CS 3100, Models of Computation, Spring 20, Lec 15

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

bit.ly/3100s20Syllabus

Pushdown Automata

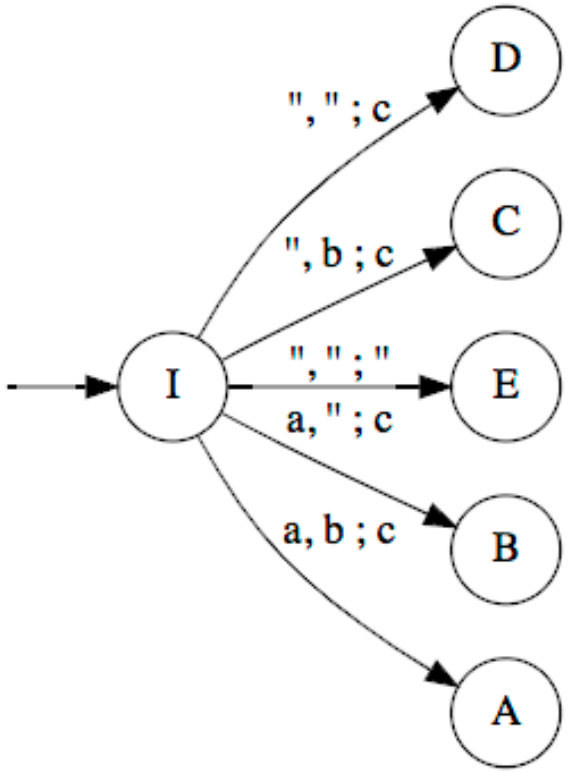
- Finite control (like DFA/NFA)
- Unbounded stack added
- The stack models the recursion stack (in a prog language)
- Allows us to store away information and match
- Still no “arbitrary counting” (other than matching in stack order)

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)
```

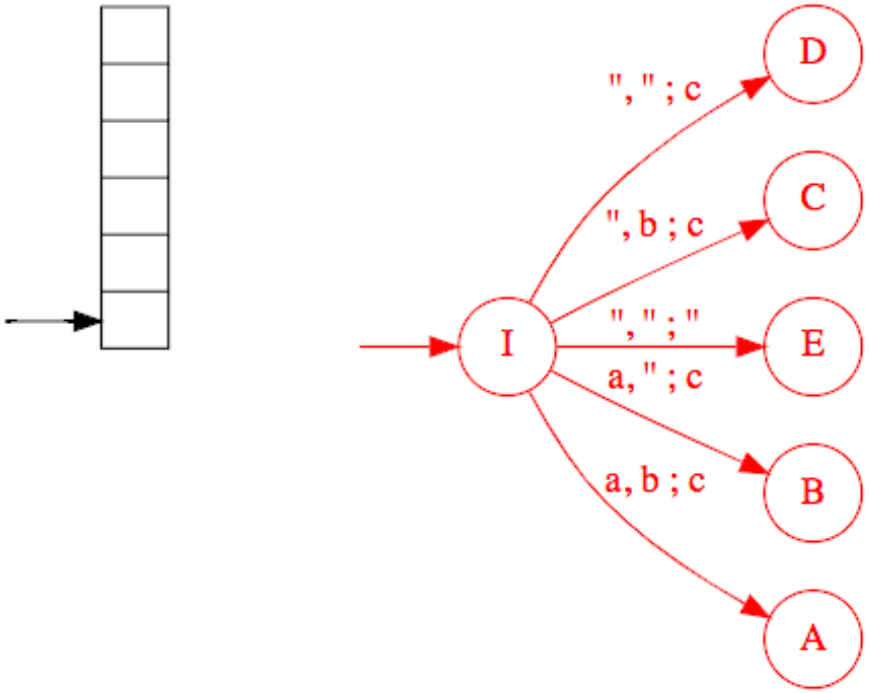
Edit | Animate | Help

Input: aa Acceptance: State Change Input

Stack Size: 6

'aa' was REJECTED

(Try running with a larger stack size or changing acceptance)

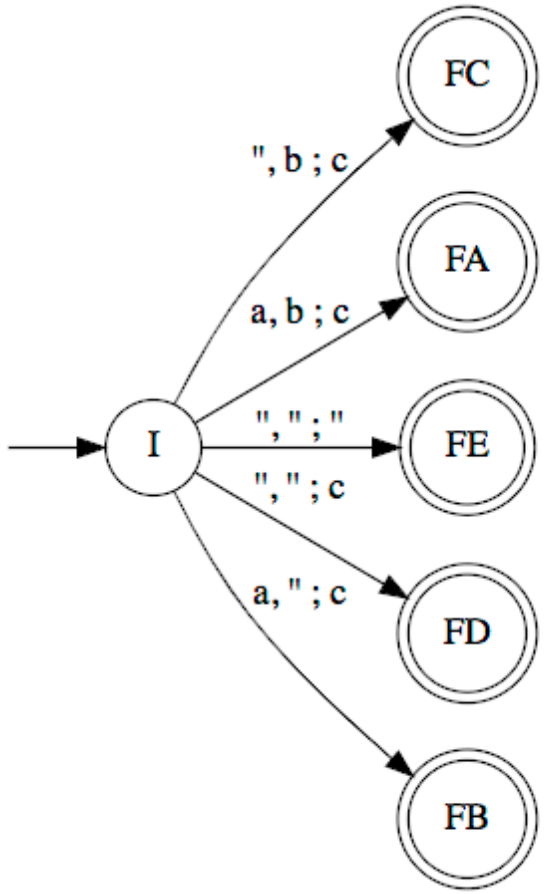


PDA ex3

In [9]:
```
pdaex2 = md2mc('''PDA
I : a, b; c -> FA

I : a, ''; c -> FB

I : '', b; c -> FC

I : '','; c -> FD

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

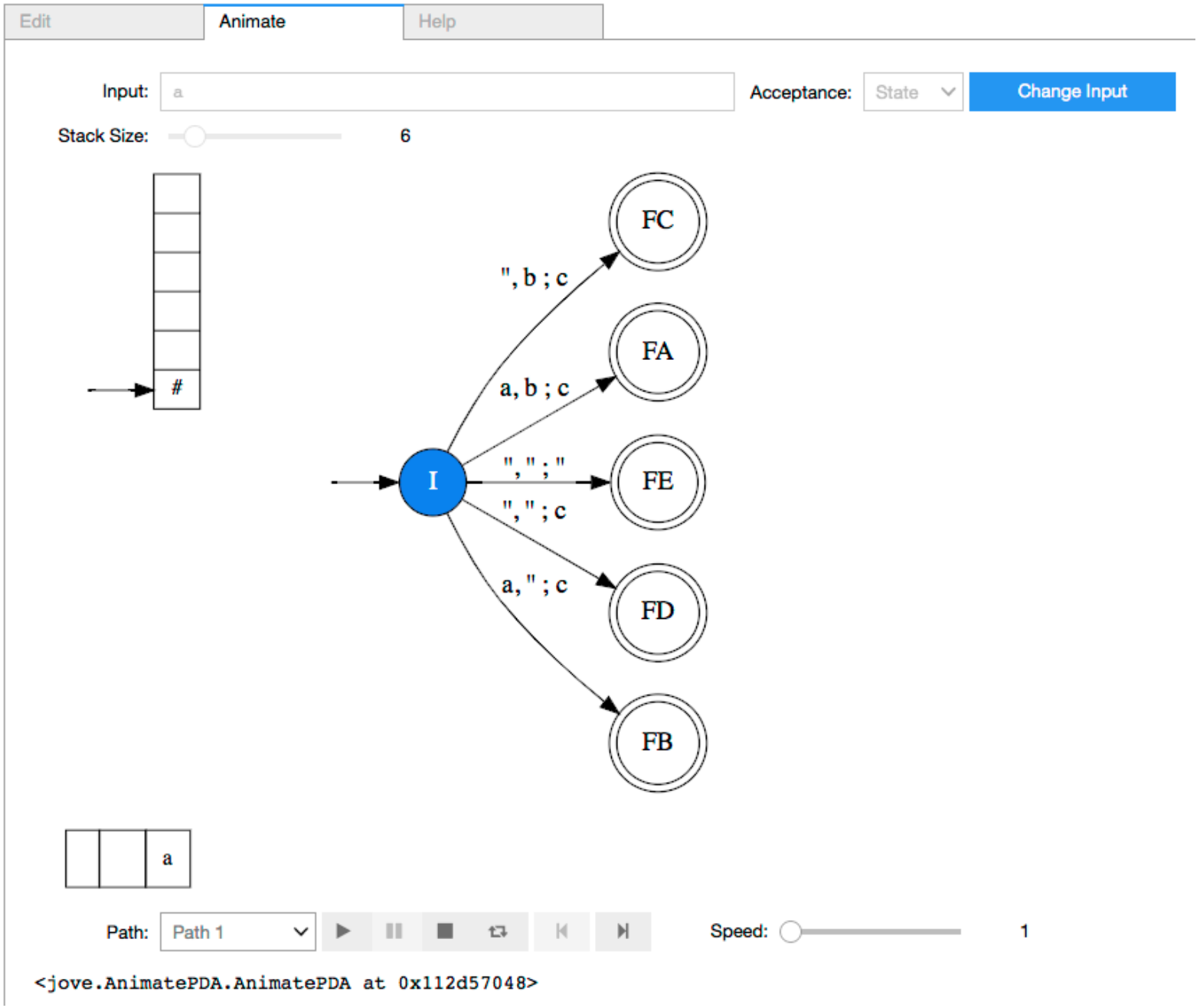
In [10]:
```
dotObj_pda(pdaex2)
```

Out[10]:



PDA ex4

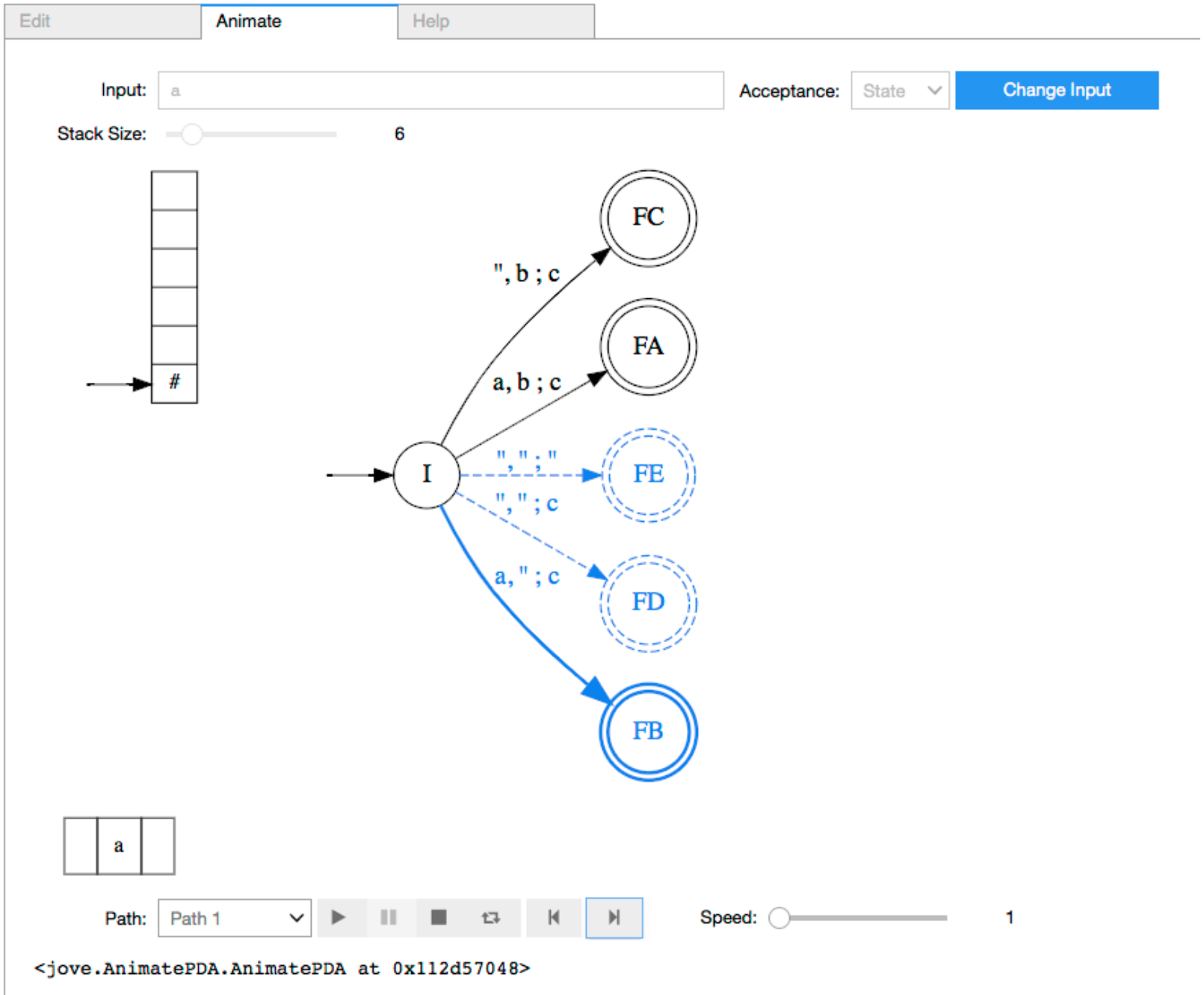
```
In [7]: 1 JoveEditor(examples=False)
```




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

PDA ex5

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




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

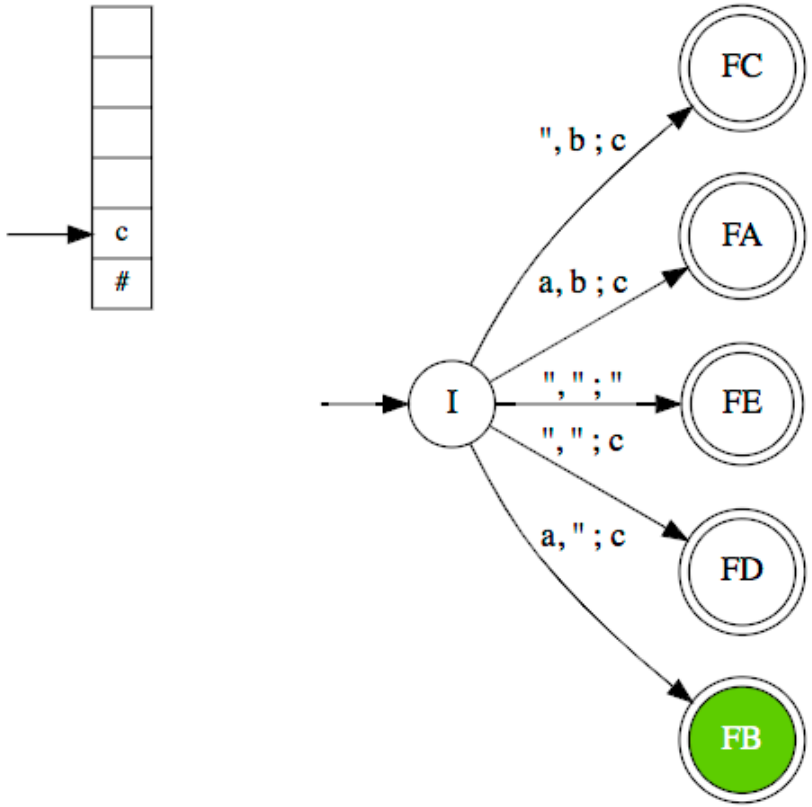
PDA

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

Edit | Animate | Help

Input: a Acceptance: State Change Input

Stack Size: 6



Path: Path 1 Speed: 1

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

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.
```

Once you are an expert, do this

```
In [14]:  1  explore_pda('a',pdaex2)
```

```
*** Exploring wrt STKMAX =  6 ; increase it if needed ***
String a accepted by your PDA in 1 ways :-)
Here are the ways:
Final state  ('FB', '', 'c#')
Reached as follows:
->  ('I', 'a', '#')
->  ('FB', '', 'c#') .
```

```
In [15]:  1  explore_pda('aa',pdaex2)
```

```
*** Exploring wrt STKMAX =  6 ; increase it if needed ***
String aa rejected by your PDA :-(
Visited states are:
{('FB', 'a', 'c#'), ('FE', 'aa', '#'), ('I', 'aa', '#'), ('FD', 'aa', 'c#')}
```

Programming a PDA for (^n)^n : n >= 0

- Algorithm
- The PDA starts with # on top of the stack
- What state should we be in when # is on top of stack initially?
  - Accept? Not accept?
- Suppose we stay in the initial state and do the push/pop game
  - And decide to accept when # shows up…
- Suppose we just provide ((( )) what must we do?

- OK we will interactively program the machine with your help!

Designing Pushdown Automata

- Like with any new programming language, there are standard approaches for programming multiple situations
- In these slides (and the accompanying Jupyter notebook), we will present many PDA designs
- These designs are significantly harder than those asked in our Asg-3. But practicing on these PDAs is highly recommended
  - Then you can discover your own programming idioms

Simple PDAs and their languages

- pdaeven1 : Checks whether the input string is even in length
  - Approach: Ignore the stack, and pretend to be an NFA for this language!
  - Interaction: Students now help define pdaodd1 – checking for odd-length inputs, pretending to be NFA
    - Allow 5 minutes and then show solution, eliciting feedback/input

Simple PDAs and their languages

- pdaeven2 : Same specification as pdaeven1
  - Approach: Keep stacking {a,b}. Then nondeterministically switch over to matching something on the stack with something in the input
  - If there is an exact match, the length of the input string must be even
  - Interaction: Students now help define pdaodd2 – checking for odd length using nondet. Hint: guess some input to be the midpoint and skip over it; then match around that input!
    - i.e. Keep stacking
    - Boom! Guess that this is the midpoint, skip over one character
      - go to another state
    - Now match what is in the stack…
    - Students finish this design at home

More Involved PDAs and their languages

- pdaabc = a PDA that implements the equation
  - # a = 2 #b + 3 #c

  - Approach:
    - Convert each b to two B's when stacking
    - Convert each c to three C's when stacking
      - i.e. "we change the currency"
    - This way, an 'a' and a 'B' have the same "value"
    - The same way, an 'a' and a 'C' have the same value

More Involved PDAs and their languages

- pdaabc = a PDA that implements the equation
  - # a = 2 #b + 3 #c
  - Approach (continued)
    - When we get a ‘b’, we must check to see if there are two stacked and waiting ‘a’ on the stack; if so, cancel; else push the “unmet obligations” on the stack
    - When we get a ‘c’, we must check to see if there are three stacked and waiting ‘a’ on the stack; if so, cancel; else push the ”unmet obligations” on the stack
    - Students can be asked to solve #a = 2 #b alone. Then introduce the Cs and then finish the construction

One more involved PDA

- PDA aibjck does this
- { a^i b^j c^k d^l : if i=2 then j=k else l > k }
- This means, we have a leading path that checks for the number of a's which could be 0, 1, 2, or more than 2
- Based on the # a's seen, we can take multiple actions
  - 0 or 1 a's : look for #d > #c
  - 2 a's : look for #b = #c
  - 3 or more a's : look for #d > #c

Concluding Remarks

- These PDAs were hastily designed (during an airplane ride)
  - They could have bugs
  - Finding and fixing them would be good
- PDAs are very low level code
  - The only way to get them right is through a rigorous mathematical proof
  - In this class, we will:
    - Document their design well
    - Test well
    - Think of all the ”invariant conditions”
      - E.g. Can the stack contain aaaBaaa etc? This is not allowed
      - Such conditions need to be thought of in looking for “2 a’s underneath”