LING/C SC/PSYC 438/538

Lecture 17

Sandiway Fong

Administrivia

Mathematical Theory of Computation

- Regular Languages:
  - Finite State Automata (FSA)
    = Mathematical Regular Expressions (regex just **not** Perl regex)
  - Basic FSA implementation
  - ε-transitions
  - Backreferences and FSA
- First, some remarks on Google Natural Language
  - (we discussed it in the homework review on Monday)

Unnatural Google Natural Language

Google NL is unnatural from at least several different perspectives:

Large amounts of training data REQUIRED

- Children learn from almost no evidence (Chomsky)

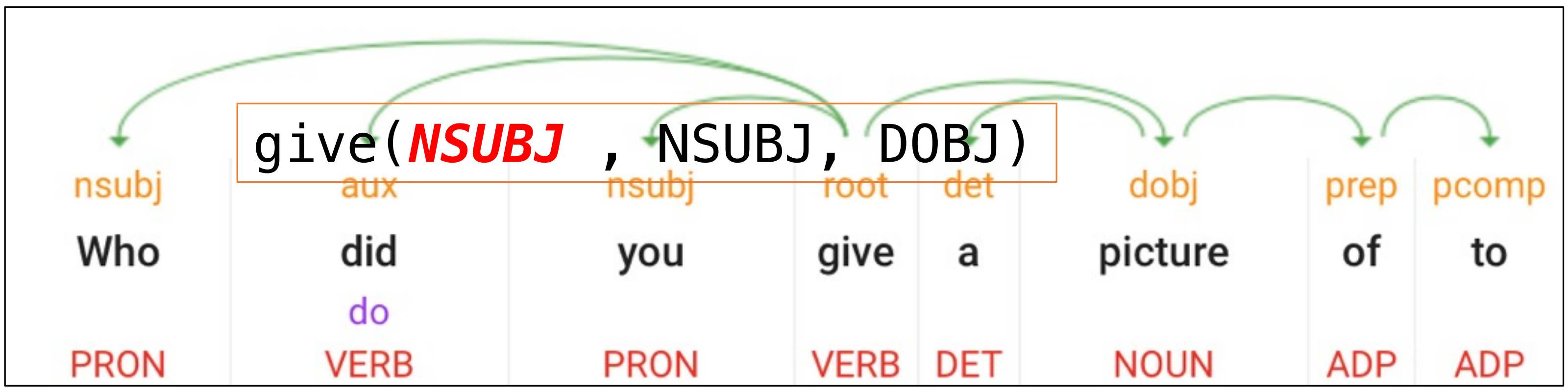
It invents analyses

- Not attested in Natural Language
- Not attested in its training data
- **Verbs generally have just one subject (nsubj/csubj) and a max of three core arguments (e.g. nsubj + dobj + iobj)**
- **Google NL:** can get four arguments (e.g. nsubj + iobj + iobj + ccomp)
- **Google NL:** can get two subjects with one predicate

Endowment

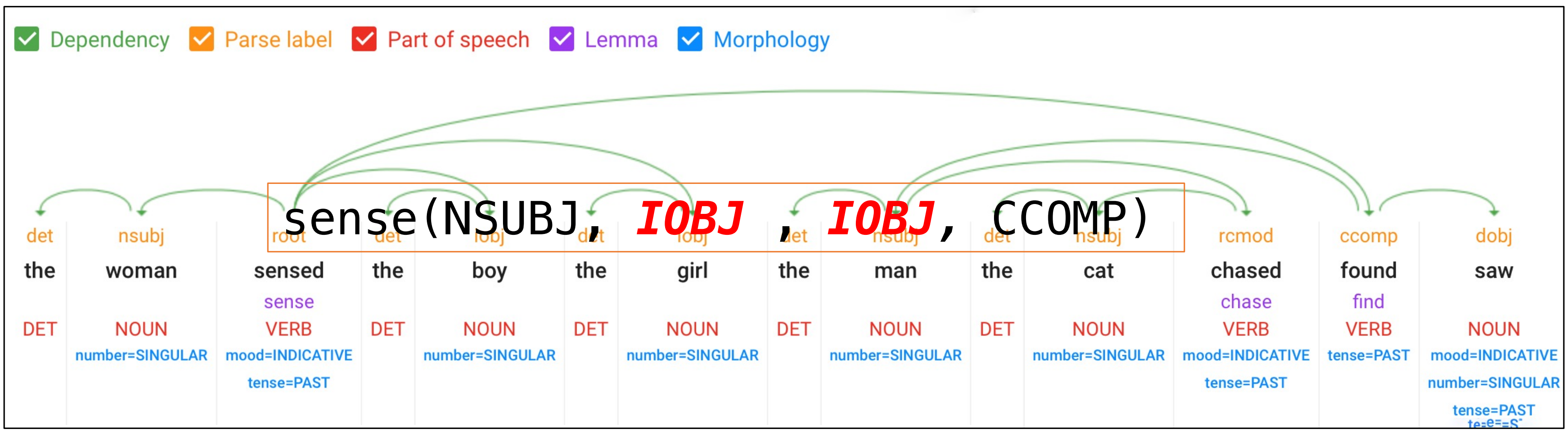
- Random seeding: initial state
- no special design (**general purpose**): no modularity

Natural Language: verb takes one subject



https://cloud.google.com/natural-language

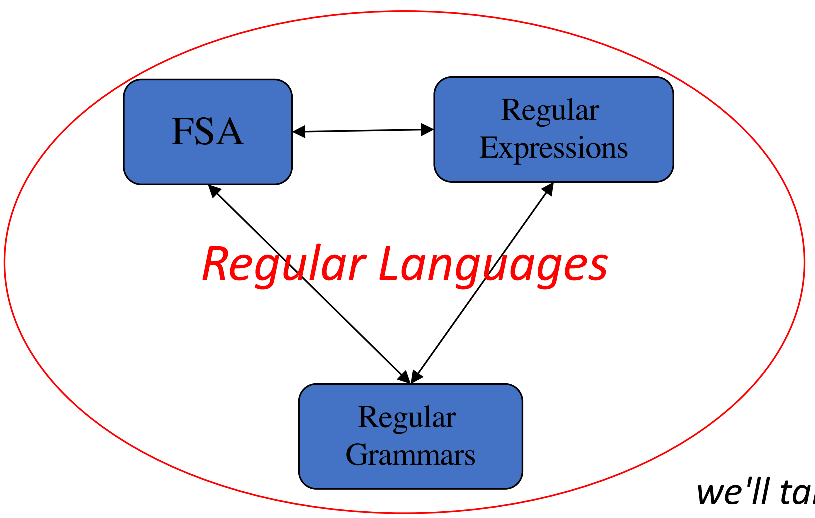
Natural Language: verb takes 3 arguments max



https://cloud.google.com/natural-language

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

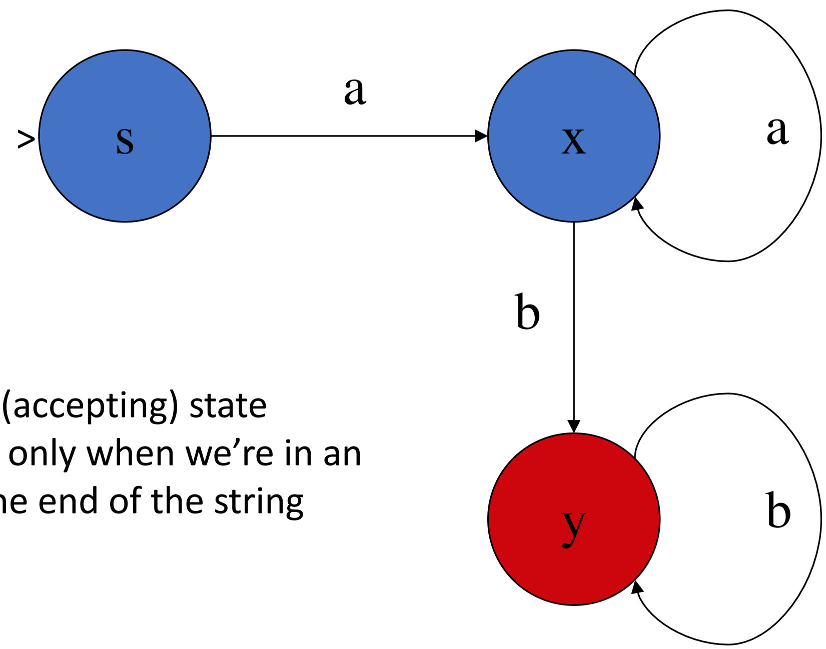
Note: in formal language theory: a language $=_{def}$ set of strings
(we don't specify how it's generated)

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, *λ
  - **Notation**:
    - λ is the empty string (or string with zero length), sometimes **ε** 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

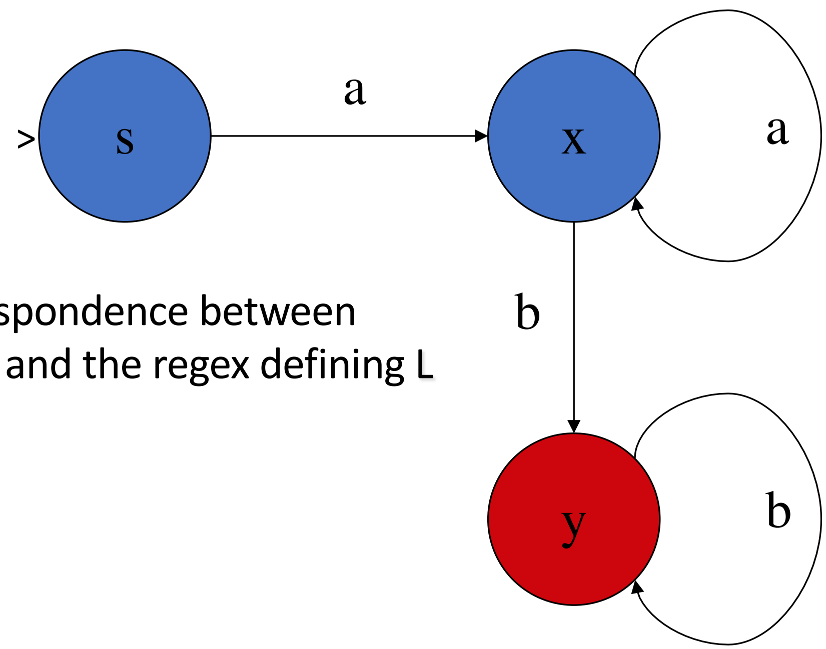


Conventions (*used here*):

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

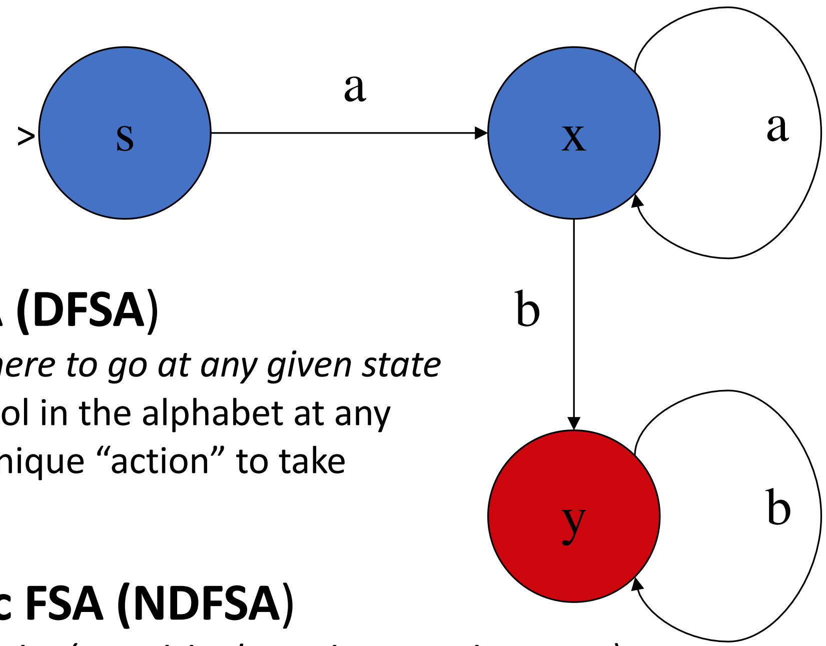


There is a natural correspondence between components of the FSA and the regex defining L

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)

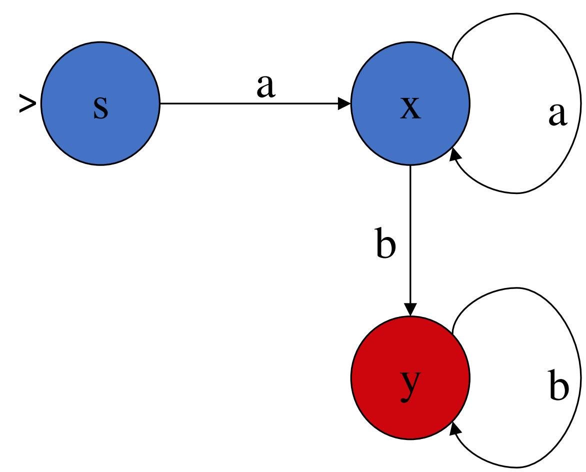
Finite State Automata (FSA)

- **more formally**
  - (Q,s,f,Σ,δ)

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 (**Σ**): {a, b}
5. transition function δ:

signature: character × state → state

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



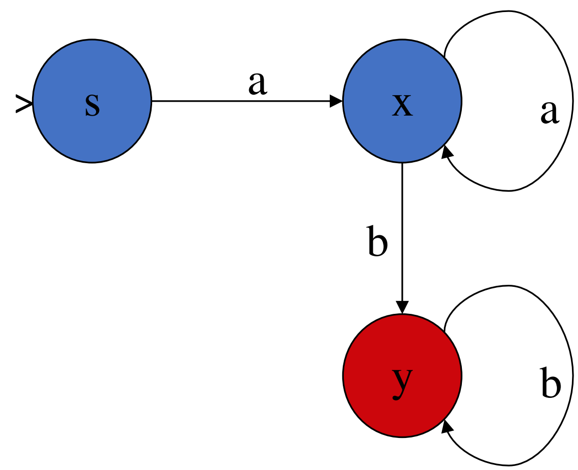
Finite State Automata (FSA)

- **In Perl**

  transition function δ:
  - δ(a,s)=x
  - δ(a,x)=x
  - δ(b,x)=y
  - δ(b,y)=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"
    }
);
```



Syntactic sugar for

```
%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 ε-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
   elsif transition-table[current-state,tape[index]] is empty then
      return reject
   else
      current-state ← transition-table[current-state,tape[index]]
      index ← index + 1
  end
```


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.

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

In Python

```
# mimick Perl code
import sys
tt = {'s': {'a':'x'}, 'x': {'a':'x', 'b':'y'}, 'y': {'b':'y'}}
state = 's'
for input in sys.argv[1:]:
    x = tt[state]
    if input in x:
        state = x[input]
    else:
        state = 'reject'
        break
if state == 'y':
    print "Accept"
else:
    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

```
# using tuples (state,input) as keys
import sys
tt = { ('s','a'):'x', ('x','a'):'x', ('x','b'):'y', ('y','b'):'y'}
state = 's'
for input in sys.argv[1:]:
    if (state,input) in tt:
        state = tt[(state,input)]
    else:
        state = 'reject'
        break
if state == 'y':
    print "Accept"
else:
    print "Reject"
```

- Python has a data structure called a **tuple**: $(e_1,..,e_n)$
- **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

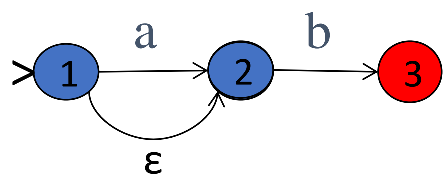
Finite State Automata (FSA)

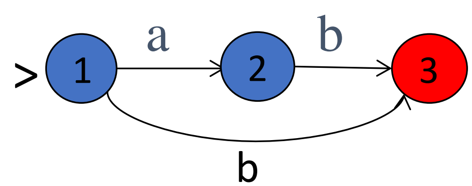
- **Practical applications**
  - *can be encoded and run efficiently on a computer*
  - *widely used*
  - **encode regular expressions (e.g. Perl regex)**
  - **morphological analyzers**
    - Different word forms, e.g. want, wanted, unwanted (suffixation/prefixation)
    - *see chapter 3 of textbook*
  - **speech recognizers**
    - Markov models
    - = FSA + probabilities

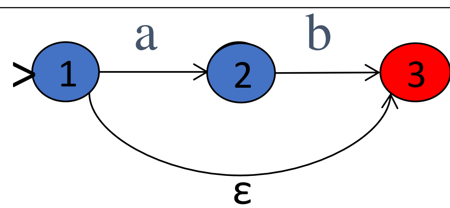
  - *and much more …*

ε-transitions

- jump from state to another state with the empty character
  - **ε-transition** (*textbook*) or **λ-transition**
  - no increase in expressive power (*meaning we could do without the* **ε-transition**)
- **examples**







what's the equivalent without the ε-transition?

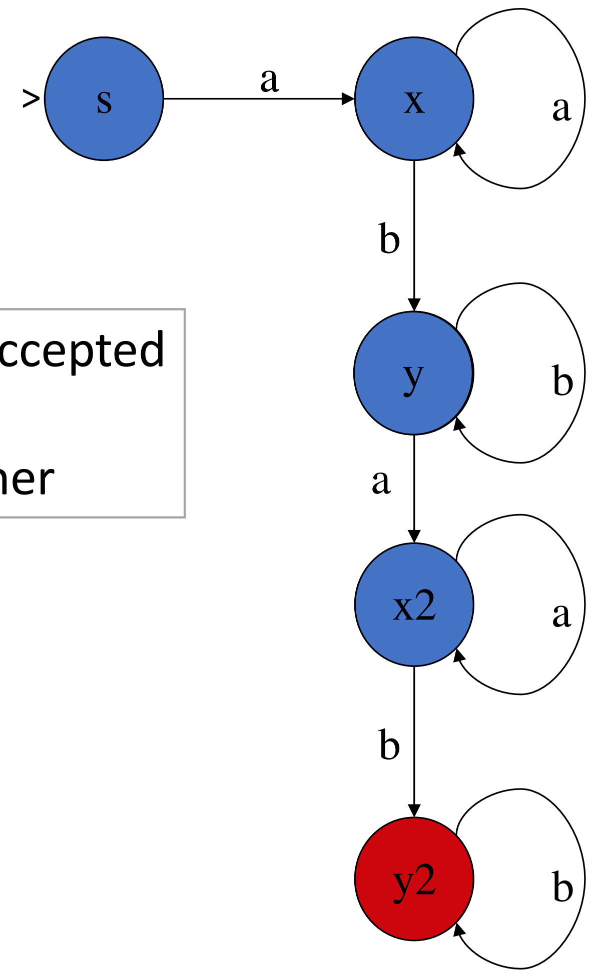
ε-transitions

- Can be used to help encode:
  1. Multiple start states
  2. Multiple end states

- Next time, we'll see:
  - Then we can get rid of the ε-transition (*by construction*)

Backreferences and FSA

- Deep question:
  - why are backreferences impossible in FSA?



Example: Suppose you wanted a machine that accepted /(a+b+)\1/
One idea: link two copies of the machine together

Doesn't work!
Why?

Backreferences and FSA

- fsa.perl

```
%delta = (
    s => { a   => "x" },
    x => { a   => "x", b   => "y" },
    y => { b   => "y", a   => "x2" },
    x2 => { a   => "x2", b   => "y2" },
    y2 => { b  => "y2"});
$state = "s";

foreach $c (split(//,@ARGV[0])) {
    $state = $delta{$state}{$c};
}

print (($state eq "y2") ? "Accept\n" : "Reject\n")
```

- Perl implementation: number of a's and b's in the two halves don't have to match:
- perl fsa.perl aabba
- Reject
- perl fsa.perl aabbaaaabbbb
- Accept
- perl fsa.perl aabbaaaab
- Accept