite-state automata

An extension: ϵ-moves.

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

Example: Automata with ϵ-moves
The language accepted by the following automaton is $\{$*do, undo, done, undone*$\}$:



Finite-state automata

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

Finite-state automata

Example: Finite-state automata and regular expressions

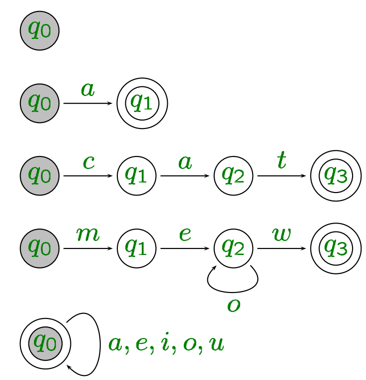
$\emptyset$

a

$((c \cdot a) \cdot t)$

$(((m \cdot e) \cdot (o)^*) \cdot w)$

$((a + (e + (i + (o + u)))))^*$



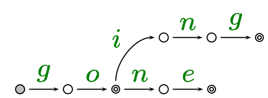
Operations on finite-state automata

- Concatenation
- Union
- Intersection
- Minimization
- Determinization

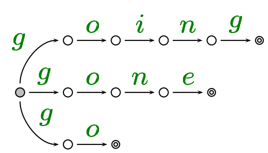
Minimization and determinization

Example: Equivalent automata

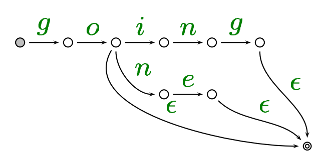
A_1



A_2



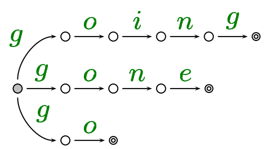
A_3



Applications of finite-state automata in language processing

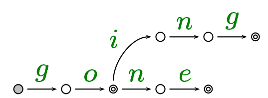
Lexicon:

go, gone, going :



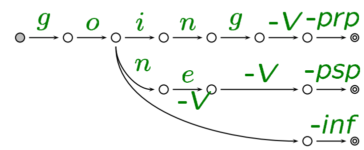
This automaton can then be determinized and minimized:

go, gone, going :



Applications of finite-state automata in language processing

A naïve morphological analyzer:



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.

Regular relations

Part-of-speech tagging:

I	know	some	new	tricks	
PRON	V	DET	ADJ	N	
said	the	Cat	in	the	Hat
V	DET	N	P	DET	N

Regular relations

Morphological analysis:

I I-PRON-1-sg	know know-V-pres	some some-DET-indef	new new-ADJ
tricks trick-N-pl	said say-V-past	the the-DET-def	Cat cat-N-sg
in in-P	the the-DET-def	Hat hat-N-sg	

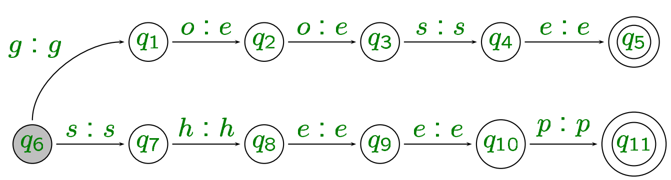
Regular relations

Singular-to-plural mapping:

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

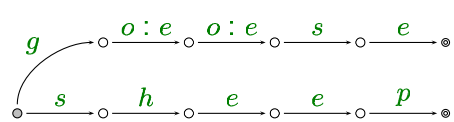
Finite-state transducers

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, Σ_1 and Σ_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.

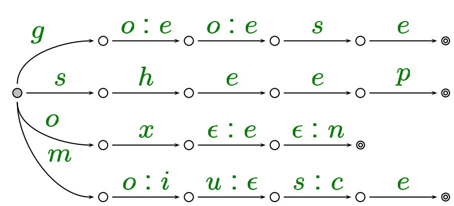


Finite-state transducers

Shorthand notation:



Adding ϵ-moves:



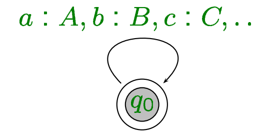
Finite-state transducers

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.

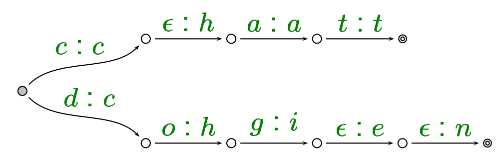
$T(w) = \{u \mid (q_0, w, u, q_f \in \hat{\delta})$ for some $f \in F\}$.

Finite-state transducers

Example: The uppercase transducer



Example: English-to-French



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 $a \in \Sigma_a$, $(q_1, a, b, q_2) \in \delta$

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 $(q_1, b, a, q_2) \in \delta$.

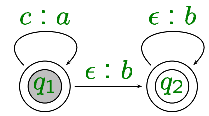
Properties of finite-state transducers

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

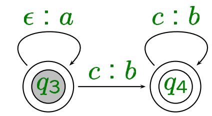
The number of states in the composition transducer might be $|Q_1 \times Q_2|$.

Properties of finite-state transducers

Transducers are not closed under intersection.



T_1



T_2

$$T_1(c^n) = \{a^n b^m \mid m \geq 0\}$$
$$T_2(c^n) = \{a^m b^n \mid m \geq 0\} \Rightarrow$$
$$(T_1 \cap T_2)(c^n) = \{a^n b^n\}$$

Transducers with no ϵ-moves are closed under intersection.

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
`c a t`	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

`[ ]`	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

`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 – denoting relations

`A .x. B`	Cartesian product; relates every string in `A` to every string in `B`
`a:b`	shorthand for `[a .x. b]`
`%+Pl:s`	shorthand for `[%+Pl .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.
`\A`	any single symbol, minus strings in `A`. Equivalent to `[? - A]`, e.g.,
`\b`	any single symbol, except 'b'. Compare to:
`~A`	the complement of `A`, i.e., `[?* - A]`

Introduction to XFST – example

```
[ [l e a v e %+VBZ .x. l e a v e s] |
[l e a v e %+VB .x. l e a v e] |
[l e a v e %+VBG .x. l e a v i n g] |
[l e a v e %+VBD .x. l e f t] |
[l e a v e %+NN .x. l e a v e] |
[l e a v e %+NNS .x. l e a v e s] |
[l e a f %+NNS .x. l e a v e s ] |
[l e f t %+JJ .x. l e f t] ]
```


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 [d o g | c a t];
xfst> undefine Myvar;
```

```
xfst> define var1 [b i r d | f r o g | d o g];
xfst> define var2 [d o g | c a t];
xfst> define var3 var1 | var2;
xfst> define var4 var1 var2;
xfst> define var5 var1 & var2;
xfst> define var6 var1 - var2;
```


Introduction to XFST – variables

```
xfst> define Root [w a l k | t a l k | w o r k];
xfst> define Prefix [0 | r e];
xfst> define Suffix [0 | s | e d | i n g];
xfst> read regex Prefix Root Suffix;
xfst> words
xfst> apply up walking
```


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

$$upper \rightarrow lower \; || \; leftcontext \; \_ \; rightcontext$$

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 %^MorphmeBoundary -> 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 sabbae
sbaabe
```


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

```
xfst> clear stack;
xfst> read regex [a|e|i|o|u]+ -> %[ ... %];
xfst> apply down feeling
f[e][e]l[i]ng
f[ee]l[i]ng
xfst> apply down poolcleaning
p[o][o]lcl[e][a]n[i]ng
p[oo]lcl[e][a]n[i]ng
p[o][o]lcl[ea]n[i]ng
p[oo]lcl[ea]n[i]ng
xfst> read regex [a|e|i|o|u]+ @-> %[ ... %];
xfst> apply down poolcleaning
p[oo]lcl[ea]n[i]ng
```


Introduction to XFST – shallow parsing

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+ -> %{ ... %}]
```

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

```
xfst> apply down danvn
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 `@->` instead of the simple replace operator `->` yields the desired result:

```
[(d) a* n+ @-> %{ ... %}]
```

```
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.

Introduction to XFST – the coke machine

The construction `A^n` denotes the concatenation of `A` with itself n times.

Thus the expression `[n .x. c^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^5] | [d .x. c^10] | [q .x. c^25]]*
```

The solution:

```
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]*
.o.
[c^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 ExactChange ReturnChange .o.
[~$[q q q | [q q | d] d [d | n] | n 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 passed 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]]] ;
```