

LING/C SC/PSYC 438/538

Lecture 17


Sandiway Fong

Today's Topics

- Regex example:
 - `xkcd simplewriter`
- FSA *contd.*
 - formal definition
 - Perl and Python implementations
 - e-transitions
 - single vs. multiple start states

xkcd:simplewriter

<https://xkcd.com/simplewriter/>

 SIMPLE WRITER <i>WRITE LIKE UP GOER FIVE AND THING EXPLAINER</i>
PUT WORDS HERE Let me explain why this is a good idea. Writing simpler helps people understand complicated concepts .
YOU USED SOME LESS SIMPLE WORDS complicated concepts



RELATIVITY
THE SPECIAL AND GENERAL THEORY

BY
ALBERT EINSTEIN, Ph.D.
PROFESSOR OF PHYSICS IN THE UNIVERSITY OF BERLIN

TRANSLATED BY
ROBERT W. LAWSON, M.Sc.
UNIVERSITY OF SHEFFIELD



NEW YORK
HENRY HOLT AND COMPANY
1921



xkcd:simplewriter

PUT WORDS HERE

THE **present** book is **intended**, as far as possible, to give an **exact insight** into the **theory** of **Relativity** to those readers who, from a **general scientific** and **philosophical** point of view, are interested in the **theory**, but who are not **conversant** with the **mathematical apparatus** of **theoretical physics**. The work **presumes** a **standard** of **education** corresponding to that of a **university matriculation examination**, and, despite the **shortness** of the book, a **fair amount** of **patience** and force of will on the part of the reader. The **author** has **spared** himself no pains in his **endeavour** to **present** the main ideas in the simplest and most **intelligible** form, and on the whole, in the **sequence** and **connection** in which they actually **originated**.

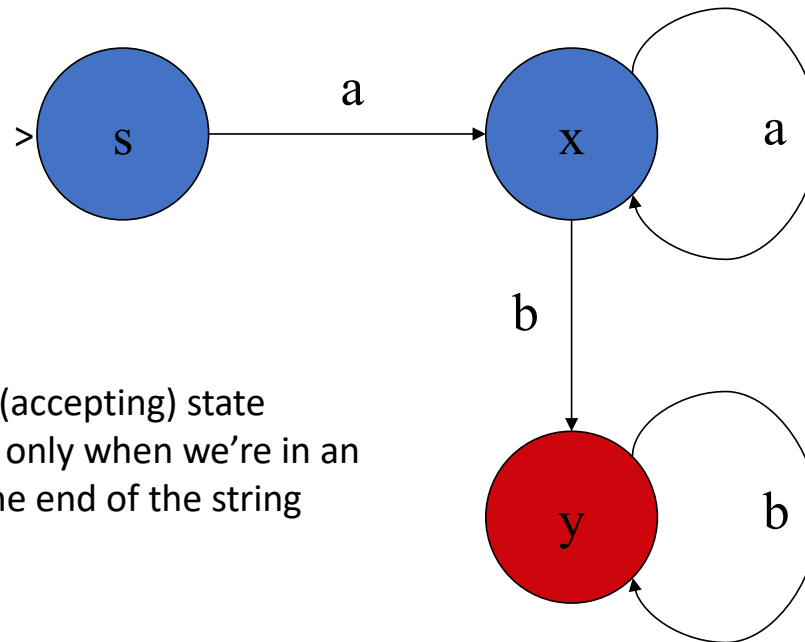
xkcd:simplewriter

- `grep -o '|' words.js | wc -l`
- **3633** (3634 words)

```
5window.__WORDS = "understandings|understanding|conversations|disappearing|info...  
rmations|grandmothers|grandfathers|questionings|conversation|information|appro...  
aching|understands|immediately|positioning|questioning|grandmother|travellings...  
|questioners|recognizing|recognizers|televisions|remembering|rememberers|expre...  
ssions|discovering|disappeared|interesting|grandfather|straightest|controllers...  
|controlling|considering|remembered|cigarettes|companying|completely|spreading...  
s|considered|continuing|controlled|stationing|controller|straighter|stretching...  
|businesses|somebodies|soldiering|countering|darknesses|situations|directions|...  
disappears|younglings|suggesting|afternoons|breathings|distancing|screenings|s...  
choolings|especially|everything|everywhere|explaining|explainers|expression|br...  
anchings|revealings|repeating|surprising|rememberer|somewheres|television|the...  
mselves|recognizer|recognizes|recognized|belongings|finishings|travelling|ques...  
tioner|beginnings|travelings|questioned|followings|pretending|forgetting|forge...  
tters|forwarding|positioned|travellers|gatherings|perfecting|understand|unders...  
tood|weightings|approaches|officering|numberings|happenings|mentioning|letteri...  
ngs|husbanding|imaginings|approached|apartments|whispering|interested|discover...  
ed|spinnings|clearings|climbings|spendings|clothings|colorings|soundings|truck...  
ings|somewhere|troubling|companies|companied|beautiful|computers|confusing|con...  
siders|travelers|youngling|continues|continued|traveller|traveling|yellowing|a...  
partment|beginning|wheelings|travelled|sometimes|something|appearing|cornering...
```

Finite State Automata (FSA)

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

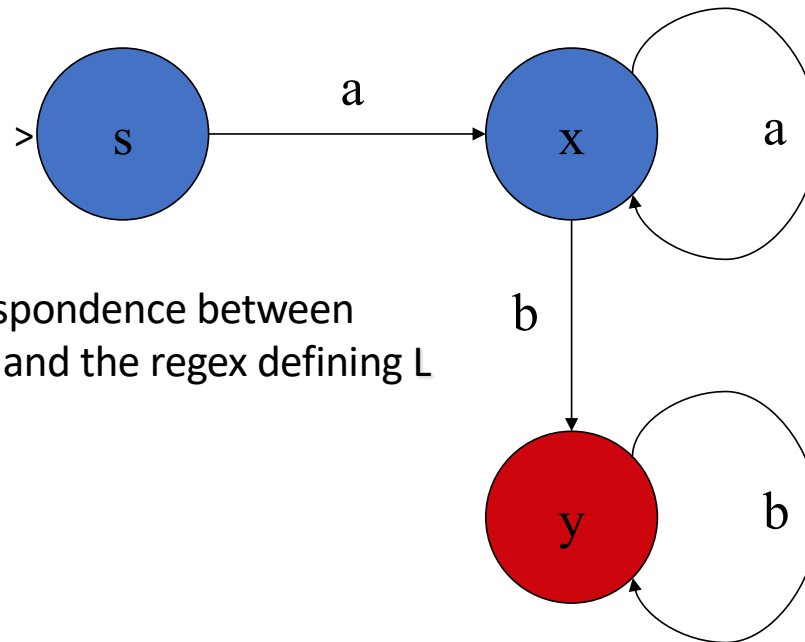


Conventions (*used here*):

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

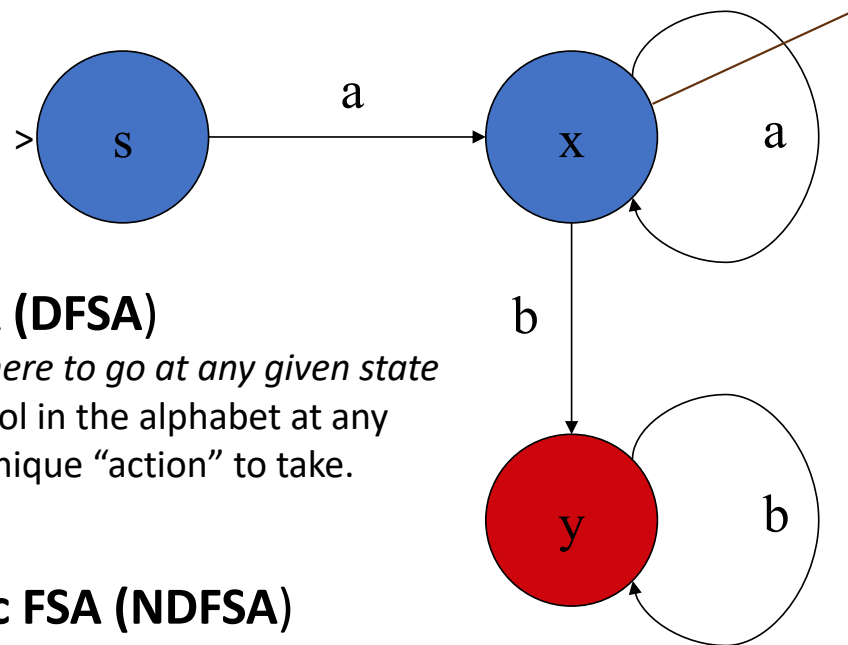
Note:

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

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

Finite State Automata (FSA)

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



Note: multiple exiting arrows, i.e. *two paths*, but still **deterministic!**

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 (**Σ**): $\{a, b\}$

5. transition function δ :

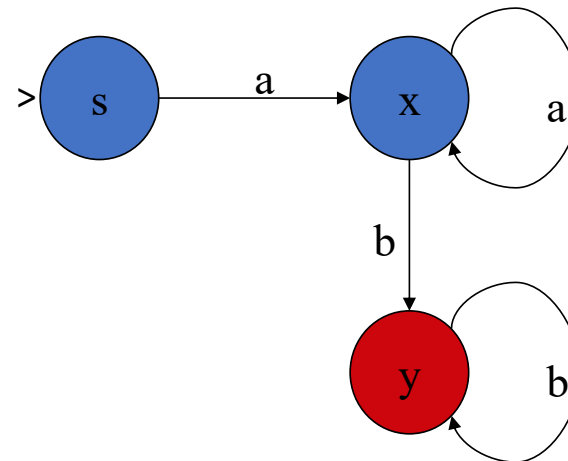
signature: character \times state \rightarrow state

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

- $\delta(a, x) = x$

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

- $\delta(b, y) = y$



Finite State Automata (FSA)

- In Perl**

transition function δ :

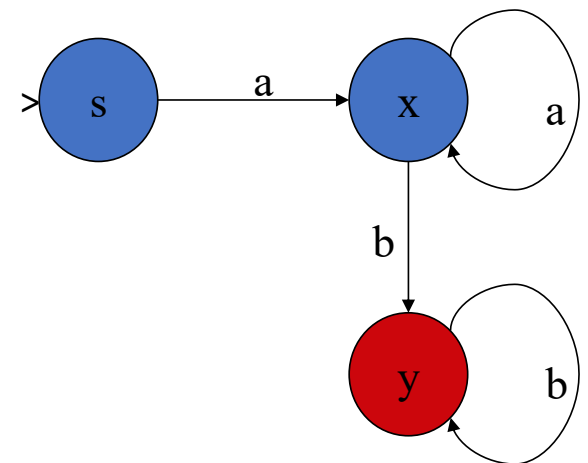
- $\delta(a,s)=x$
- $\delta(a,x)=x$
- $\delta(b,x)=y$
- $\delta(b,y)=y$

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 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.pl a b a b
- **Reject**
- perl fsm.pl 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 1610892 Sandiway Fong
```

```
    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
2 import sys
3 tt = {'s': {'a': 'x'}, 'x': {'a': 'x', 'b': 'y'}, 'y': {'b': 'y'}}
4 state = 's'
5 for 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
12 if state == 'y':
13     print "Accept"
14 else:
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

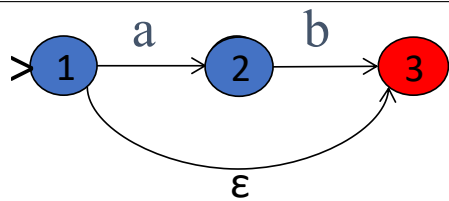
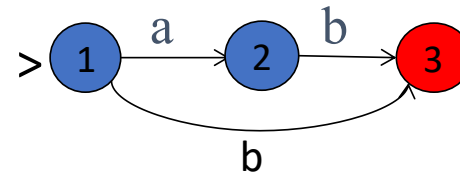
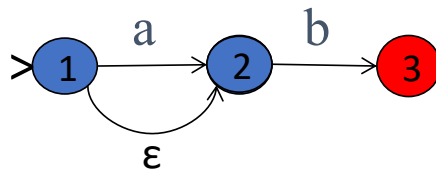
Finite State Automata (FSA)

- **Practical applications**

- *can be encoded and run efficiently on a computer*
- *widely used*
- **encode regular expressions (e.g. Perl regex)**
- **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

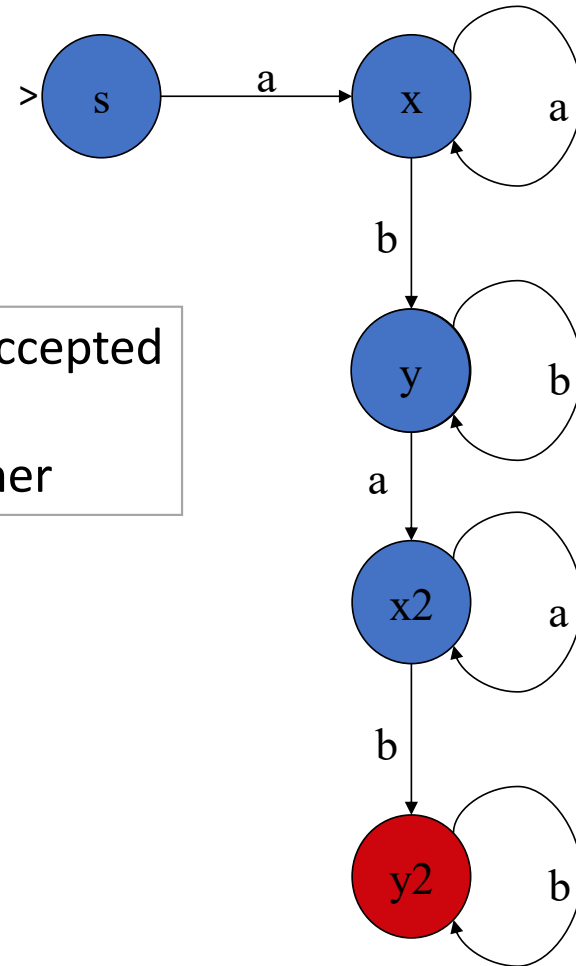
- Can be used to help encode:
 1. Multiple start states
 2. Multiple end states
- Next time, we'll see:
 - Then we can get rid of the ε -transition (*by construction*)

Backreferences and FSA

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

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

Doesn't work!
Why?



Backreferences and FSA

- `fsa.perl`

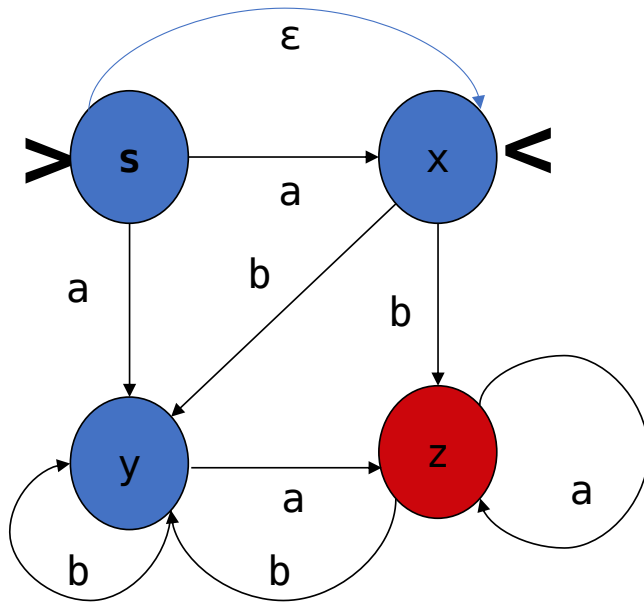
```
1 %delta = (
2     s => { a => "x" },
3     x => { a => "x", b => "y" },
4     y => { b => "y", a => "x2" },
5     x2 => { a => "x2", b => "y2" },
6     y2 => { b => "y2" });
7 $state = "s";
8
9 foreach $c (split(//, @ARGV[0])) {
10     $state = $delta{$state}{$c};
11 }
12
13 print (($state eq "y2") ? "Accept\n" : "Reject\n");
```

Perl:

- note line 10: next state is a function of previous state and current symbol **ONLY**
- \therefore # of a's and b's in the two halves don't have to match:
- `perl fsa.perl aabba`
- **Reject**
- `perl fsa.perl aabbbaaabbbb`
- **Accept**
- `perl fsa.perl aabbbaaab`
- **Accept**

Multiple start states

- Example: simulate this by using an ϵ -transition:



- Multiple final states vs. a single state: also same expressive power.
- Doesn't have to have any final states at all:
 $L(\text{machine}) = \{\}$
- What's the simplest possible FSA?