

# LING/C SC/PSYC 438/538

Lecture 16

Prof. Sandiway Fong

## Today's Topics

- Homework 10: only one question!
  - (but maybe challenging)
- Finite State Automata (FSA)

# Homework 10

- Consider again the nested object relative clause construction from Homework 9
- Examples:
  1. the boy who the girl saw
  2. the boy who the girl who the woman found saw
  3. the boy who the girl who the woman who the dog chased found sawetc.

# Homework 10

- Given a **recursive** Perl regex that computes the predicate argument structure for examples 1–3 (from the previous slide).
- Show your code working on each example.
- Example output:
  3. the boy who the girl who the **woman who the dog chased** found saw
    - `saw(girl, boy)`
    - `found(woman, girl)`
    - `chased(dog, woman)`



Hint: doesn't have  
to be in this order  
of output

# Homework 10

## Hints:

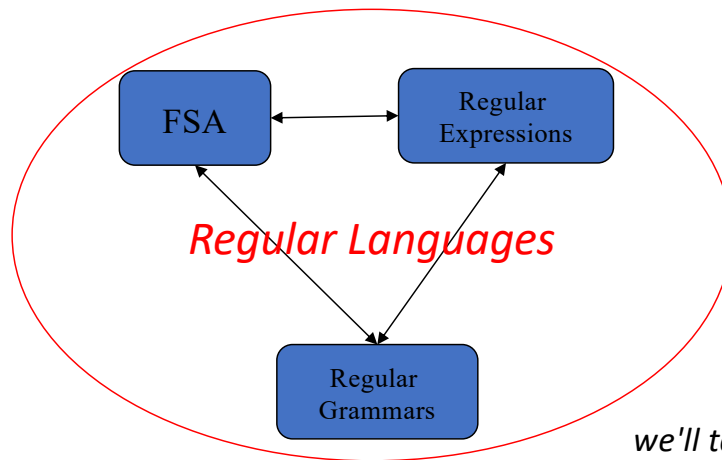
- `perl -le '$_ = qq{@ARGV}; /regex/'` the boy who the girl saw
- **recursive** means we don't need a loop or the **global flag: g**
- **recursive** (`?n`) where `n = 0` (whole regex), `n = 1` (1st parentheses group) etc.
- **lookahead**: (`?=regex`)
- **negative lookahead**: (`?!regex`) may be useful if you want to block regex
- **Perl code**: (`?{...}`) could be embedded inside a regex to do output
- e.g. (`?{print "$n(...)"}`)

# Homework 10

- Email to me: [sandiway@arizona.edu](mailto:sandiway@arizona.edu)
- Subject: 438/538 Homework 10: ***YOUR NAME***
- Submit everything in one PDF file
- Submit a real file, not a link
- **Screenshots**: make sure text is readable – not too tiny!
- Due **Sunday** midnight:
  - to be reviewed in class next **Tuesday**

# Regular Languages

- Three formalisms:
  - All formally equivalent (no difference in expressive power)
  - i.e. if you can encode it using a RE, you can do it using a FSA or regular grammar, and so on ...



**Note:** Perl regexs are more powerful than the math characterization:

- backreferences `\n`,
- recursive regexs `(?n)`,
- insertion of general code `(?{...})`

*we'll talk about formal equivalence next time...*

# Regular Languages

- A regular language is the set of strings
  - (including possibly the empty string)
  - (set itself could also be empty)
  - (set can be infinite)
  - generated by a regex/FSA/Regular Grammar

**Note:** in formal language theory: a language =<sub>def</sub> set of strings  
(we don't specify how it's generated)

# Regular Languages

- **Example:**

- Language:  $L = \{ a^+b^+ \}$

*“one or more a’s followed by one or more b’s”*

**L is a regular language**

- described by a regular expression (*we’ll define it formally next time*)

- **Note:**

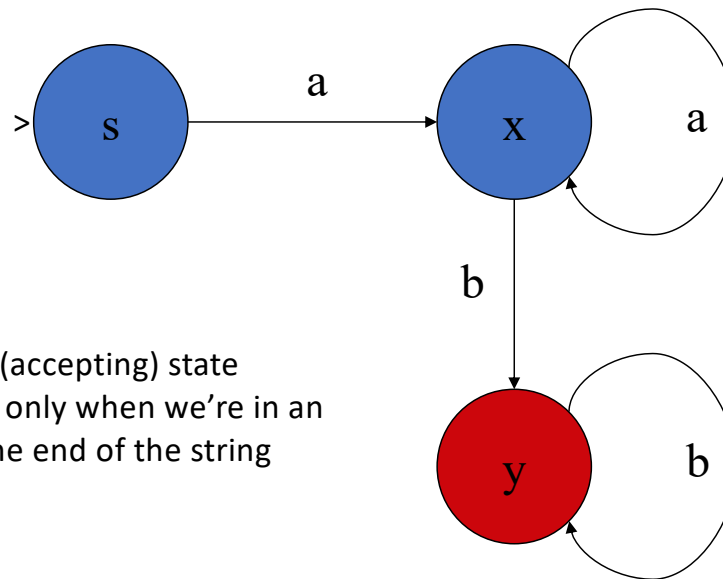
- infinite set of strings belonging to language L
  - e.g. abbb, aaaab, aabb, \*abab, \* $\lambda$

- **Notation:**

- $\lambda$  is the empty string (or string with zero length), sometimes  $\epsilon$  is used instead
- \* means string is not in the language

# Finite State Automata (FSA)

- $L = \{ a^+b^+ \}$  can be also be generated by the following FSA

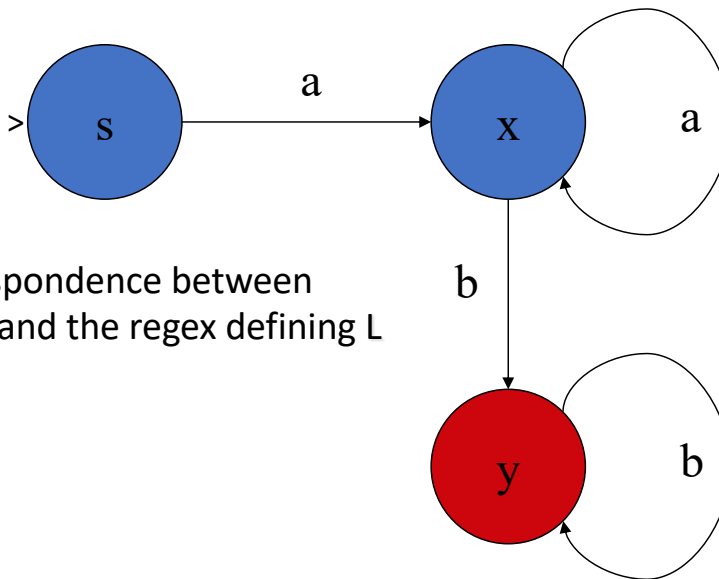


Conventions (*used here*):

1. > Indicates start state
2. Red circle indicates end (accepting) state
3. we accept a input string only when we're in an end state **and** we're at the end of the string

# Finite State Automata (FSA)

- $L = \{ a^+b^+ \}$  can be also be generated by the following FSA



There is a natural correspondence between components of the FSA and the regex defining  $L$

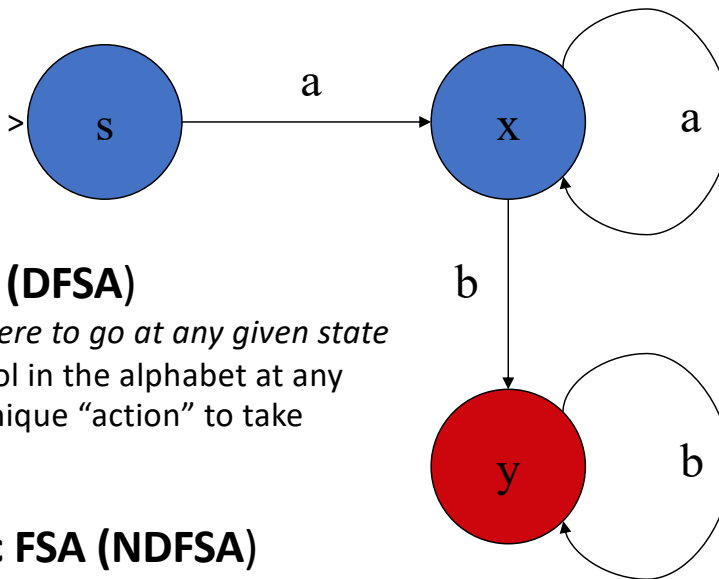
Note:

$L = \{a^+b^+\}$

$L = \{aa^*bb^*\}$

# Finite State Automata (FSA)

- $L = \{ a^+b^+ \}$  can be also be generated by the following FSA



## **deterministic FSA (DFSA)**

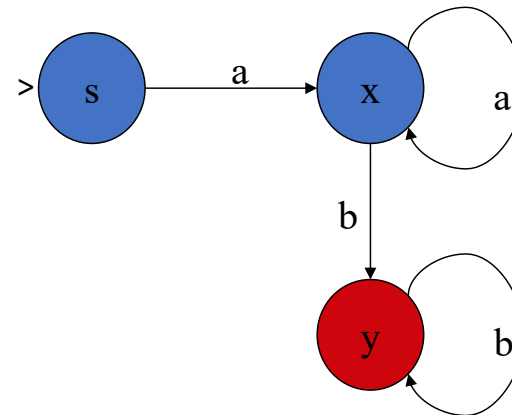
*no ambiguity about where to go at any given state*  
i.e. for each input symbol in the alphabet at any given state, there is a unique “action” to take

## **non-deterministic FSA (NDFSA)**

*no restriction on ambiguity (surprisingly, no increase in power)*

# Finite State Automata (FSA)

- **more formally**
  - $(Q, s, f, \Sigma, \delta)$ 
    1. set of states (**Q**):  $\{s, x, y\}$  *must be a **finite** set*
    2. start state (**s**): s
    3. end state(s) (**f**): y
    4. alphabet ( **$\Sigma$** ):  $\{a, b\}$
    5. transition function  $\delta$ :  
*signature: character  $\times$  state  $\rightarrow$  state*
      - $\delta(a, s) = x$
      - $\delta(a, x) = x$
      - $\delta(b, x) = y$
      - $\delta(b, y) = y$



# Finite State Automata (FSA)

- **In Perl**

transition function  $\delta$ :

- $\delta(a,s)=x$
- $\delta(a,x)=x$
- $\delta(b,x)=y$
- $\delta(b,y)=y$

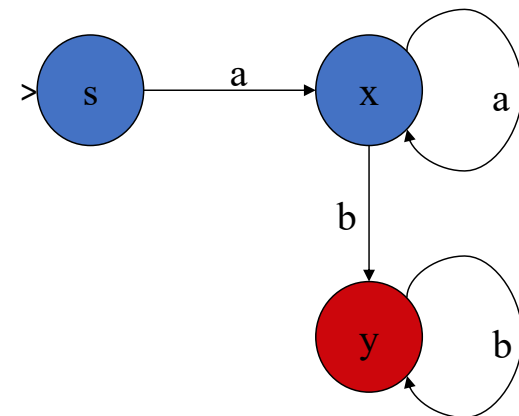
**Syntactic sugar for**

```
%transitiontable = (  
  "s", { "a", "x", },  
  "x", { "a", "x", "b", "y" },  
  "y", { "b", "y" },  
);
```

We can simulate our 2D transition table using a hash table  
whose elements are themselves also hash tables

(*anonymized*; **note**: `{..}` = hashes)

```
%transitiontable = (  
  s => {  
    a => "x"  
  },  
  x => {  
    a => "x",  
    b => "y"  
  },  
  y => {  
    b => "y"  
  }  
);
```



**Example:**

```
print "$transitiontable{s}{a}\n";
```

# Finite State Automata (FSA)

- Given transition table encoded as a (nested) hash
- How to build a **decider** (Accept/Reject) in Perl?

## **Complications to think about:**

- How about  $\epsilon$ -transitions?
- Multiple end states?
- Multiple start states?
- Non-deterministic FSA?

# Finite State Automata (FSA)

```
%transitiontable = (  
    s => {a    => "x"},  
    x => {a    => "x", b    => "y"},  
    y => {b    => "y"}  
);  
$state = "s";  
foreach $c (@ARGV) {  
    $state = $transitiontable{$state}{$c};  
}  
if ($state eq "y") { print "Accept\n"; }  
else { print "Reject\n" }
```

- Example runs:

- perl fsm.prl a b a b
- **Reject**
- perl fsm.prl a a a b b
- **Accept**

# Finite State Automata (FSA)

- Perl one-liner:

```
perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s";  
for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"'
```

# Finite State Automata (FSA)

- Perl one-liner examples:

- `perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' a`
- `perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' a b`
- Accept

```
~$ perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' a
~$ perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' a b
Accept
~$ perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' a b b
Accept
~$ perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' a a b b
Accept
~$ perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' a a b
Accept
~$ perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' a b b a
~$ perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"' b a a b
~$ perl -le '%h=(s=>{a=>"x"},x=>{a=>"x",b=>"y"},y=>{b=>"y"}); $s="s"; for $c (@ARGV) {$s=$h{$s}{$c}}; print "Accept" if $s eq "y"'
~$
```

# Finite State Automata (FSA)

```
function D-RECOGNIZE(tape, machine) returns accept or reject  
  
  index ← Beginning of tape  
  current-state ← Initial state of machine  
  loop  
    if End of input has been reached then  
      if current-state is an accept state then  
        return accept  
      else  
        return reject  
    elseif transition-table[current-state, tape[index]] is empty then  
      return reject  
    else  
      current-state ← transition-table[current-state, tape[index]]  
      index ← index + 1  
  end
```

this is *just* **pseudo-code**  
not any real programming language  
but can be easily translated

**Figure 2.12** An algorithm for deterministic recognition of FSAs. This algorithm returns *accept* if the entire string it is pointing at is in the language defined by the FSA, and *reject* if the string is not in the language.

# In Python

```
1# mimick Perl code
2import sys
3tt = {'s': {'a':'x'}, 'x': {'a':'x', 'b':'y'}, 'y': {'b':'y'}}
4state = 's'
5for input in sys.argv[1:]:
6    x = tt[state]
7    if input in x:
8        state = x[input]
9    else:
10        state = 'reject'
11        break
12if state == 'y':
13    print "Accept"
14else:
15    print "Reject"
```

1. Python dictionary = Perl hash
  1. key:value
2. sys.argv = @ARGV  
(but numbered from 1, not 0)
3. [1:] slices the command line

# In Python

```
1# using tuples (state,input) as keys
2import sys
3tt = { ('s','a'):'x', ('x','a'):'x', ('x','b'):'y', ('y','b'):'y'}
4state = 's'
5for input in sys.argv[1:]:
6    if (state,input) in tt:
7        state = tt[(state,input)]
8    else:
9        state = 'reject'
10        break
11if state == 'y':
12    print "Accept"
13else:
14    print "Reject"
```

- Python has a data structure called a **tuple**:  $(e_1, \dots, e_n)$
- **Note**: Python lists use `[..]`
- In Python, crucially tuples (but not lists) can also be dictionary keys

**Note**: Many other ways of encoding FSA in Python, e.g. using object-oriented programming (classes)

<https://wiki.python.org/moin/FiniteStateMachine#FSA> - Finite State Automation in Python

# Finite State Automata (FSA)

- **Practical applications**

- *can be encoded and run efficiently on a computer*
- *widely used*
- **encode regular expressions (e.g. Perl regex – without the bells and whistles)**
- **morphological analyzers**
- e.g. want, want**ed**, **un**wanted; (suffixation/prefixation)
- car, car**s**; bike, bike**s**; box, box**es**;
- slow, slow**er**, slow**est**, slow**ly**, slow**ness**; (proud)
- realize, realiz**ation**, realiz**able**;
- Markov models
  - = FSA + transition probabilities
- *and much more ...*