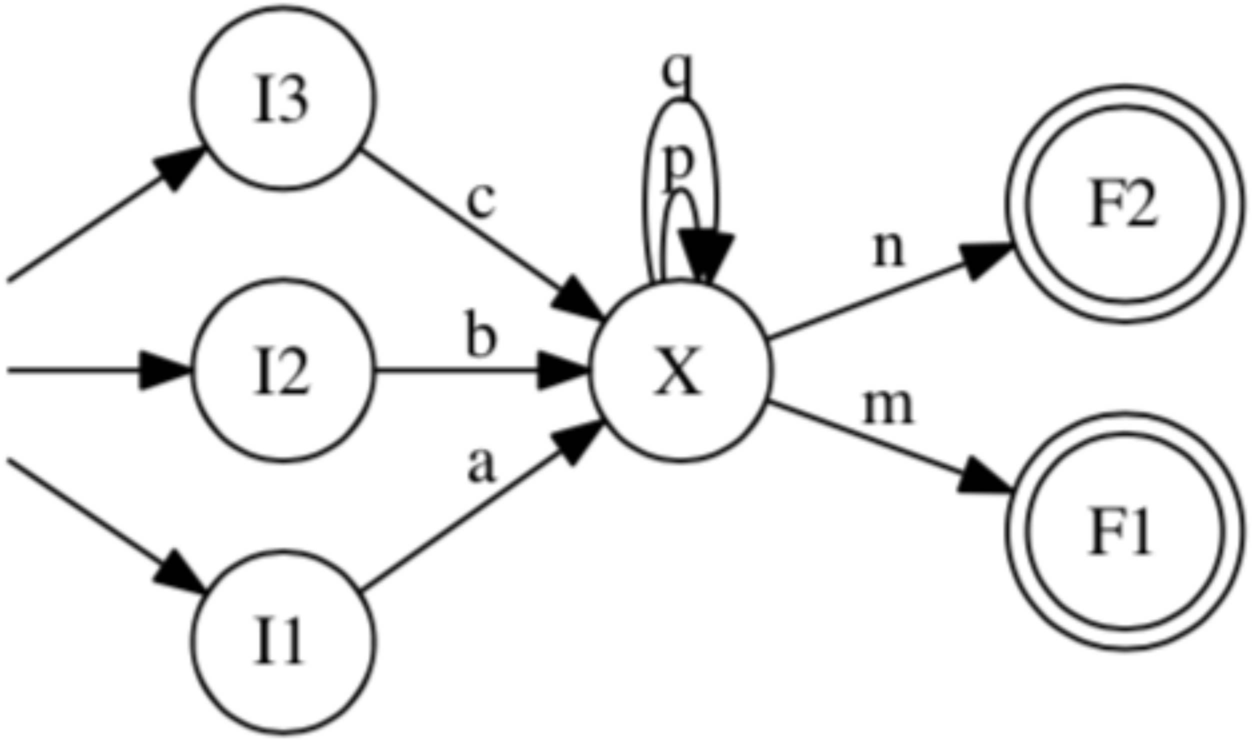
CS 3100, Models of Computation, Spring 20, Lec 11

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

cp ../Lec10.pptx ./Lec11.ppt

bit.ly/3100s20Syllabus

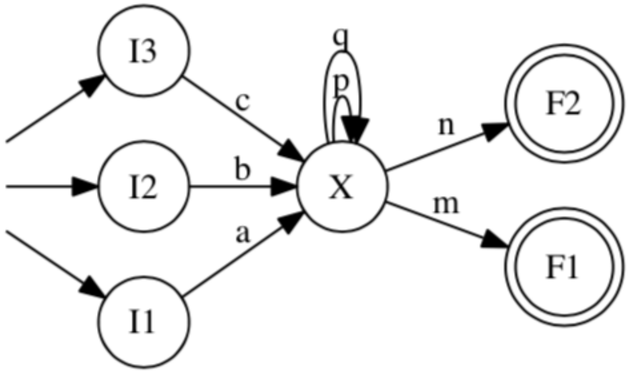
NFA to RE conversion



What is the language of this NFA?

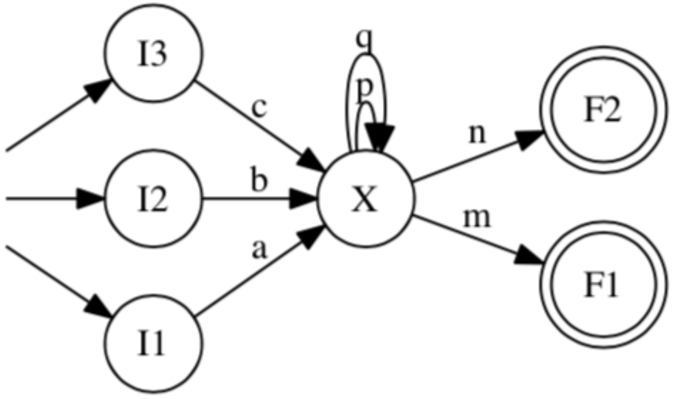
Let’s convert this to an RE and check our work against this RE

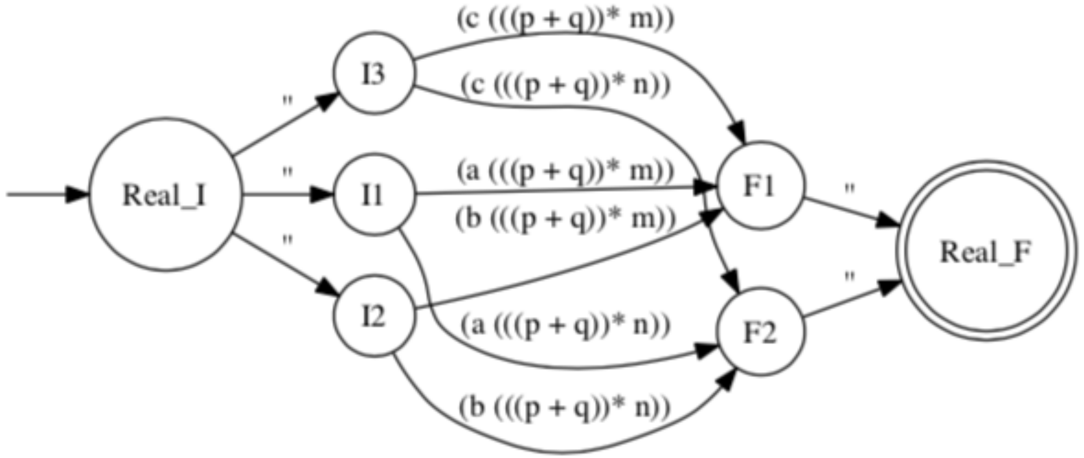
NFA to RE conversion

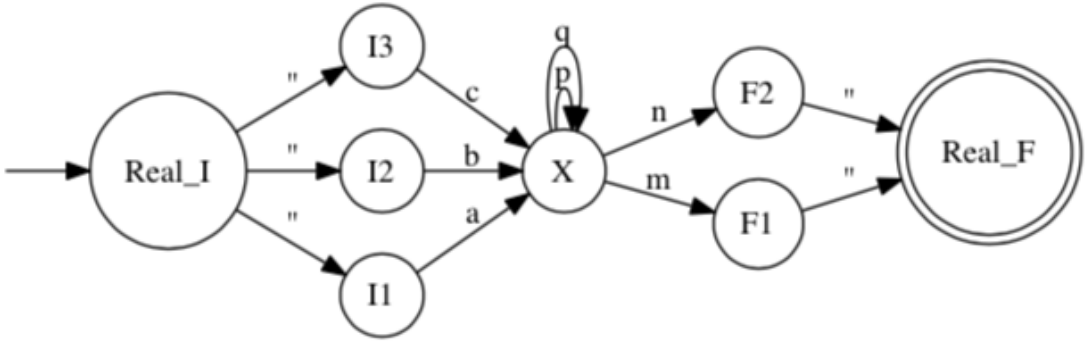


(space for work)

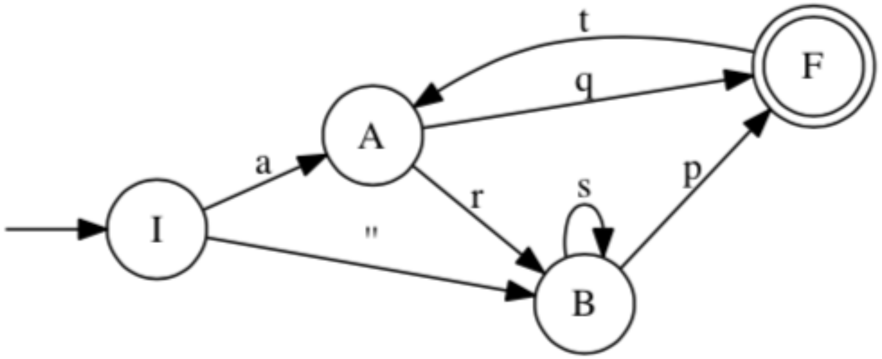
NFA to RE conversion: Results







Another example of NFA to RE



(space for work)

Lecture 11, covering many topics

- Ch11 basics
  - Basics of CFL and CFG
- Ch12 basics
  - Basics of PDA
  - Conversion of CFG to PDA
- Ch9 and before
  - NFA to RE and related topics

Grammars are widely used: e.g. English

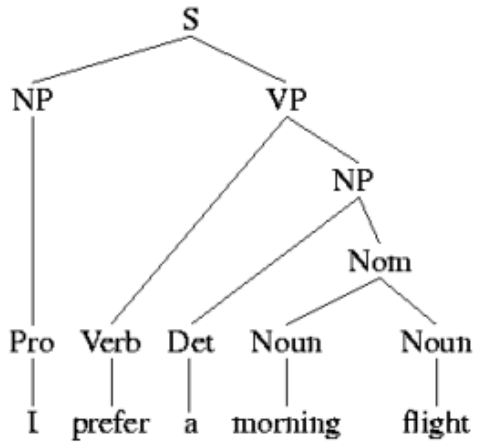
- From http://www.cs.uccs.edu/~jkalita/work/cs589/2010/12Grammars.pdf

Context-Free Rules and Trees

Noun → *flight* | *breeze* | *trip* | *morning* | ...
Verb → *is* | *prefer* | *like* | *need* | *want* | *fly* ...
Adjective → *cheapest* | *non-stop* | *first* | *latest* | *other* | *direct* | ...
Pronoun → *me* | *I* | *you* | *it* | ...
Proper-Noun → *Alaska* | *Baltimore* | *Los Angeles* | *Chicago* | *United* | *American* | ...
Determiner → *the* | *a* | *an* | *this* | *these* | *that* | ...
Preposition → *from* | *to* | *on* | *near* | ...
Conjunction → *and* | *or* | *but* | ...

The lexicon for L_0

S → *NP VP*	I + want a morning flight
NP → *Pronoun*	I
| *Proper-Noun*	Los Angeles
| *Det Nominal*	a + flight
Nominal → *Noun Nominal*	morning + flight
| *Noun*	flights
VP → *Verb*	do
| *Verb NP*	want + a flight
| *Verb NP PP*	leave + Boston + in the morning
| *Verb PP*	leaving + on Thursday
PP → *Preposition NP*	from + Los Angeles



The grammar for L_0

Grammars used for Formal Languages also

- Regular expressions (also known as Regular Grammars)
- Context-free Grammars (we are about to study this now)

- Later we will mention the Chomsky Hierarchy that includes:
  - Context-sensitive grammars
  - Unrestricted grammars

We study Languages & Grammars & Machines

- Languages versus Grammars versus Machines
- Regular languages – Regular Expressions – NFA
- Context-free languages – Context-free Grammars - PDA

Example of design

- Language: L_wwr = { ww^R for w in {a,b}* }
  - PL proof of non-regularity
    - Otherwise, why bother designing anything but a RegExp or NFA ??
- Design a Context-free Grammar for it
  - Recursive programming that generates strings in L_wwr
- Design a PDA
  - Direct design
    - Stack things
    - Nondet switch to begin matching
  - Design by converting CFG to PDA
    - Standard design for any CFG

Language L_wwr and PL proof of non-reg.

A CFG for L_wwr

How to read/interpret a PDA

- Each PDA carries an additional state : the stack state
  - The stack keeps changing as the PDA runs
  - There is no theoretical depth-bound for the stack
    - In practice, Jove limits it… to finish running on time
  - Thus the total state is: < Control state (or "circle"), Stack state >
- Each transition has
  - Input , Top-Of-Stack ; Stack-Push-String
- If the Top-Of-Stack field is '' , then works like an NFA
  - Except the Stack-Push-String may get pushed
- If the Top-Of-Stack field is a member of Sigma, then the runtime stack's top must match that Sigma entry for the transition to fire
  - When the transition fires, the Stack-Push-String is pushed in

Example PDA

A PDA directly from L_wwr

A PDA by converting L_wwr's CFG to a PDA

Basics

- A CFL is the language of a PDA
- A CFL is the language defined by a CFG

- For every PDA there is an equivalent CFG, and Vice-Versa

Grammars

- What we are going to study are **formal grammars**

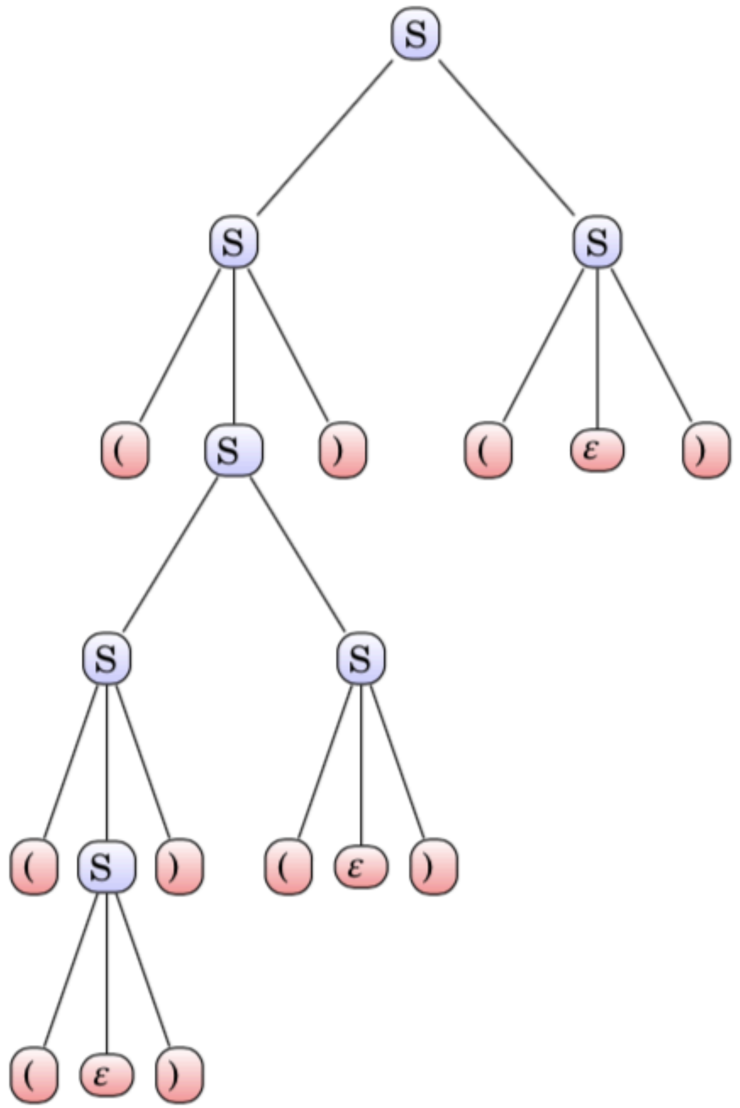
Example from Formal Languages:

S -> (S) | ‘’

A Formal Grammar and its Parse Tree

```
S  ->  ''  |  (S)  |  SS
```

Parse trees depict how a sentence in the language of the grammar can be derived by applying the grammar rules.



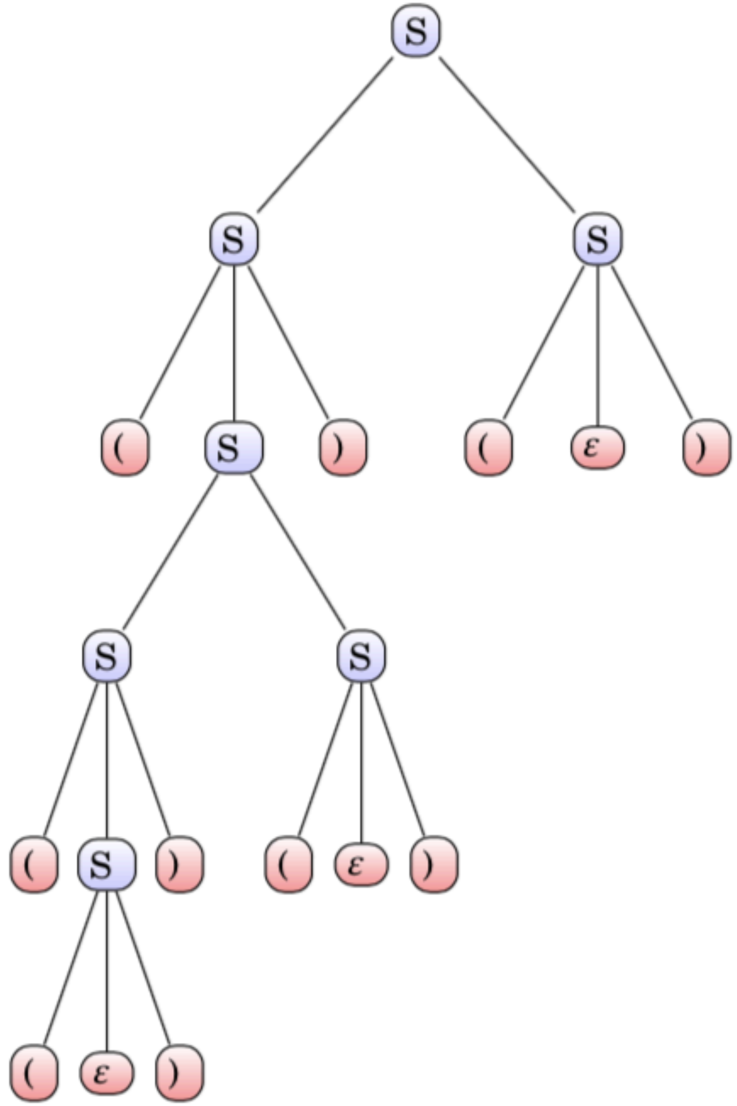
Contrast this grammar to

S1 -> (S1) | ‘’

Is Language(S1) contained in Language(S) or vice-versa?

Context-free Grammars: Derivation Sequences

S => SS => (S)S => (SS)S => ((S)S)S =>
(((S))S)S => (((”))S)S => ((())S)S => ((())(S))S =>
((())(”))S => ((())())S => ((())())(S) => ((())())(”) =>
((())())()



Important Notational Convention

Instead of writing three CFG productions, such as

S → ‘’
S → (S)
S → SS

We write one short-hand

S → ‘’ | (S) | SS

IT IS ONLY A SHORT-HAND !! MEANING THERE ARE 3 PRODUCTIONS HERE !!

Exercises

- Design a CFG for the set of all odd-length strings over {0,1}
    - Start with a recursive programming mind-set
    - Basis case: smallest odd-length is 0 or 1
    - Else it is …. What is the recursive rule? Can you think of two recursive rules?

Exercises

- Design a CFG for the set of all odd-length strings over {0,1}
  - Recursive rule 1:
    - Odd length = a smaller odd-length of 1 and then TWO MORE

ExerciseS

- Design a CFG for the set of all odd-length strings over {0,1}
  - Recursive rule 2:
    - Odd length = an Even length grammar and then ONE MORE

Solutions for Odd Length: All are OK for us

- OddL -> Len1 Len1 OddL | Len1
- Len1 -> 0 | 1

- OddL -> OddL Len1 Len1 | Len1
- Len1 -> 0 | 1

- OddL -> Len1 OddL Len1 | Len1
- Len1 -> 0 | 1

In general, right-most recursion is easier for humans to wrap their head around

For compiler parser generators, it does not matter.

Related Exercise

- Design CFG for Even Length
- Then design Odd Length in terms of Even Length

Solution for Even and Odd

- OddL -> Ch EvenL
- EvenL -> Ch Ch EvenL | ‘’
- Ch -> 0 | 1

- Another approach : control how the basis case ends!
- EvenL -> Ch Ch EvenL | ‘’
- OddL -> Ch Ch OddL | Ch

Exercises

CFG for all even-length strings that end in a 1

Again break up the rules into convenient “subroutines”

EvEnd1 -> SomeNonTerminal 1

What must SomeNonTerminal be?

Exercises

EvEnd1 -> SomeNonTerminal 1

SomeNonTerminal has to be Odd...

Exercise: a^n b^n

- Grow the strings inside-out

Exercise: Equal number of a’s and b’s

- Begin with template

- S -> …..a…..b….. Or S -> …..b….a…..

- CFGs have to “tally at the top” and then recurse within

Equal a’s and b’s

- Finally we can do a CFG of the form
- S -> aSbS | bSaS | ‘’

Converting a CFG to a PDA

- Set up “S” (the top level symbol) as the parsing goal
- Pop the current parsing goal
- Push the RHS of the rules

Converting a CFG to a PDA

Given a grammar S -> aSbS | bSaS | ‘’

md2mc(”’PDA
I : ‘’,”; S -> Work
Work : ‘’, S -> aSbS
…
’’’)

Grammars vs. Ambiguity

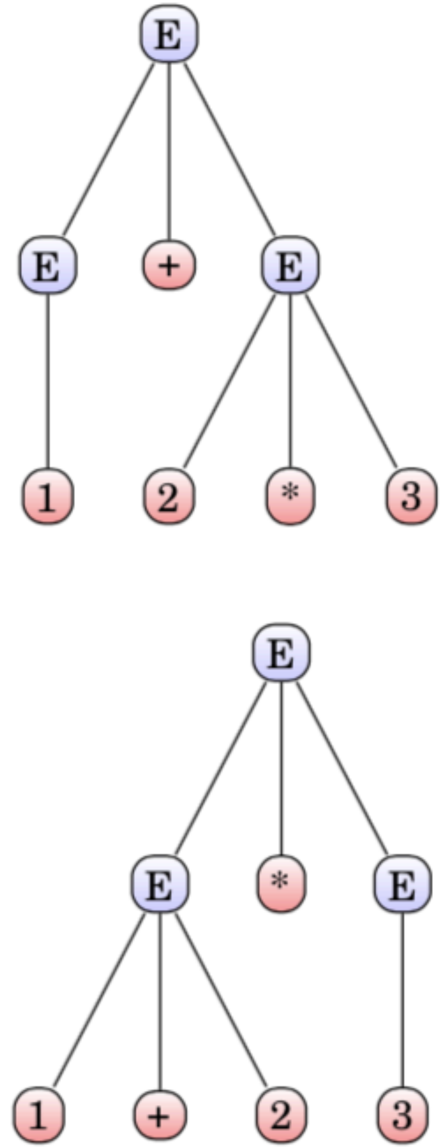
- A grammar G1 may be ambiguous
- Another grammar G2 such that L(G1) = L(G2) may be unambiguous
  - I.e. no string has two distinct parse trees

- While L(G1) = L(G2), there is only one parse-tree for L(G2)
- Parse trees determine how
  - A calculator evaluates
  - A compiler generates code
- Let us review the expression grammar (next slide)

Ambiguity and Disambiguation

```
E -> 1 | 2 | 3 | ~E | E+E | E*E | (E)
```

```
E -> E+T | T
T -> T*F | F
F -> 1 | 2 | 3 | ~F | (E)
```



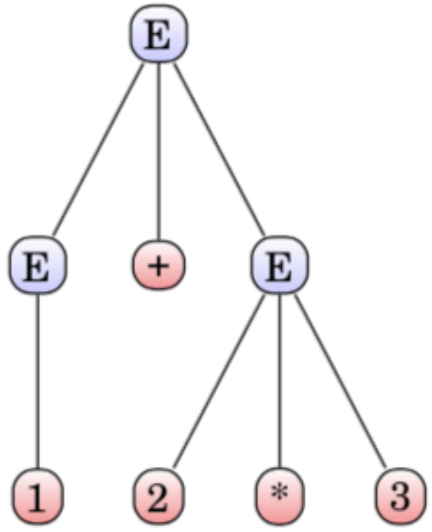
Ambiguity and disambiguation

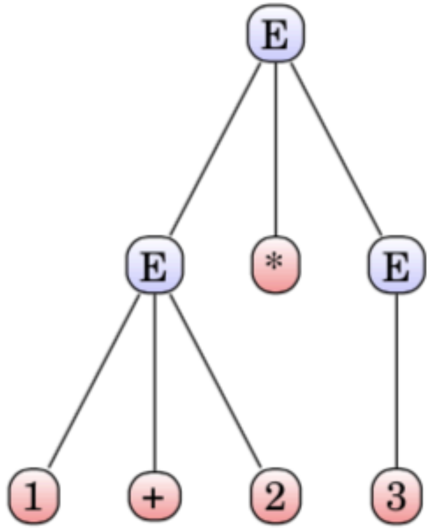
```
E -> 1 | 2 | 3 | ~E | E+E | E*E | (E)
```

```
E -> E+T | T
T -> T*F | F
F -> 1 | 2 | 3 | ~F | (E)
```

Gist : by changing the grammar,

- The same set of strings are still derivable
- Ambiguity goes away !!
- The basic idea is to "layer the grammar"





Pushdown Automata

- Finite control (like DFA/NFA)
- Unbounded stack added
- The stack models the recursion stack (in a prog language)
- Allows us to store away information and match
- Still no “arbitrary counting” (other than matching in stack order)

PDA ex

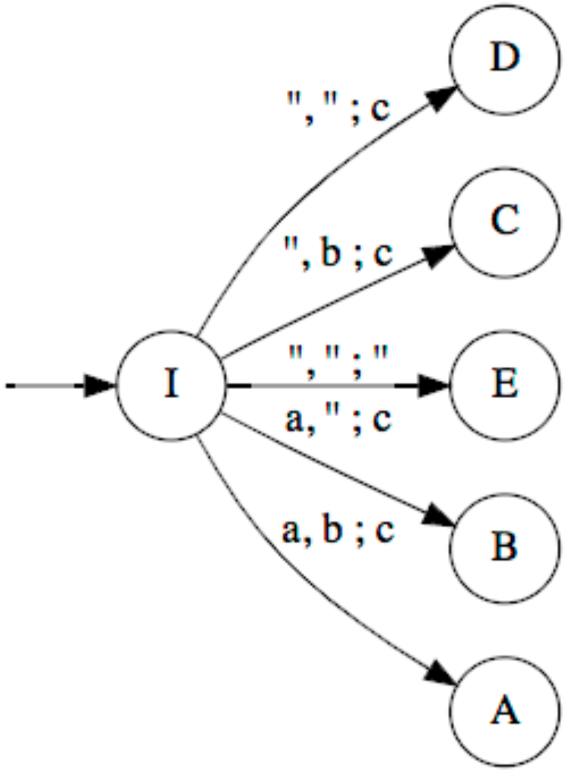
In [4]:

```
pdaex1 = md2mc('''PDA
I : a, b; c -> A

I : a, ''; c -> B

I : '', b; c -> C

I : '',''; c -> D

I : '',''; '' -> E
''')
```

In [6]:

```
dotObj_pda(pdaex1)
```

Out[6]:



How our PDAs are set up

- The input is as before
  - Contains the string to be examined
- The stack is an unbounded last-in first-out stack
  - Like any other unbounded stack
- We initialize the stack with #
  - A single character # sits on top of the stack when the PDA is powered up
  - The stack has nothing else (i.e. the stack has exactly one thing – the #)
  - Whenever # is on top of the stack, we know that the stack is empty
  - When we something else on top of the stack, we know it is not empty
  - … see next slide for more facts…

How our PDAs are set up

- We initialize the stack with #
  - A single character # sits on top of the stack when the PDA is powered up
  - The stack has nothing else (i.e. the stack has exactly one thing – the #)
  - Whenever # is on top of the stack, we know that the stack is empty
  - When we something else on top of the stack, we know it is not empty
- We put something on the stack by pushing it
  - Only one character (symbol) at a time is pushed
- We remove by popping
  - Only one symbol is popped
- When we pop all we pushed, we see # reappear on top of the stack
  - Then we know the stack is empty!
  - That is the ONLY test for stack emptiness

How a PDA accepts a string

- In every state (a “single circle” or ”double circle”)
  - It CAN looks at both the input
  - And the stack top
- In every state
  - It CAN ALSO IGNORE THE INPUT
  - It CAN ALSO IGNORE THE STACK
  - It can ignore both
  - It can ignore neither
- It chooses a step based on how you have programmed the PDA
  - Programming the PDA means providing it with transitions
  - …more facts next slide…

How a PDA accepts a string

- The PDA is deemed to have accepted a string when it "accepts by final state"
- A PDA is also deemed to have accepted a string when it empties the stack – we will NEVER study this idea in this course .

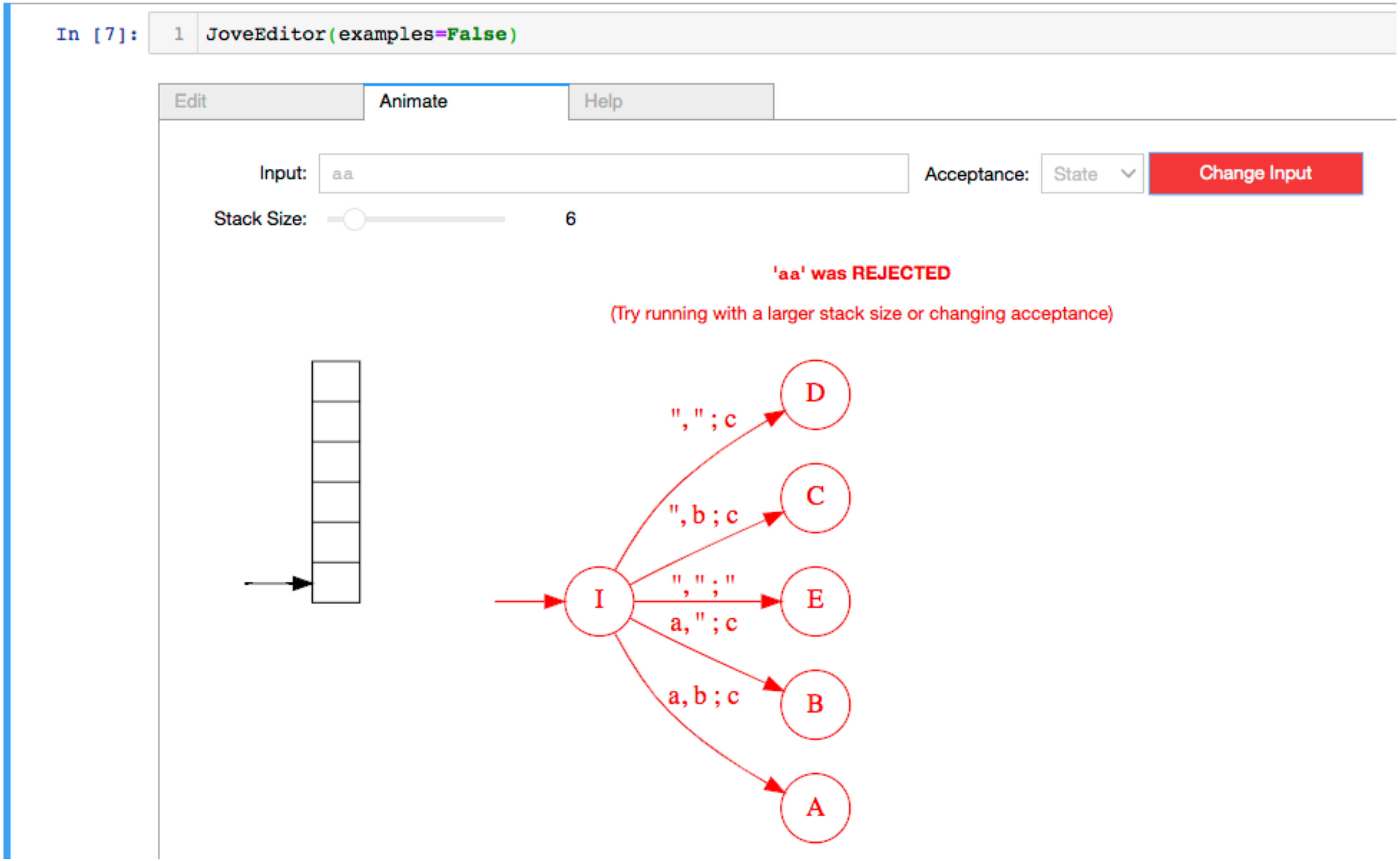
How a PDA accepts a string

- The PDA is deemed to have accepted a string when it “accepts by final state”
- ALL OUR PDAs are non-deterministic
  - Deterministic PDAs are there
  - They are useless for us
    - Some others care about them

How a PDA accepts a string

- The PDA is deemed to have accepted a string when it “accepts by final state”
- ALL OUR PDAs are non-deterministic
  - Deterministic PDAs are there
  - They are useless for us
    - Some others care about them
- THEREFORE we can say
  - A PDA accepts a string when ONE OF ITS NON-DETERMINISTIC journeys ends up in a final state (an “F” or “IF” state)
    - With the input all gone – fully consumed
    - The stack may have stuff in it or nothing in it
    - The contents of the stack are immaterial when the PDA “accepts”
      - I.e. acceptance == IN A FINAL STATE + INPUTS ALL-GONE!

PDA ex2



In [7]:
1 JoveEditor(examples=False)
Edit
Animate
Help
Input:
aa
Acceptance:
State
Change Input
Stack Size:
6
'aa' was REJECTED
(Try running with a larger stack size or changing acceptance)
D
"," ; c
C
", b ; c
I
"," ; "
E
a, " ; c
B
a, b ; c
A

PDA ex3

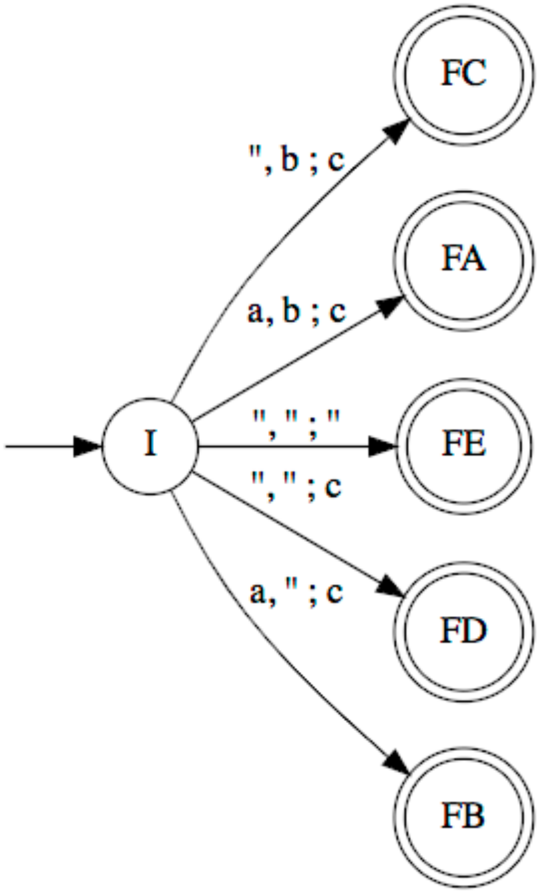
In [9]:

```
pdaex2 = md2mc('''PDA
I : a, b; c -> FA

I : a, ''; c -> FB

I : '', b; c -> FC

I : '',''; c -> FD

I : '',''; '' -> FE
''')
```

In [10]:

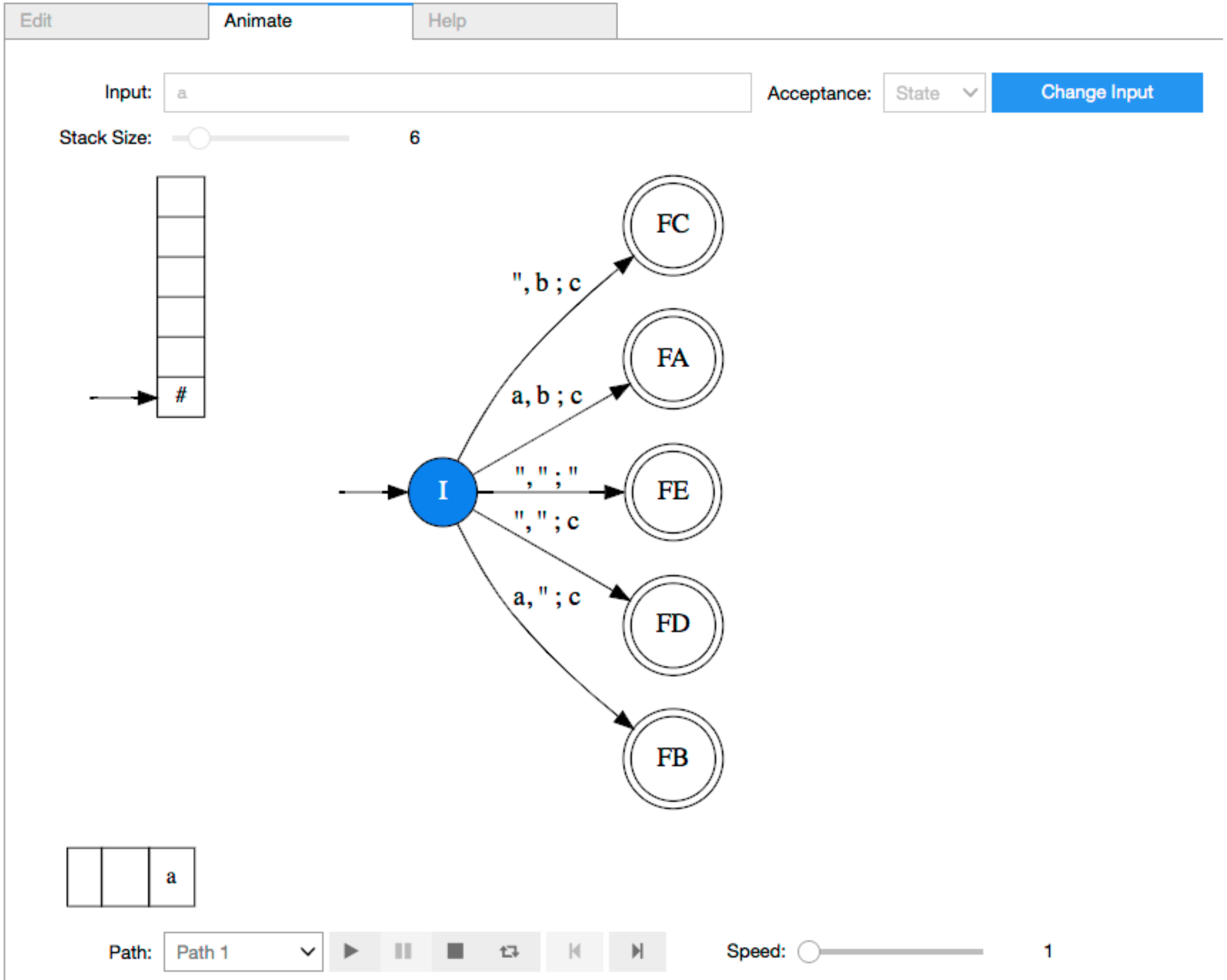
```
dotObj_pda(pdaex2)
```

Out[10]:



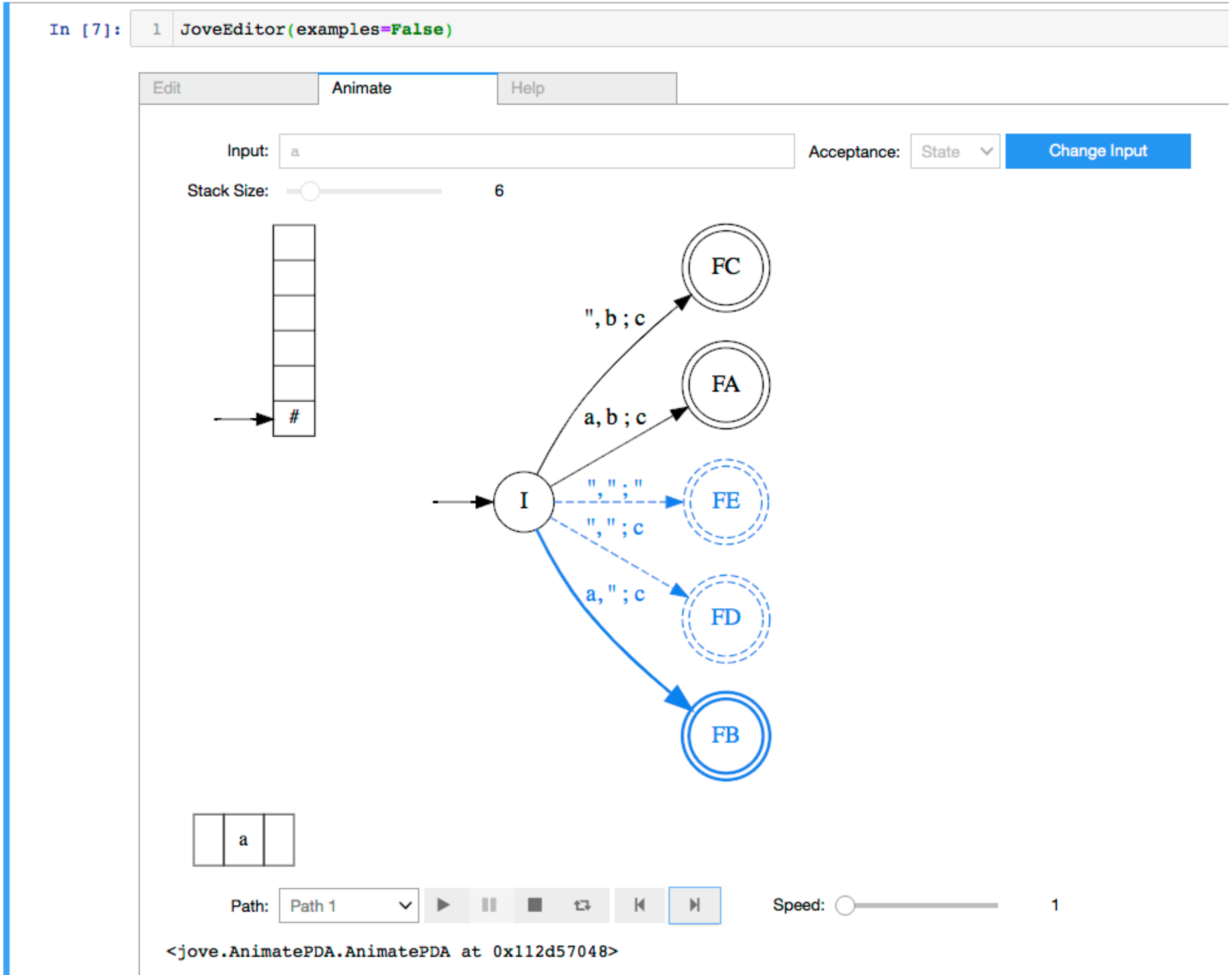
PDA ex4

In [7]:
```
JoveEditor(examples=False)
```



<jove.AnimatePDA.AnimatePDA at 0x112d57048>

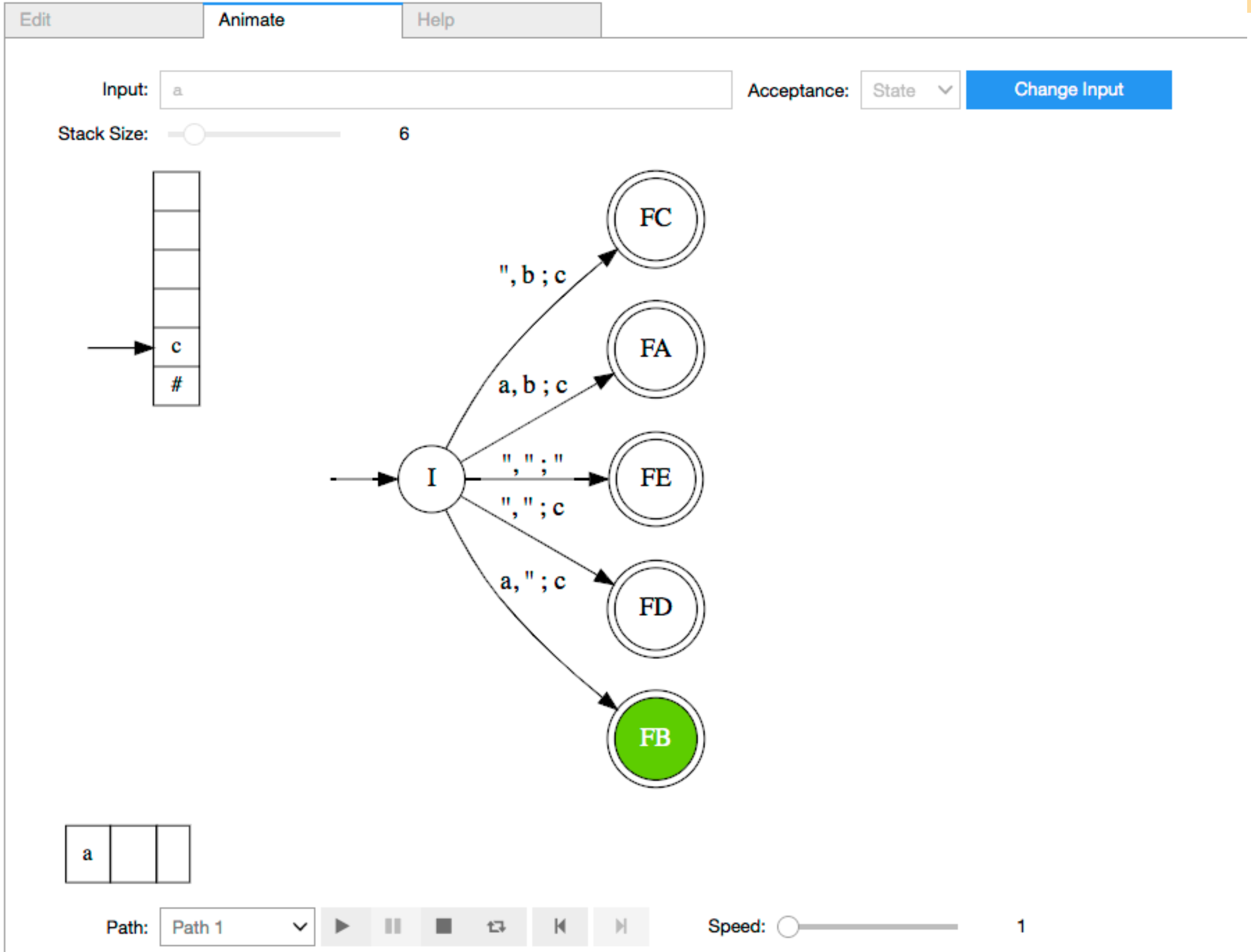
PDA ex5



In [7]:
1 JoveEditor(examples=False)
Edit
Animate
Help
Input:
a
Acceptance:
State
Change Input
Stack Size:
6
#
FC
FA
FE
FD
FB
I
'', b ; c
a, b ; c
'', '' ; ''
'', '' ; c
a, '' ; c
a
Path:
Path 1
Speed:
1
<jove.AnimatePDA.AnimatePDA at 0x112d57048>

PDA

```
In [7]:  1  JoveEditor(examples=False)
```




```
<jove.AnimatePDA.AnimatePDA at 0x112d57048>
```

Design a PDA for #a = 2 times #b

- Design a trick to do this “double match”
- Implement it exploiting nondeterminism

- Now try other variants
  - #a = 2#b + #c etc

Solution

```
# num a's = 2 times num b's
# for every b, match two a's
# idea : currency conversion: each b is converted into two c's
P2a1b = md2mc('''PDA
I : a,#; a# -> I
I : a,a; aa -> I
I : b,#; cc# -> I
I : a,c; '' -> I
I : b,a; '' -> W
W  : '',a; '' -> I
W  : '',#; c  -> I
I : '',''; '' -> F
''')
```

Design a PDA or CFG first?

- Depends on the language
- Design a PDA for #1 > #0
  - VERSUS
- Design a CFG for #1 > #0

Once you are an expert, do this

```
In [11]:  1  help(explore_pda)
```

```
Help on function explore_pda in module jove.Def_PDA:

explore_pda(inp, P, acceptance='ACCEPT_F', STKMAX=6, chatty=False)
    A handy routine to print the result of run_pda plus making
    future extensions to explore run-results.
```

Putting it all together

- Design a CFG for “equal a’s and b’s”
- Translate this CFG into a PDA using a standard algorithm that imitates parsing

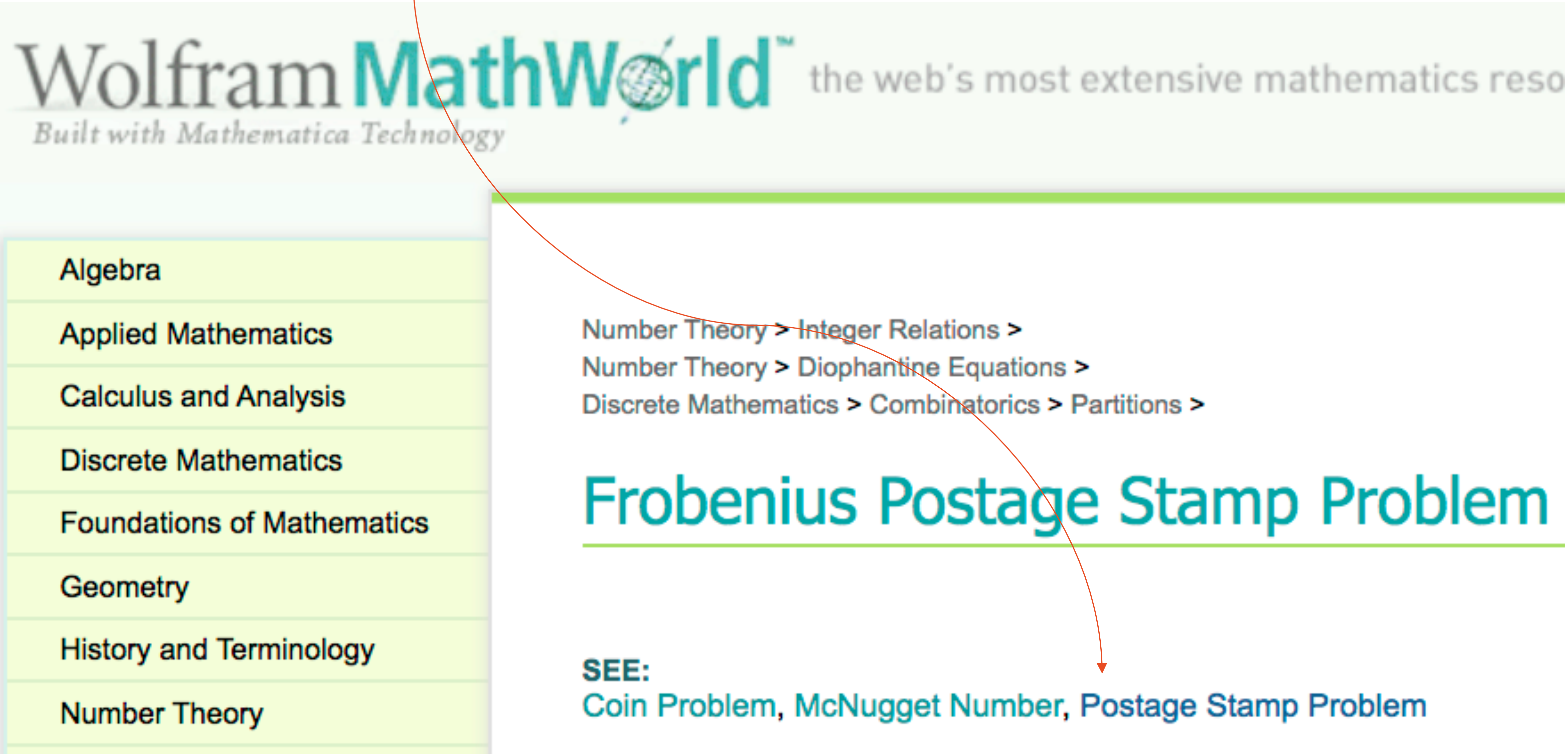
Walk the Kleene-Pipeline

$$\text{RE} \xrightarrow{\text{Study Again}} \text{NFA} \xrightarrow{\text{Review Once More}} \text{DFA} \xrightarrow{\text{Practice Once More}} \text{minDFA}$$

Brand New

Trivial, but a Useful Stepping-Stone (this way we don't need to implement a separate DFA to RE procedure)

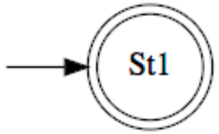
The Postage-Stamp Problem



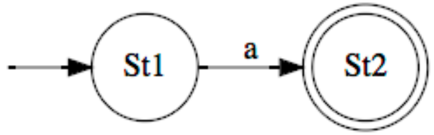
RE → NFA examples

```
dotObj_nfa(re2nfa("''"))
```

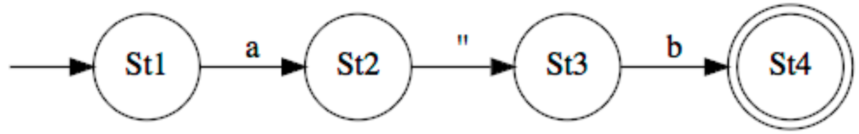
```
Generating LALR tables
```



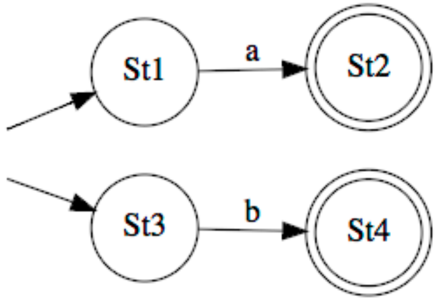

```
dotObj_nfa(re2nfa("a"))
```



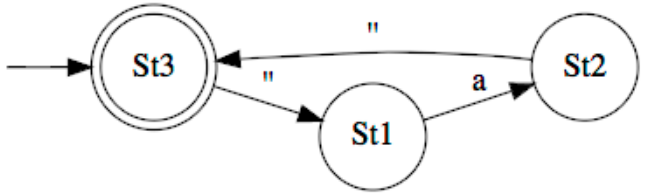

```
dotObj_nfa(re2nfa("ab"))
```




```
dotObj_nfa(re2nfa("a+b"))
```




```
dotObj_nfa(re2nfa("a*"))
```



What are RE?

- Epsilon
- a in Sigma
- If R1 and R2 are RE, then R1 + R2 is an RE
- If R1 and R2 are RE, then R1 R2 is an RE
- If R is an RE, then (R) is an RE
- If R is an RE, then R* is an RE
- Nothing else is an RE

Cover RE → NFA for each case

- Epsilon
- a in Sigma
- If R1 and R2 are RE, then R1 + R2 is an RE
- If R1 and R2 are RE, then R1 R2 is an RE
- If R is an RE, then (R) is an RE
- If R is an RE, then R* is an RE
- Nothing else is an RE

Cover RE → NFA for each case

- Epsilon
- a in Sigma
- If R1 and R2 are RE, then R1 + R2 is an RE
- If R1 and R2 are RE, then R1 R2 is an RE
- If R is an RE, then (R) is an RE
- If R is an RE, then R* is an RE
- Nothing else is an RE

Cover RE → NFA for each case

- Epsilon
- a in Sigma
- If R1 and R2 are RE, then R1 + R2 is an RE
- If R1 and R2 are RE, then R1 R2 is an RE
- If R is an RE, then (R) is an RE
- If R is an RE, then R* is an RE
- Nothing else is an RE

Cover RE → NFA for each case

- ❑ Epsilon
- ❑ a in Sigma
- ❑ If R1 and R2 are RE, then R1 + R2 is an RE
- ❑ If R1 and R2 are RE, then R1 R2 is an RE
- ❑ If R is an RE, then (R) is an RE
- ❑ If R is an RE, then R* is an RE
- ❑ Nothing else is an RE

Cover RE → NFA for each case

- Epsilon
- a in Sigma
- If R1 and R2 are RE, then R1 + R2 is an RE
- If R1 and R2 are RE, then R1 R2 is an RE
- If R is an RE, then (R) is an RE
- If R is an RE, then R* is an RE
- Nothing else is an RE

Cover RE → NFA for each case

- Epsilon
- a in Sigma
- If R1 and R2 are RE, then R1 + R2 is an RE
- If R1 and R2 are RE, then R1 R2 is an RE
- If R is an RE, then (R) is an RE
- If R is an RE, then R* is an RE
- Nothing else is an RE