LING/C SC/PSYC 438/538

Lecture 16

Sandiway Fong

Today's Topics

- Prime number testing using Perl regex
- Finite State Automata (FSA)

Prime Number Testing using Perl Regular Expressions

- Another example:
  - the set of prime numbers is **not** a **regular** language (*can't do It with FSA/regex*)
  - L_{prime} = {2, 3, 5, 7, 11, 13, 17, 19, 23,.. }

Prime number - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/**Prime_number**
A **prime number** (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself. A natural number greater than 1 that is not a ...

Turns out, we can use a short Perl regex to determine membership in this set .. and to factorize numbers

`/^(11+?)\1+$/`

Prime Number Testing using Perl regex

- $L = \{1^n \mid n \text{ is prime}\}$ is not a **regular** language

can be proved mathematically using the Pumping Lemma for regular languages (*later*)

- Keys to making this work:
  - \1 backreference
  - unary notation for representing numbers, e.g.
    - 11111 “five ones” = 5
    - 111111 “six ones” = 6
  - unary notation allows us to factorize numbers by repetitive pattern matching
    - (11)(11)(11) “six ones” = 6
    - (111)(111) “six ones” = 6
  - numbers that can be factorized in this way aren’t prime!
    - no way to get nontrivial subcopies of 11111 “five ones” = 5
- Then /^(11+?)**\1**+$/ will match anything that’s greater than 1 that’s not prime

Prime Number Testing using Perl regex

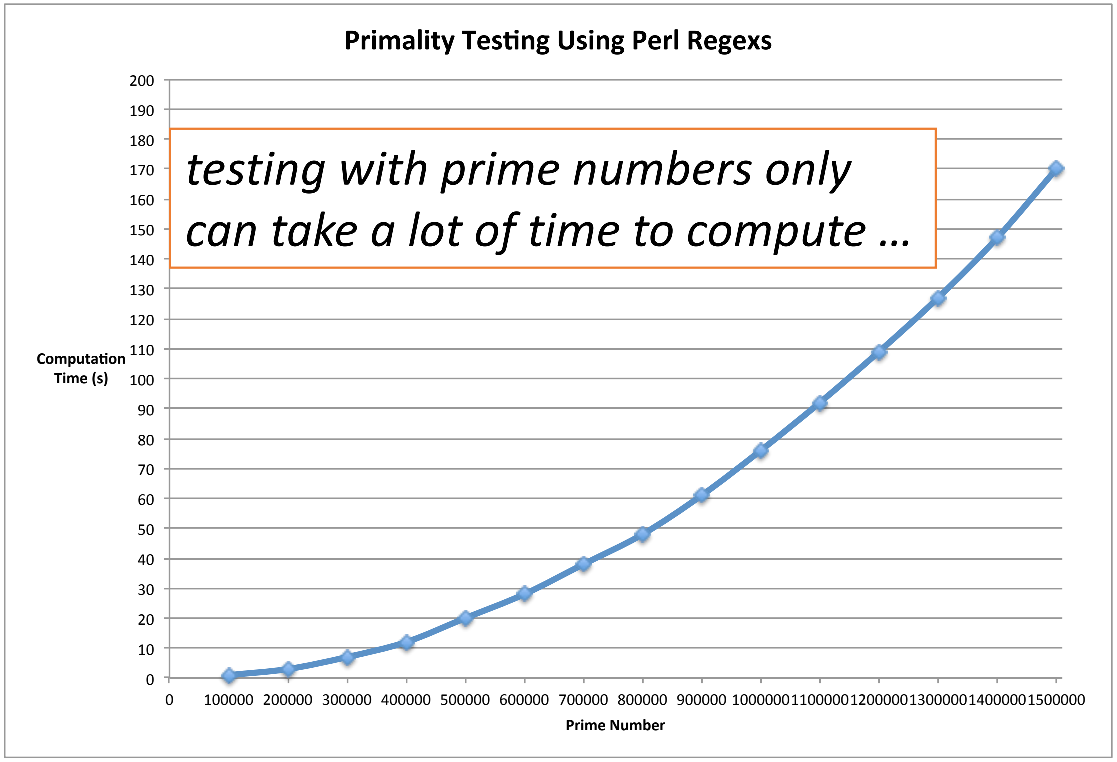
Question: is the non-greedy operator necessary?

- Let's analyze this Perl regex `/^(11+?)\1+$/`
  - `^` and `$` anchor both ends of the strings, forces `(11+?)\1+` to cover the entire string
  - `(11+?)` is the non-greedy (*shortest*) match version of `(11+)`
  - `\1+` provides one or more copies of what we previously matched in `(11+?)`
- Examples:

```
perl -le '$n = shift; $u = "1" x $n; print "$n prime" if $u !~
/^(11+?)\1+$/' 101
101 prime
perl -le '$n = shift; $u = "1" x $n; print "$n prime" if $u !~
/^(11+?)\1+$/' 103
103 prime
perl -le '$n = shift; $u = "1" x $n; print "$n prime" if $u !~
/^(11+?)\1+$/' 105
```

Prime Number Testing using Perl regex

Prime Numbers
100003
200003
300007
400009
500009
600011
700001
800011
900001
1000003
1100009
1200007
1300021
1400017
1500007



Prime Number Testing using Perl regex

- `/^(11+?)\1+$/` vs. `/^(11+)\1+$/`
- i.e. non-greedy vs. greedy matching
- finds smallest factor vs. largest
  - `90021 factored using 3, not a prime` (0 secs)

  `vs.`
  - `90021 factored using 30007, not a prime` (0 secs)

Puzzling behavior: same output non-greedy vs. greedy
`900021 factored using 300007, not a prime` (48 secs vs. 13 secs)

Prime Number Testing using Perl regex

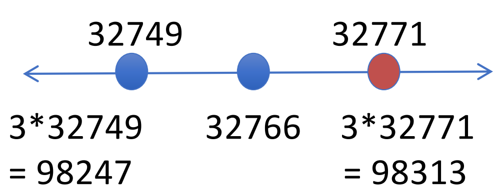
- http://www.xav.com/perl/lib/Pod/perlre.html

The following standard quantifiers are recognized:

```
*      Match 0 or more times
+      Match 1 or more times
?      Match 1 or 0 times
{n}    Match exactly n times
{n,}   Match at least n times
{n,m}  Match at least n but not more than m times
```

(If a curly bracket occurs in any other context, it is treated as a regular character.) The ``*'' modifier is equivalent to {0,}, the ``+'' modifier to {1,}, and the ``?'' modifier to {0,1}. n and m are limited to integral values less than a preset limit defined when perl is built. This is usually 32766 on the most common platforms.

nearest primes to preset limit



Prime Number Testing using Perl regex

- 32749 x 3 = 98247
- 32771 x 3 = 98313
- *When preset limit is exceeded: Perl's regex matching fails quietly*
- **Why does it report 32771?**

```
bash-3.2$ perl prime.perl 98247
Time 0: 98247 factored using 3, not a prime
bash-3.2$ perl prime.perl 98313
Time 1: 98313 factored using 32771, not a prime
```

Prime Number Testing using Perl Regular Expressions

- Can also get non-greedy to skip several factors
- Example: pick non-prime 164055 = 3 x 5 x 10937 (prime factorization)

Non-greedy: missed factors 3 and 5 …

```
bash-3.2$ perl prime.perl 164055
Time 0: 164055 factored using 15, not a prime
bash-3.2$ perl primeg.perl 164055
Time 1: 164055 factored using 54685, not a prime
```

greedy version

Because
3 * 54685 = 164055
5 * 32811 = 164055
32766 limit
15 * 10937 = 164055

Prime Number Testing using Perl Regular Expressions

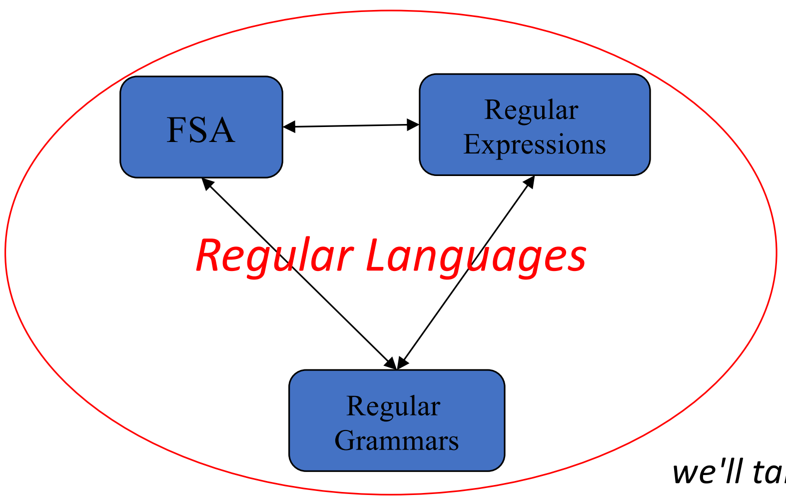
- Results are still right so far though:
  - *wrt*. prime vs. non-prime
- But we predict it will report an incorrect result for
  - `1,073,938,441`
  - It should claim (incorrectly) that this is prime since 1073938441 = 32771^2
  - (32766 is the limit for the number of bundles)

32611	32621	32633	32647	32653	32687	32693	32707	32713	32717
32719	32749	32771	32779	32783	32789	32797	32801	32803	32831
32833	32839	32843	32869	32887	32909	32911	32917	32933	32939

https://primes.utm.edu/lists/small/10000.txt

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

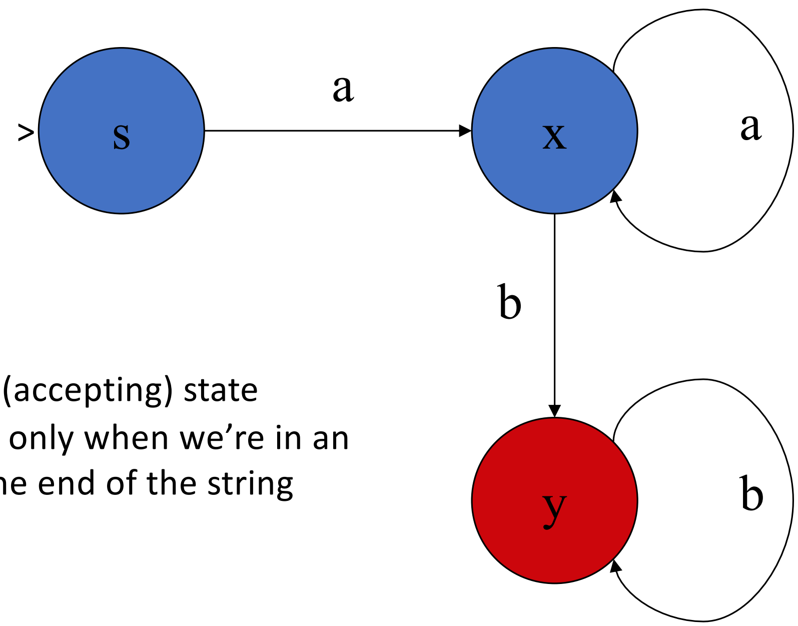
Note: in formal language theory: a language $=_{def}$ set of strings
(we don't specify how it's generated)

Regular Languages

- **Example**:
  - Language: $\mathbf{L = \{ a^+b^+ \}}$
    "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)

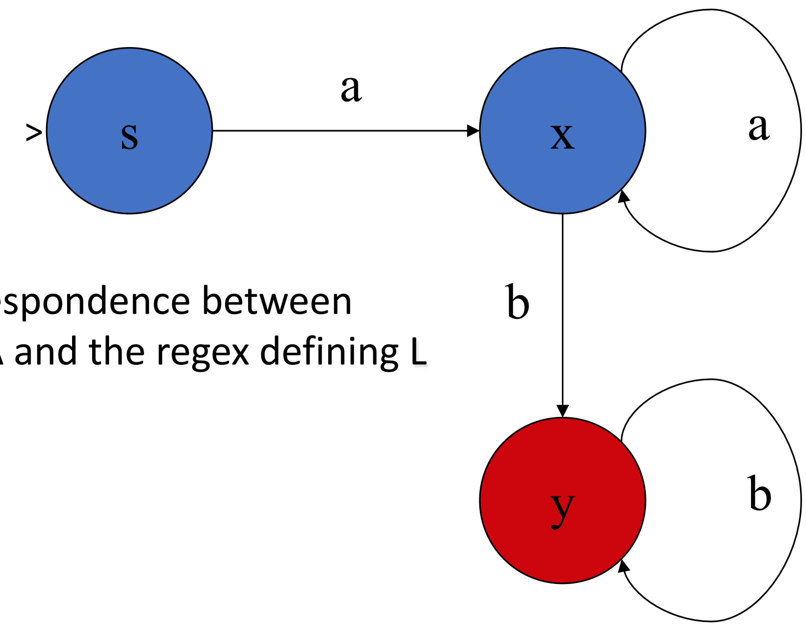
- L = { $a^{+}b^{+}$ } can be also be generated by the following FSA



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

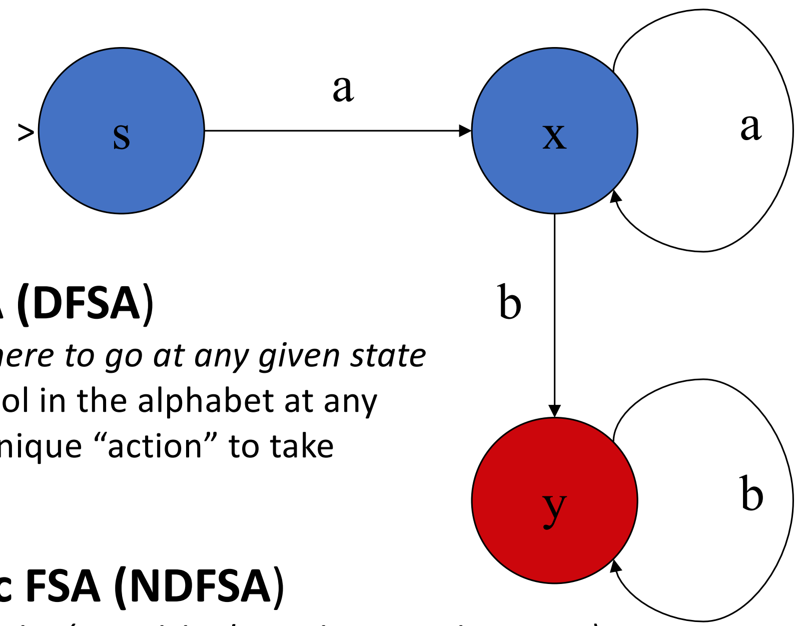


There is a natural correspondence between components of the FSA and the regex defining L

Note:
$L = \{a^+b^+\}$
$L = \{aa^*bb^*\}$

Finite State Automata (FSA)

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



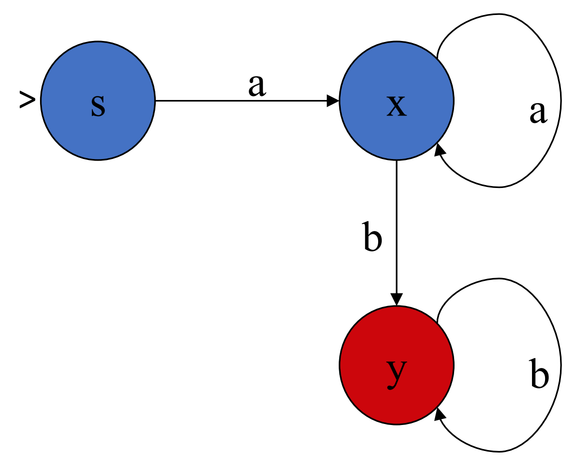
deterministic FSA (DFSA)
no ambiguity about where to go at any given state
i.e. for each input symbol in the alphabet at any given state, there is a unique "action" to take

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**

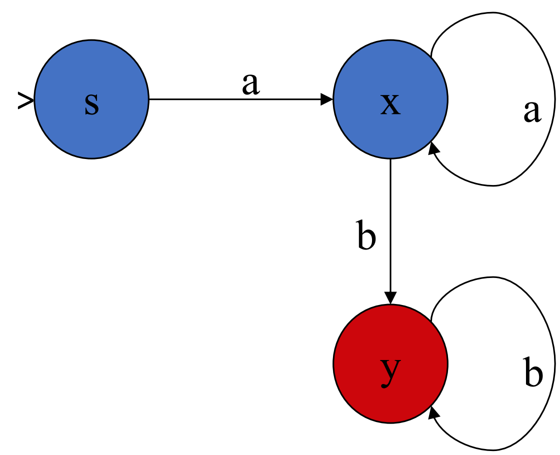
  transition function δ:

  - δ(a,s)=x
  - δ(a,x)=x
  - δ(b,x)=y
  - δ(b,y)=y

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 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

```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

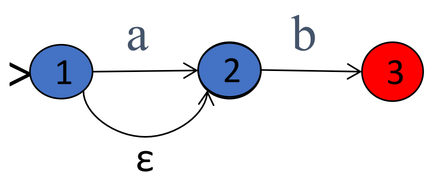
Finite State Automata (FSA)

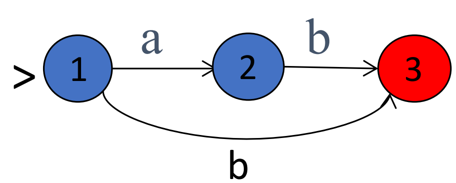
- **Practical applications**
  - *can be encoded and run efficiently on a computer*
  - *widely used*
  - **encode regular expressions (e.g. Perl regex)**
  - **morphological analyzers**
    - Different word forms, e.g. want, wanted, unwanted (suffixation/prefixation)
    - *see chapter 3 of textbook*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

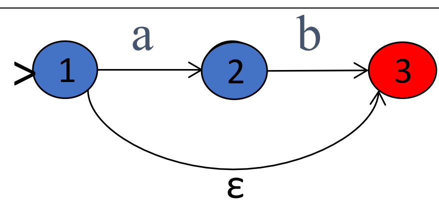
  - *and much more …*

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

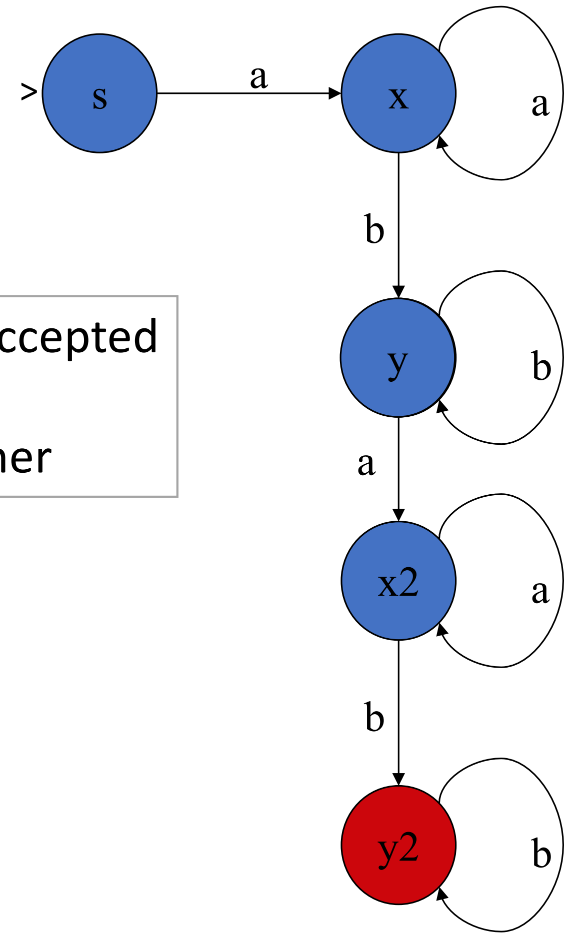
- Next time, we'll see:
  - Then we can get rid of the ε-transition (*by construction*)

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`

```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`