TEACHING AUTOMATA THEORY AND FORMAL LANGUAGES WITH DAFNY

R. ETTINGER

DAFNY 2026
JANUARY 11, 2026 | POPL | RENNES, FRANCE

The Traditional Challenge

Traditional Automata Theory Courses:

-  Abstract mathematical definitions on paper
-  Hand-drawn state diagrams
-  Manual execution traces
- ❓ “Does my machine actually work?”

Student Experience:

- Errors discovered days later (after submission)
- Debugging on paper is tedious and error-prone

Dafny to the Rescue

What if students could:

- ✅ **Run** their automata on test inputs?
- ✅ **Debug** with immediate feedback?
- ✅ **Verify** correctness formally?
- ✅ **Understand** through hands-on implementation?

Answer: Use Dafny!

My Formal Methods Journey

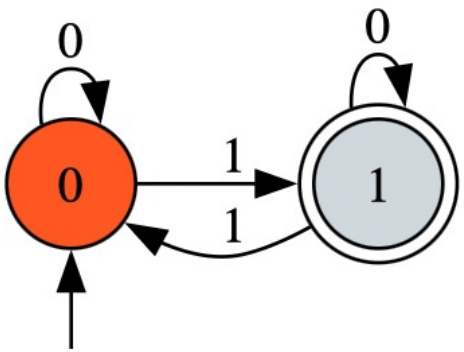
- Corret-by-Construction Program Transformation Algorithms [Oxford University, 2001-2006]
  - My doctoral thesis
  - Calculational proofs (on paper) via assertion propagation
  - Based on Dijkstra and Scholten's Predicate Calculus and Program Semantics

- Teaching with Dafny
  - Assertive Programming (some Ada/SPARK too)
    - Ben-Gurion University of the Negev (BGU) [2013-2024]
    - AFEKA Academic College of Engineering in Tel-Aviv [2016-2021]
    - Academic College of Tel Aviv – Yaffo (MTA) [2013-2015]
  - Logic and Set Theory
    - Ramat Gan Academic College (RGAC) [2021-2024]
  - **This talk: Automata Theory and Formal Languages**
    - RGAC [Fall 2021, Spring 2023]
    - AFEKA [Summer 2020]

- Formal Verification of Firmware and Software [NVIDIA, since Dec. 2020]
  - Model Checking and Deductive Verification
  - Sequential and Concurrent code
  - Languages: C , C++, Ada/SPARK

From Math to Verifiable Executable Code

- Strictly follow the Textbook by Elaine Rich
- Formulated substantial portions in Dafny
- This talk's focus: DFSMs
  - Deterministic Finite State Machines



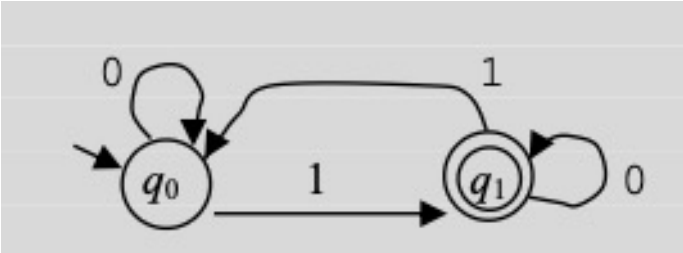
DETERMINISTIC FINITE STATE MACHINES (DFSMS): DEEP DIVE

Traditional Definition (Textbook)

Formally, a ***deterministic FSM*** (or ***DFSM***) M is a quintuple $(K, \Sigma, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- δ is the transition function. It maps from:

$$\underset{\text{state}}{K} \times \underset{\text{input symbol}}{\Sigma} \text{ to } \underset{\text{state}}{K}.$$

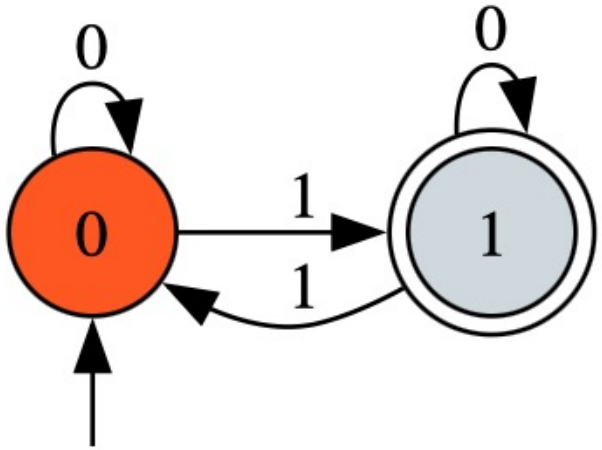


An Executable Definition in Dafny

```
264  module DFSM { // from lecture 3
268      type State = nat
269
270      type TransitionFunction = (State, Symbol) --> State
271      type DFSM = (set<State>, set<Symbol>, TransitionFunction, State, set<State>)
272
273      function K(M: DFSM): set<State> { M.0 }
274      function Sigma(M: DFSM): set<Symbol> { M.1 }
275      function Delta(M: DFSM): TransitionFunction { M.2 }
276      function s(M: DFSM): State { M.3 }
277      function A(M: DFSM): set<State> { M.4 }
```

Example DFSM: Checking for Odd Parity

- **Language:** Binary strings with **odd** number of 1s
- **Machine:**



- **DEMO: DFSM Visualizer on “10011”**

FSM Visualizer (Dafny-generated)

Machine type

DFSM

Input string

10011

Run / Reset

◄ Prev

Next ►

Step

0

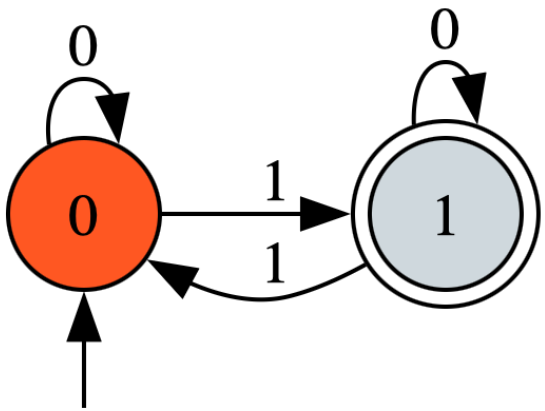
Step 0 of 5

Configuration 0:

< | 10011>

Zoom −

Zoom +



FSM Visualizer (Dafny-generated)

Machine type

DFSM

Input string

10011

Run / Reset

◄ Prev

Next ►

Step

1

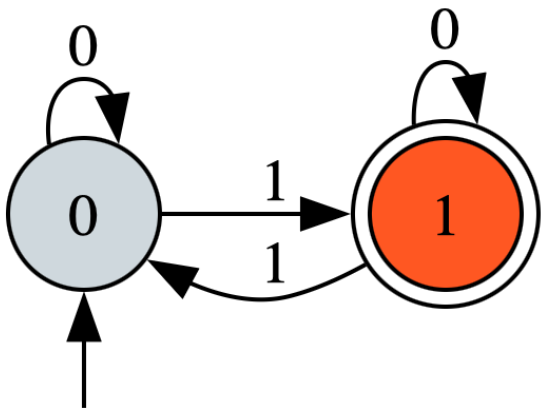
Step 1 of 5

Configuration 1:

```
<1 | 0011>
```

Zoom −

Zoom +



FSM Visualizer (Dafny-generated)

Machine type

DFSM

Input string

10011

Run / Reset

◄ Prev

Next ►

Step

2

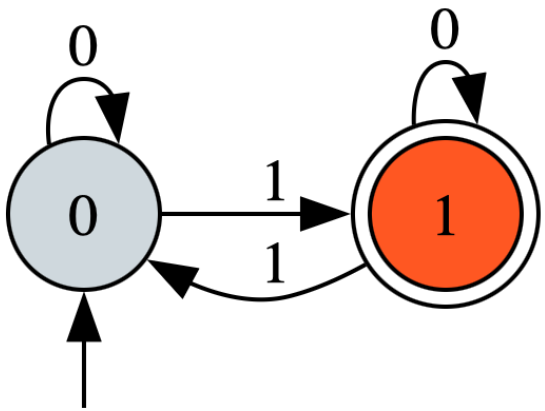
Step 2 of 5

Configuration 2:

<10 | 011>

Zoom −

Zoom +



FSM Visualizer (Dafny-generated)

Machine type

DFSM

Input string

10011

Run / Reset

◄ Prev

Next ►

Step

3

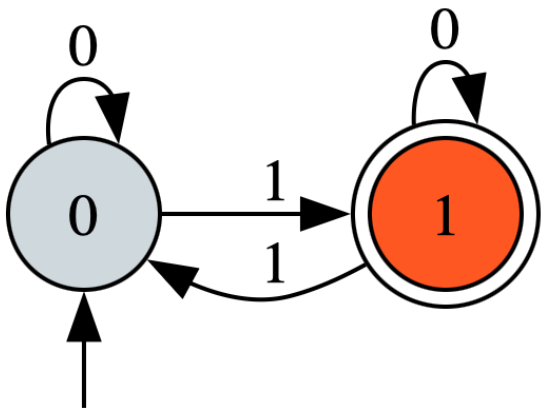
Step 3 of 5

Configuration 3:

<100 | 11>

Zoom −

Zoom +



FSM Visualizer (Dafny-generated)

Machine type

DFSM

Input string

10011

Run / Reset

◄ Prev

Next ►

Step

4

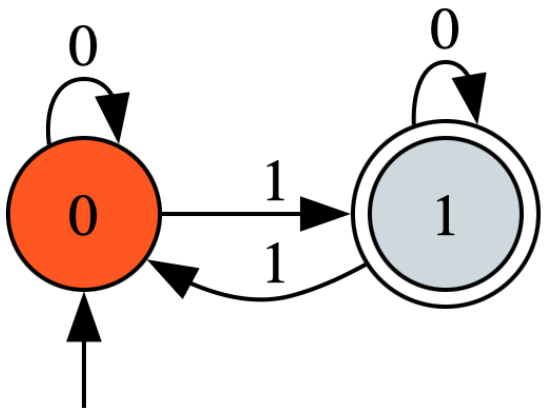
Step 4 of 5

Configuration 4:

<1001 | 1>

Zoom −

Zoom +



FSM Visualizer (Dafny-generated)

Machine type

DFSM

Input string

10011

Run / Reset

◄ Prev

Next ►

Step

5

Step 5 of 5

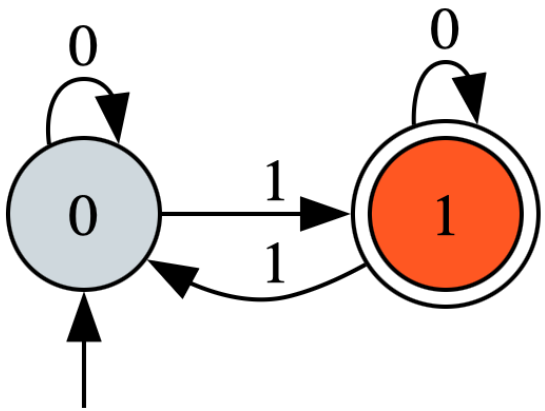
Configuration 5:

<10011 | >

Result: ✅ Accepted

Zoom −

Zoom +



Test on Selected Input Strings

```
120     method Main() {
121         // "running" a DFSM:
122         var accepted := dfsmsimulate(M_5_4, "1001");
123         accepted := dfsmsimulate(M_5_4, "10011");
```

```
rettinger@L2PDYF17FR automata-and-formal-languages % dafny run simulators/FSM_Simulators.dfy

Dafny program verifier finished with 11 verified, 0 errors
Run a DFSM on input string 1001
Configuration #0: <1001, 0>
Configuration #1: < 001, 1>
Configuration #2: <  01, 1>
Configuration #3: <   1, 1>
Configuration #4: <    , 0>
Result: Rejected
Run a DFSM on input string 10011
Configuration #0: <10011, 0>
Configuration #1: < 0011, 1>
Configuration #2: <  011, 1>
Configuration #3: <   11, 1>
Configuration #4: <    1, 0>
Configuration #5: <     , 1>
Result: Accepted
```

Example 5.4: Machine Definition

```
90      const q0: State := 0
91      const q1: State := 1
92      const K_5_4 := {q0, q1}
93      const Sigma_5_4 := {'0', '1'}
94      function delta_5_4(k: State, c: Symbol): State
95          requires k in K_5_4 && c in Sigma_5_4
96      {
97          if k == q0 && c == '0' then q0
98          else if k == q0 && c == '1' then q1
99          else if k == q1 && c == '0' then q1
100         else assert k == q1 && c == '1'; q0
101     }
102     const s_5_4 := q0
103     const A_5_4 := {q1}
104     const M_5_4 := (K_5_4, Sigma_5_4, delta_5_4, s_5_4, A_5_4)
```

Basic Definitions: Languages as Infinite Sets of Strings

```
1   module FormalLanguages { // from lectures 1 and 2
2       type Symbol = char
3       type String = seq<Symbol>
4       type Language = iset<String> // slide 34
5
6       // slide 23
7       const EPSILON: String := "" // == []
8
9       // slide 24
10      function Length(s: String): nat { |s| }
11
12      function Count(c: Symbol, s: String): nat
13          decreases |s|
14      {
15          if |s| == 0 then
16              0
17          else if s[0] == c then
18              1 + Count(c, s[1..])
19          else
20              Count(c, s[1..])
21      }
```

Language for Example 5.4

Mathematical Specification:

$L\_5\_4 = \{w \in \{0,1\}^* \mid w \text{ has odd number of 1s}\}$

Dafny Specification:

```
21  module Lecture03 {
84      // Example 5.4 in the textbook - a finite state machine that accepts all binary numbers having "odd parity"
85      ghost const L_5_4 := iset w | ValidString(w, Alphabet_0_1) && Count('1', w) % 2 == 1
```

Textbook Simulator

Here's a simple interpreter for a deterministic FSM $M = (K, \Sigma, \delta, s, A)$:

dfsmsimulate(M: DFSM, w: string) =
1. $st = s$.
2. Repeat:
   2.1. c = get-next-symbol(w).
   2.2. If $c \neq$ end-of-file then:
        $st = \delta(st, c)$.
   until c = end-of-file.
3. If $st \in A$ then accept else reject.

```
3   module FSM_SIMULATION {
9       method dfsmsimulate(M: DFSM, w: String) returns (accepted: bool)
10
11
12      {
13
14          var st := s(M);
15          var i := 0;
16
17          while i < |w|
18
19
20
21
22          {
23
24              var c := w[i];
25
26              st := Delta(M)(st, c);
27
28              i := i+1;
29          }
30
31          accepted := st in A(M);
32
33      }
```

```
3   module FSM_SIMULATION {
9       method dfsmsimulate(M: DFSM, w: String) returns (accepted: bool)
10
11
12      {
13          print "Run a DFSM on input string ", w;
14          var st := s(M);
15          var i := 0;
16          var blanks := ""; // just for pretty printing
17          while i < |w|
18
19
20
21
22          {
23              print "\nConfiguration #", i, ": <", blanks, w[i..], ", ", st, ">";
24              var c := w[i];
25
26              st := Delta(M)(st, c);
27              blanks := blanks + " ";
28              i := i+1;
29          }
30          print "\nConfiguration #", i, ": <", blanks, w[i..], ", ", st, ">";
31          accepted := st in A(M);
32          print "\nResult: ", if accepted then "Accepted" else "Rejected", "\n";
33      }
```

```
3   module FSM_SIMULATION {
9       method dfsmsimulate(M: DFSM, w: String) returns (accepted: bool)
10          requires ValidDFSM(M) && ValidString(w, Sigma(M))
11          ensures accepted <==> w in L(M)
12      {
13          print "Run a DFSM on input string ", w;
14          var st := s(M);
15          var i := 0;
16          var blanks := ""; // just for pretty printing
17          while i < |w|
18
19
20
21
22          {
23              print "\nConfiguration #", i, ": <", blanks, w[i..], ", ", st, ">";
24              var c := w[i];
25
26              st := Delta(M)(st, c);
27              blanks := blanks + " ";
28              i := i+1;
29          }
30          print "\nConfiguration #", i, ": <", blanks, w[i..], ", ", st, ">";
31          accepted := st in A(M);
32          print "\nResult: ", if accepted then "Accepted" else "Rejected", "\n";
33      }
```

For the Precondition

```
1    module FormalLanguages { // from lectures 1 and 2
133      ghost predicate ValidString(w: String, sigma: set<Symbol>) {
134          forall c :: c in w ==> c in sigma
135      }
```

```
264  module DFSM { // from lecture 3
280      ghost predicate ValidDFSM(M: DFSM) {
281          s(M) in K(M) && A(M) <= K(M) && forall k, c :: k in K(M) &&
282
             c in Sigma(M) ==> Delta(M).requires(k, c) && Delta(M)(k, c) in K(M)
         }
```

For the Postcondition

```
264  module DFSM { // from lecture 3
284      ghost function L(M: DFSM): Language {
285          iset w: String | ValidString(w, Sigma(M)) && AcceptedByDFSM(w, M)
286      }
287
288      ghost predicate AcceptedByDFSM(w: String, M: DFSM) {
289          ValidDFSM(M) && ValidString(w, Sigma(M)) && Yield_Full(w, M) in A(M)
290      }
291
292      ghost function Yield_Full(w: String, M: DFSM): (res: State)
293          requires ValidDFSM(M) && ValidString(w, Sigma(M))
294          ensures res in K(M)
295      {
296          Yield_Full_From(w, s(M), M)
297      }
```

For the Postcondition (cont.)

```
264   module DFSM { // from lecture 3
299       ghost function Yield_Full_From(w: String, state: State, M: DFSM): (res: State)
300           requires ValidDFSM(M) && state in K(M) && ValidString(w, Sigma(M))
301           ensures res in K(M)
302       {
303           if |w| == 0 then state
304           else
305               var c := w[0];
306               var next_w := w[1..];
307               assert Delta(M).requires(state, c);
308               var next_state := Delta(M)(state, c);
309               Yield_Full_From(next_w, next_state, M)
310       }
311
312       // Slide 12: regular language definition
313       type RegularLanguage = lang: Language | exists M: DFSM :: L(M) == lang witness *
314   }
```

```
3   module FSM_SIMULATION {
9       method dfsmsimulate(M: DFSM, w: String) returns (accepted: bool)
10          requires ValidDFSM(M) && ValidString(w, Sigma(M))
11          ensures accepted <==> w in L(M)
12      {
13          print "Run a DFSM on input string ", w;
14          var st := s(M);
15          var i := 0;
16          var blanks := ""; // just for pretty printing
17          while i < |w|
18              invariant 0 <= i <= |w| && st in K(M)
19              invariant ValidDFSM(M) && ValidString(w, Sigma(M))
20              invariant Yield_Full_From(w[i..], st, M) in A(M) <==> w in L(M)
21              decreases |w|-i
22          {
23              print "\nConfiguration #", i, ": <", blanks, w[i..], ", ", st, ">";
24              var c := w[i];
25              assert st in K(M) && c in Sigma(M);
26              st := Delta(M)(st, c);
27              blanks := blanks + " ";
28              i := i+1;
29          }
30          print "\nConfiguration #", i, ": <", blanks, w[i..], ", ", st, ">";
31          accepted := st in A(M);
32          print "\nResult: ", if accepted then "Accepted" else "Rejected", "\n";
33      }
```

The Machine Correctness Question

We have:
- Machine M_5_4 (implementation)
- Language L_5_4 (specification)

Question: Does M_5_4 accept exactly L_5_4?

In Dafny:

```
 21  module Lecture03 {
102      lemma Lemma_5_4()
103          ensures L(M_5_4) == L_5_4
```

This is what we'll prove! (formally, auto-actively)

Base Case: An Empty String

```
21  module Lecture03 {
102     lemma Lemma_5_4()
103         ensures L(M_5_4) == L_5_4
104     {
105         forall w | ValidString(w, Alphabet_0_1) ensures w in L_5_4 <==> w in L(M_5_4) {
106             if |w| == 0 {
107                 assert w !in L_5_4 by { assert Count('1', w) == 0; }
108                 assert w !in L(M_5_4) by { assert Yield_Full_From(w, q0, M_5_4) == q0 && q0 !in A(M_5_4);}
109             }
110 >           else {…
131             }
132         }
133     }
```

General Case: A Non-Empty String

```
111             assert |w| > 0;
112             var c := w[0];
113             var next_w := w[1..];
114             assert ValidString(next_w, Sigma(M_5_4));
115             var next_state := Delta(M_5_4)(q0, c);
116             calc {
117                 w in L(M_5_4);
118             ==
119                 ValidString(w, Sigma(M_5_4)) && AcceptedByDFSM(w, M_5_4);
120             ==
121                 Yield_Full(w, M_5_4) in A(M_5_4);
122             ==
123                 Yield_Full_From(w, s(M_5_4), M_5_4) in A(M_5_4);
124             ==
125                 Yield_Full_From(w, q0, M_5_4) in A(M_5_4);
126             == { Lemma_5_4_q0(w); }
127                 Count('1', w) % 2 == 1;
128             ==
129                 w in L_5_4;
130             }
```

Lemma for q_0: An Empty String

```
21  module Lecture03 {
135     lemma Lemma_5_4_q0(w: String)
136         requires ValidString(w, Alphabet_0_1)
137         ensures Count('1', w) % 2 == 1 <==> Yield_Full_From(w, q0, M_5_4) in A(M_5_4)
138         decreases |w|
139     {
140         if |w| == 0 {
141             assert Count('1', w) % 2 == 0;
142             assert Yield_Full_From(w, q0, M_5_4) == q0 && q0 !in A(M_5_4);
143         }
144 >       else {…
173         }
174     }
```

Lemma for state q_0: A Non-Empty String on '0'

```
145         assert ValidString(w, Alphabet_0_1);
146         assert |w| > 0;
147         var c := w[0];
148         var next_w := w[1..];
149         assert ValidString(next_w, Alphabet_0_1);
150         var next_state := Delta(M_5_4)(q0, c);
151         if w[0] == '0' {
152             calc {
153                 Yield_Full_From(w, q0, M_5_4) in A(M_5_4);
154                 ==
155                 Yield_Full_From(next_w, next_state, M_5_4) in A(M_5_4);
156                 == { assert next_state == delta_5_4(q0, '0') == q0; Lemma_5_4_q0(next_w); }
157                 Count('1', next_w) % 2 == 1;
158                 == { assert w == ['0'] + next_w; }
159                 Count('1', w) % 2 == 1;
160             }
161         }
162 >       else {…
172         }
```

Lemma for state q_0: A Non-Empty String on '1'

```
163                 calc {
164                     Yield_Full_From(w, q0, M_5_4) in A(M_5_4);
165                 ==
166                     Yield_Full_From(next_w, next_state, M_5_4) in A(M_5_4);
167                 == { assert next_state == delta_5_4(q0, '1') == q1; Lemma_5_4_q1(next_w); }
168                     Count('1', next_w) % 2 == 0;
169                 == { assert w == ['1'] + next_w; }
170                     Count('1', w) % 2 == 1;
171                 }
```

Proof Structure for the Correctness of a DFSM

- By induction on the size of the input string
- Lemma for each state
  - Mutually recursive
  - Proof follows the machine's transition function
- Main proof "calls" the lemma of the machine's start state

Deterministic FSM Summary

- Semantics of a DFSM is formally defined
- DFSM interpreter fully proved
- Students can test and prove correctness of their DFSMs

Result: Students see theory come alive!

COURSE SCOPE & ECOSYSTEM

What's Included

- **Fully Formalized:**
  - Strings, languages, basic operations
  - DFSMs with complete correctness proofs (M_5_4!)
  - NDFSMs with epsilon-closure
- **Partially Verified:**
  - Pumping Lemma for Regular Languages (some helper lemmas unverified)
  - Regular expressions and Kleene's theorem
  - CFGs with derivation generators
  - PDAs with nondeterministic simulators
  - Turing machines with operational semantics
  - Advanced algorithms (construction proofs incomplete)
- **Tools:**
  - Simulators for all automaton types (in Dafny)
  - Web visualizers (Dafny → Python → Streamlit)
  - Potential: Automated testing for assignment submissions evaluation

What's NOT Covered [with Dafny]

- **Not Included:**
  - Parsers and parsing algorithms
  - Decidability formalization
  - Complete verification of all algorithms
  - Many advanced proofs
- **Honest Assessment:**
  - Full verification requires expertise beyond 2nd-year undergrads
  - Some proofs serve as examples, not requirements
  - Focus on learning through doing, proofs as bonus

CONCLUSION

Key Takeaways

Dafny transforms automata theory education:

- Abstract → Concrete
  - Executable (imperative+declarative+functional) code
  - Immediate feedback
- Complete Auto-Active Verification Possible
  - FSMs (like the M_5_4 example) fully proven
  - No logical gaps, checked by the machine
- Students Learn by Doing
  - Implement, test, debug, understand
  - Optional proofs deepen understanding

Thanks!

- For more info please reach out to me:

ran.ettinger@gmail.com

ranger@post.bgu.ac.il

rettinger@nvidia.com

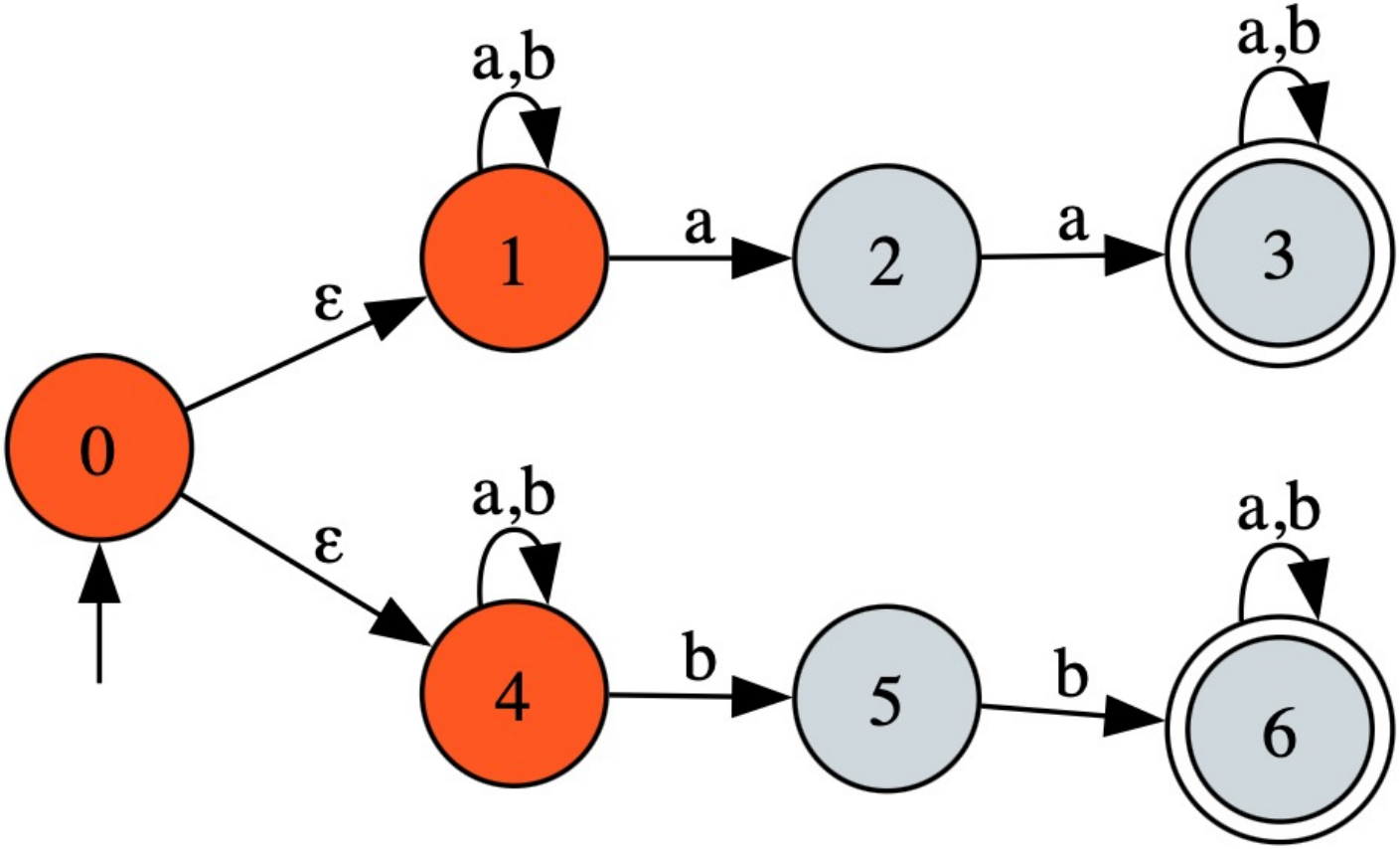
GitHub: https://github.com/ranger71

BACKUP SLIDES

Backup: Assignment Examples

- Assignment 1 (Chapter 5): DFSMs
  - Binary strings where every ‘a’ has ‘b’ before and after
  - Strings with neither “ab” nor “bb” substring
  - Binary numbers divisible by 3
- Assignment 1 (Chapter 6): NDFSMs
  - Strings containing “aa” or “bb”
  - Third-to-last symbol is ‘1’
  - Manually trace epsilon-closure
- Assignment 2 (Chapter 11): CFGs
  - Balanced parentheses (multiple types)
  - Palindromes - {a^i b^j | 2i = 3j + 1}
- Assignment 2 (Chapter 12): PDAs
  - Same languages as CFG assignment
  - Test with simulator

Example NDFSM: Strings containing “aa” or “bb”



```
3   module FSM_SIMULATION {
106     const K_59: set<State> := {0, 1, 2, 3, 4, 5, 6}
107     const sigma_59: set<Symbol> := {'a', 'b'}
108     const DELTA_59: TransitionRelation :=
109         {(0, epsilon, 1), (0, epsilon, 4),
110          (1, sym('a'), 1), (1, sym('b'), 1), (1, sym('a'), 2),
111          (2, sym('a'), 3),
112          (3, sym('a'), 3), (3, sym('b'), 3),
113          (4, sym('a'), 4), (4, sym('b'), 4), (4, sym('b'), 5),
114          (5, sym('b'), 6),
115          (6, sym('a'), 6), (6, sym('b'), 6)}
116     const s_59: State := 0
117     const A_59: set<State> := {3, 6}
118     const M_59: NDFSM := (K_59, sigma_59, DELTA_59, s_59, A_59)
```

FSM Visualizer (Dafny-generated)

Machine type

NDFSM

Input string

abba

Run / Reset

◄ Prev

Next ►

Step

0

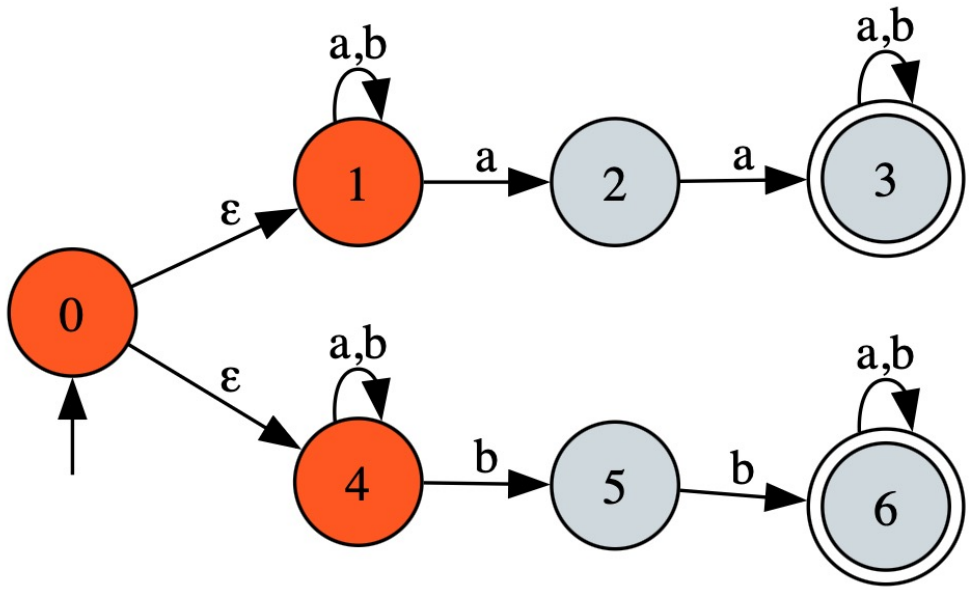
Step 0 of 4

Configuration 0:

< | abba>

Zoom −

Zoom +



FSM Visualizer (Dafny-generated)

Machine type

NDFSM

Input string

abba

Run / Reset

◄ Prev | Next ►

Step

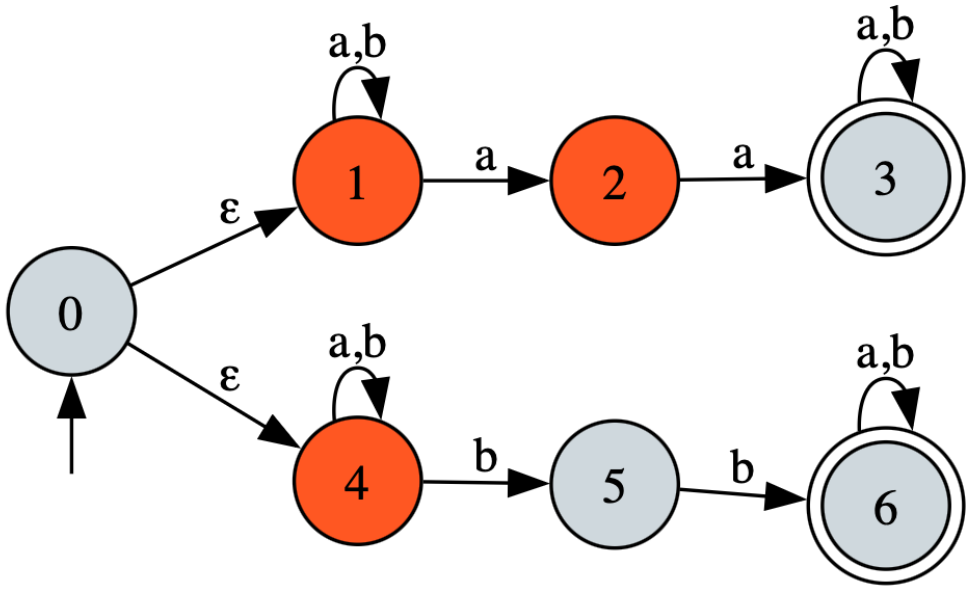
1

Step 1 of 4

Configuration 1:

<a | bba>

Zoom − | Zoom +



FSM Visualizer (Dafny-generated)

Machine type

NDFSM

Input string

abba

Run / Reset

◄ Prev

Next ►

Step

2

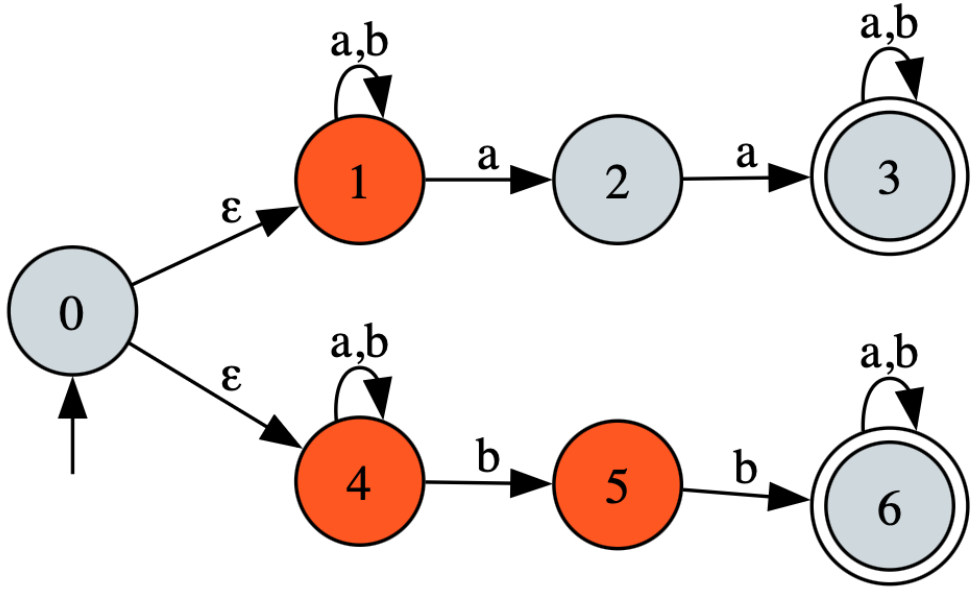
Step 2 of 4

Configuration 2:

<ab | ba>

Zoom −

Zoom +



FSM Visualizer (Dafny-generated)

Machine type

NDFSM

Input string

abba

Run / Reset

◄ Prev

Next ►

Step

3

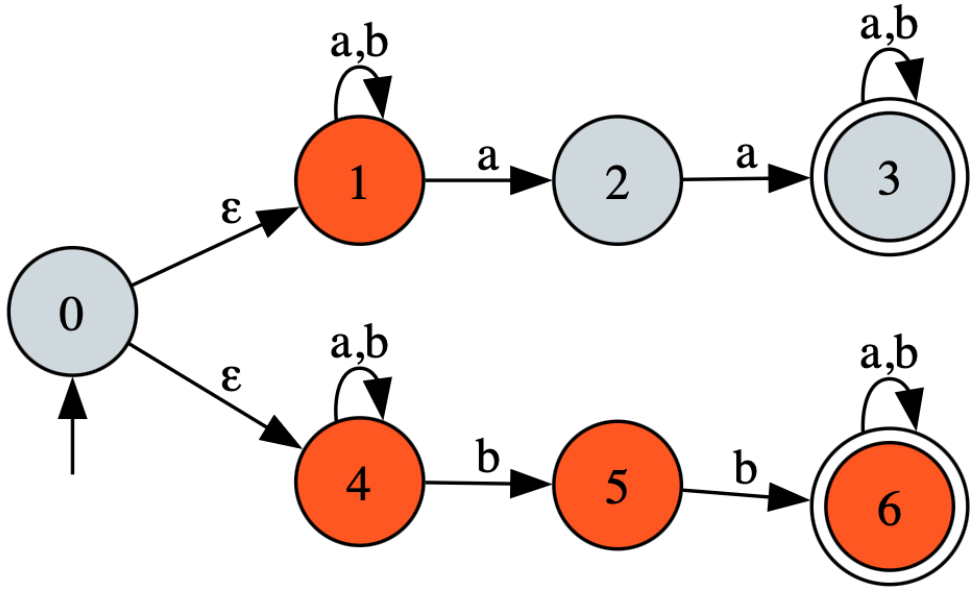
Step 3 of 4

Configuration 3:

<abb | a>

Zoom −

Zoom +



FSM Visualizer (Dafny-generated)

Machine type

NDFSM

Input string

abba

Run / Reset

◄ Prev

Next ►

Step

4

Step 4 of 4

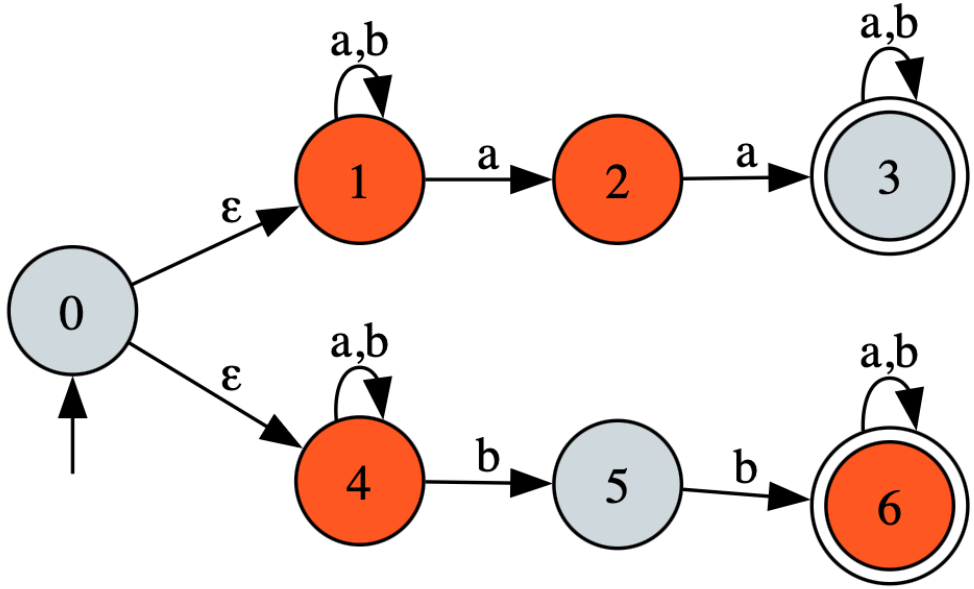
Configuration 4:

<abba | >

Result: ✅ Accepted

Zoom −

Zoom +



```
Run an NDFSM on input string aba
<aba, {0, 1, 4}>
< ba, {1, 2, 4}>
<  a, {1, 4, 5}>
<   , {1, 2, 4}>
Result: Rejected
```

```
Run an NDFSM on input string baa
<baa, {0, 1, 4}>
< aa, {1, 4, 5}>
<  a, {1, 2, 4}>
<   , {1, 2, 3, 4}>
Result: Accepted
```

```
Run an NDFSM on input string abba
<abba, {0, 1, 4}>
< bba, {1, 2, 4}>
<  ba, {1, 4, 5}>
<   a, {1, 4, 5, 6}>
<    , {1, 2, 4, 6}>
Result: Accepted
```

```
3   module PDA_Simulation {
6       type State = nat
7       datatype MaybeSymbol = epsilon | sym(Symbol)
8       // ((from state, current input string symbol or epsilon, top of the stack string to pop), (to s
9       type PDA_Transition = ((State, MaybeSymbol, String), (State, String))
10      type PDA_TransitionRelation = seq<PDA_Transition>
11      type PDA = (seq<State>, seq<Symbol>, seq<Symbol>, PDA_TransitionRelation, State, seq<State>)
12      type PDA_Configuration = (State, String, String) // state, remaining suffix of input string, co
13
14      function PDA_K(M: PDA): seq<State> { M.0 }
15      function PDA_Sigma(M: PDA): seq<Symbol> { M.1 }
16      function PDA_Gamma(M: PDA): seq<Symbol> { M.2 }
17      function PDA_Delta(M: PDA): PDA_TransitionRelation { M.3 }
18      function PDA_s(M: PDA): State { M.4 }
19      function PDA_A(M: PDA): seq<State> { M.5 }
```

```
3   module PDA_Simulation {
40      // example PDA to accept the (A^n)(B^n) language
41      const K_AnBn := [1, 2]
42      const Sigma_AnBn := ['a', 'b']
43      const Gamma_AnBn := ['a']
44      const DELTA_AnBn: PDA_TransitionRelation :=
45          [((1, sym('a'), EPSILON), (1, "a")),
46           ((1, sym('b'), "a"), (2, EPSILON)),
47           ((2, sym('b'), "a"), (2, EPSILON))]
48      const s_AnBn: State := 1
49      const A_AnBn: seq<State> := [1, 2]
50      const M_AnBn := (K_AnBn, Sigma_AnBn, Gamma_AnBn, DELTA_AnBn, s_AnBn, A_AnBn)
51      const DELTA_AnBn': PDA_TransitionRelation :=
52          [((1, sym('a'), EPSILON), (1, "a")),
53           ((1, epsilon, EPSILON), (2, EPSILON)),
54           ((2, sym('b'), "a"), (2, EPSILON))]
55      const A_AnBn': seq<State> := [2]
56      const M_AnBn' := (K_AnBn, Sigma_AnBn, Gamma_AnBn, DELTA_AnBn', s_AnBn, A_AnBn')
```

```
3   module PDA_Simulation {
169     method {:verify false} Main'()
170         decreases *
171     {
172         assert ValidPDA(M_AnBn) && ValidString("", set c | c in PDA_Sigma(M_AnBn));
173         var accepted := pdasimulate(M_AnBn, "");
174         assert accepted;
175         assert ValidPDA(M_AnBn) && ValidString("aaab", set c | c in PDA_Sigma(M_AnBn));
176         accepted := pdasimulate(M_AnBn, "aaab");
177         assert !accepted;
178         assert ValidPDA(M_AnBn) && ValidString("aaabbb", set c | c in PDA_Sigma(M_AnBn));
179         accepted := pdasimulate(M_AnBn, "aaabbb");
180         assert accepted;
181         // do the same on a nondeterministic version of a PDA accepting the same language
182         assert ValidPDA(M_AnBn') && ValidString("", set c | c in PDA_Sigma(M_AnBn));
183         accepted := pdasimulate(M_AnBn', "");
184         assert accepted;
185         assert ValidPDA(M_AnBn') && ValidString("aaab", set c | c in PDA_Sigma(M_AnBn'));
186         accepted := pdasimulate(M_AnBn', "aaab");
187         assert !accepted;
188         assert ValidPDA(M_AnBn') && ValidString("aaabbb", set c | c in PDA_Sigma(M_AnBn'));
189         accepted := pdasimulate(M_AnBn', "aaabbb");
190         assert accepted;
191     }
192 }
```

```
Run the PDA
(K = [1, 2], Sigma = ab, Gamma = a, DELTA = [((1, PDA_Simulation.MaybeSymbol.sym('a'), []),
ulation.MaybeSymbol.sym('b'), ['a']), (2, []))], s = 1, A = [2])
on input string "aaab"
Configuration #0: <(state=1, suffix="aaab", stack="")>
Found 2 next configurations. Trying each of them in order.
Configuration #1: <(state=1, suffix="aab", stack="a")>
Found 2 next configurations. Trying each of them in order.
Configuration #2: <(state=1, suffix="ab", stack="aa")>
Found 2 next configurations. Trying each of them in order.
Configuration #3: <(state=1, suffix="b", stack="aaa")>
Configuration #4: <(state=2, suffix="b", stack="aaa")>
Configuration #5: <(state=2, suffix="", stack="aa")>
At the end of the input string the stack is not empty. Backtracking.
Configuration #3: <(state=2, suffix="ab", stack="aa")>
Stuck with no potential transitions. Backtracking.
Configuration #2: <(state=2, suffix="aab", stack="a")>
Stuck with no potential transitions. Backtracking.
Configuration #1: <(state=2, suffix="aaab", stack="")>
Stuck with no potential transitions. Backtracking.
The input string "aaab" is NOT in the language accepted by this PDA.
```

```
Run the PDA
(K = [1, 2], Sigma = ab, Gamma = a, DELTA = [((1, PDA_Simulation.MaybeSymbol.sym('a'), []),
ulation.MaybeSymbol.sym('b'), ['a']), (2, []))], s = 1, A = [2])
on input string "aaabbb"
Configuration #0: <(state=1, suffix="aaabbb", stack="")>
Found 2 next configurations. Trying each of them in order.
Configuration #1: <(state=1, suffix="aabbb", stack="a")>
Found 2 next configurations. Trying each of them in order.
Configuration #2: <(state=1, suffix="abbb", stack="aa")>
Found 2 next configurations. Trying each of them in order.
Configuration #3: <(state=1, suffix="bbb", stack="aaa")>
Configuration #4: <(state=2, suffix="bbb", stack="aaa")>
Configuration #5: <(state=2, suffix="bb", stack="aa")>
Configuration #6: <(state=2, suffix="b", stack="a")>
Configuration #7: <(state=2, suffix="", stack="")>
This configuration is accepting!
The input string "aaabbb" is in the language accepted by this PDA.
```

PDA Visualizer (Dafny-generated)

Machine: M__AnBn

Zoom −

Zoom +

Input string

aaabbb

Run / Reset

◄ Prev

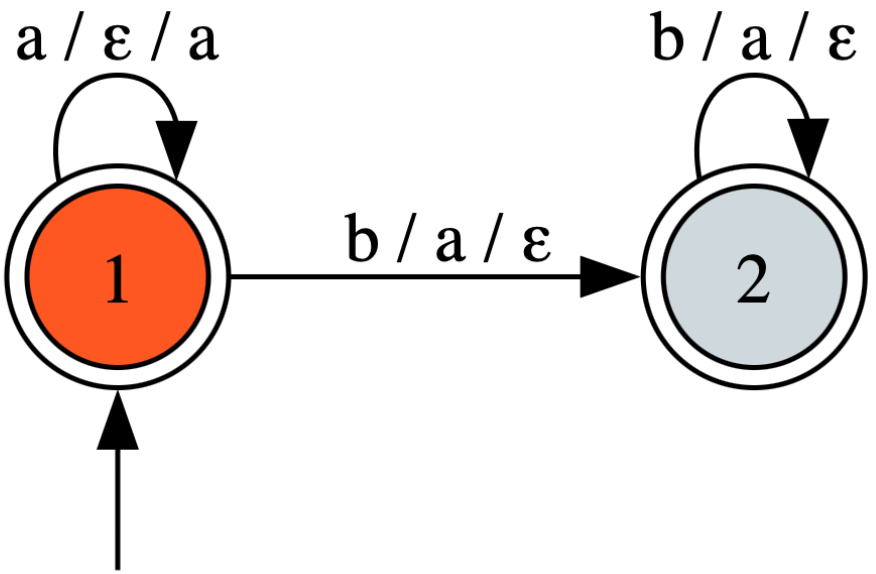
Next ►

Step

0

Step 0 of 6

Configuration 0: (state=1, suffix="aaabbb", stack="")



PDA Visualizer (Dafny-generated)

Machine: M__AnBn

Zoom −

Zoom +

Input string

aaabbb

Run / Reset

◄ Prev

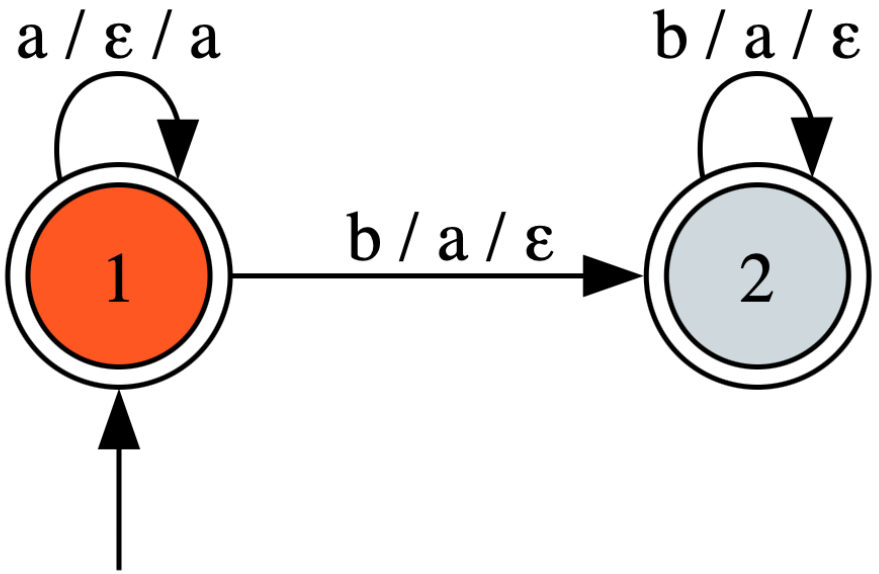
Next ►

Step

1

Step 1 of 6

Configuration 1: (state=1, suffix="aabbb", stack="a")



PDA Visualizer (Dafny-generated)

Machine: M__AnBn

Zoom −

Zoom +

Input string

aaabbb

Run / Reset

◄ Prev

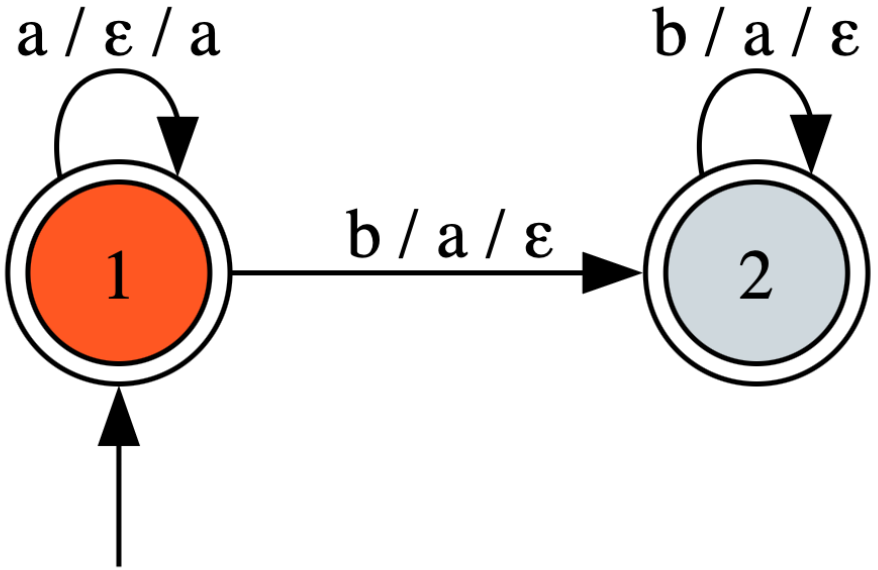
Next ►

Step

2

Step 2 of 6

Configuration 2: (state=1, suffix="abbb", stack="aa")



PDA Visualizer (Dafny-generated)

Machine: M__AnBn

Zoom −

Zoom +

Input string

aaabbb

Run / Reset

◄ Prev

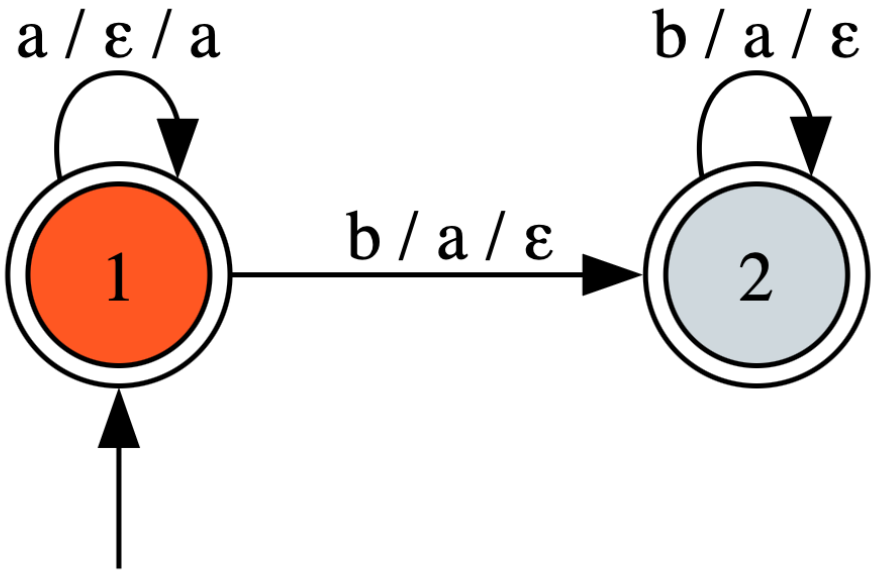
Next ►

Step

3

Step 3 of 6

Configuration 3: (state=1, suffix="bbb", stack="aaa")



PDA Visualizer (Dafny-generated)

Machine: M__AnBn

Zoom −

Zoom +

Input string

aaabbb

Run / Reset

◄ Prev

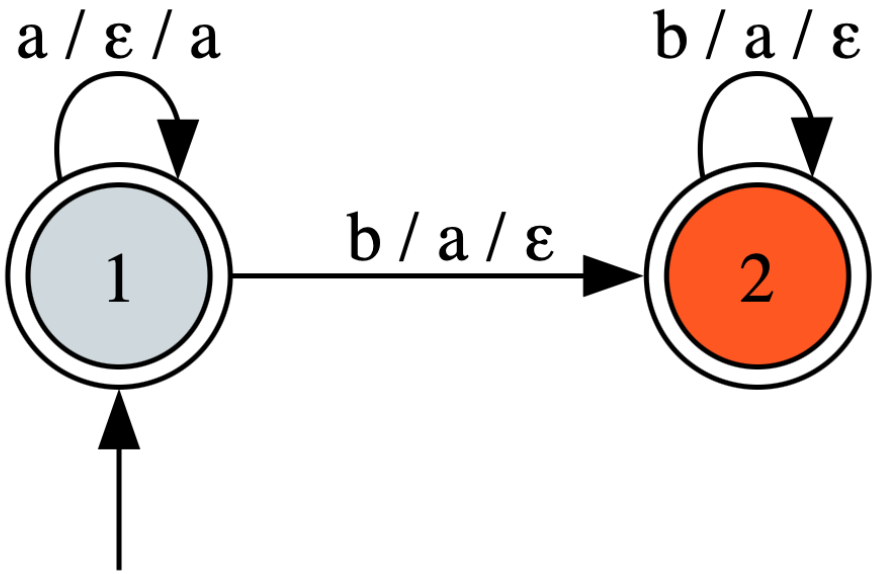
Next ►

Step

4

Step 4 of 6

Configuration 4: (state=2, suffix="bb", stack="aa")



PDA Visualizer (Dafny-generated)

Machine: M__AnBn

Zoom −

Zoom +

Input string

aaabbb

Run / Reset

◄ Prev

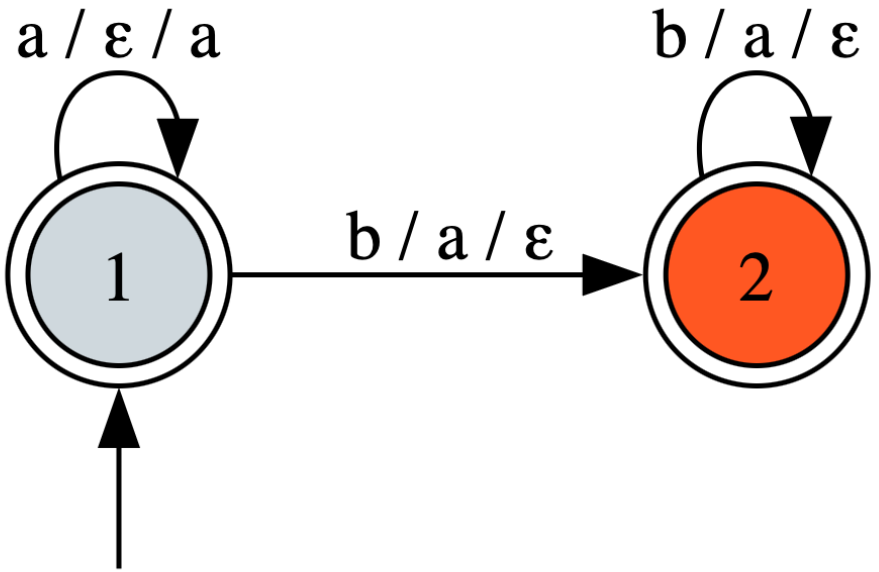
Next ►

Step

5

Step 5 of 6

Configuration 5: (state=2, suffix="b", stack="a")



PDA Visualizer (Dafny-generated)

Machine: M__AnBn

Zoom −

Zoom +

Input string

aaabbb

Run / Reset

◄ Prev

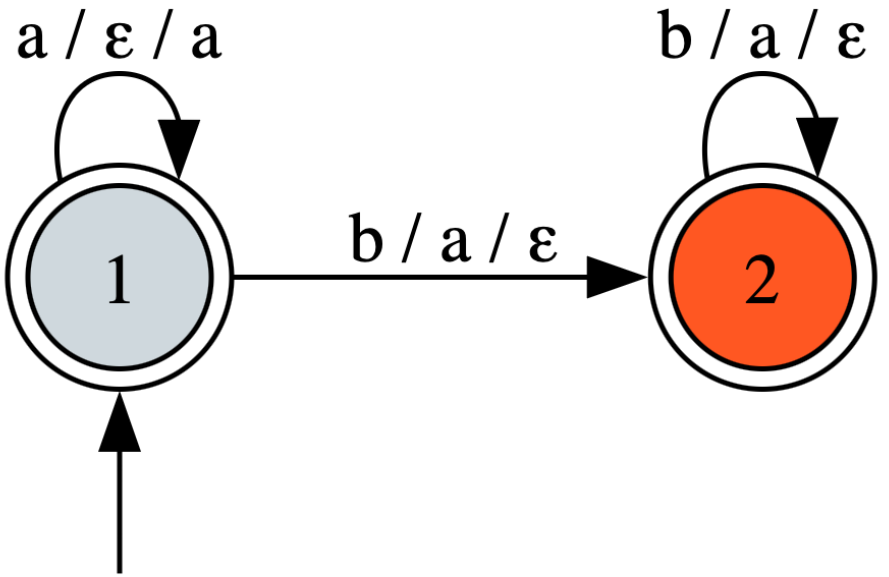
Next ►

Step

6

Step 6 of 6

Configuration 6: (state=2, suffix="", stack="")



Result: ✅ Accepted

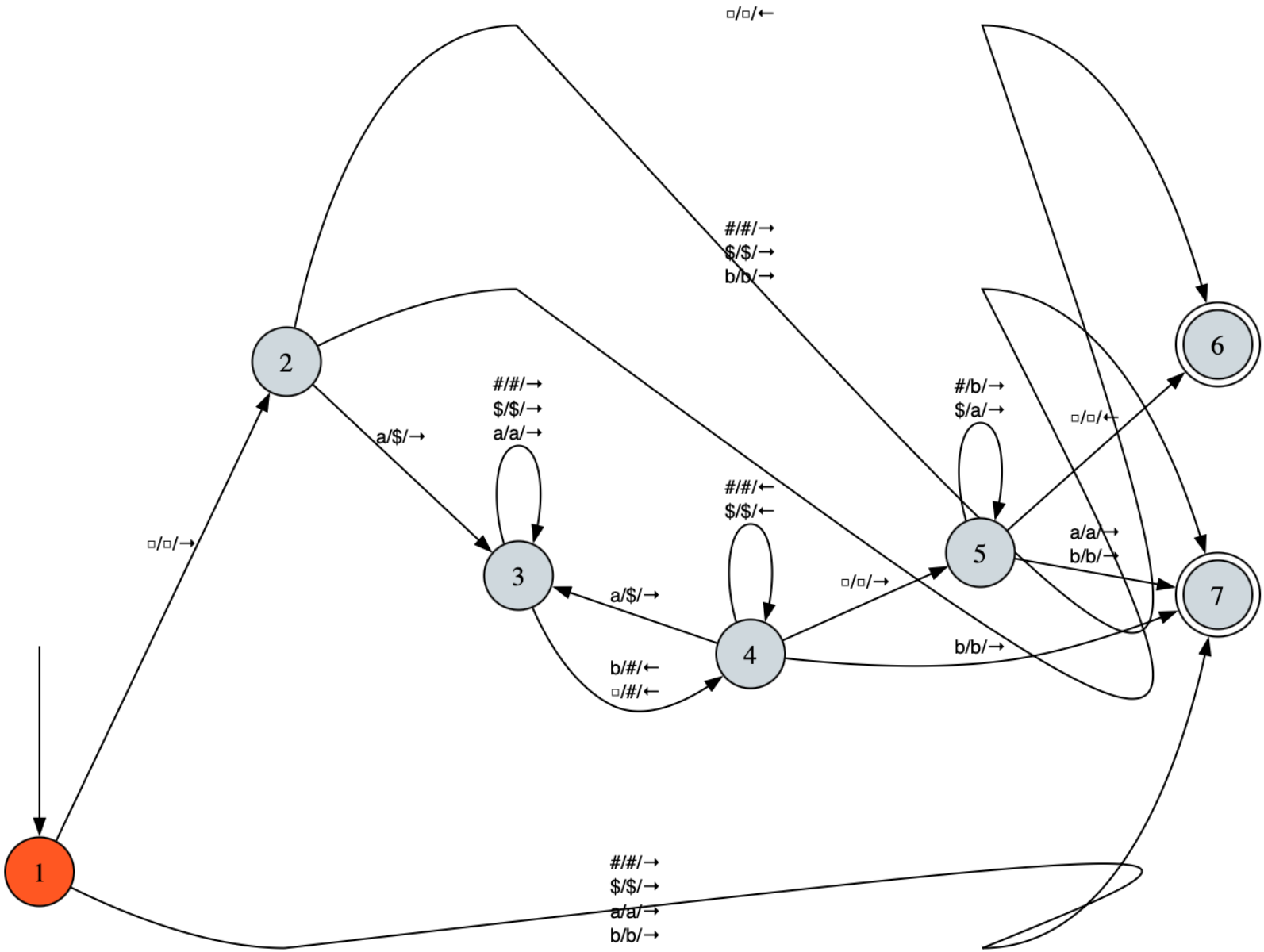
```
3   module Turing_Machine_Simulation {
6       type State = nat
7       datatype Direction = Left | Right
8       type TM_TransitionFunction = (State, Symbol) --> (State, Symbol, Direction)
9       type TuringMachine = (set<State>, set<Symbol>, set<Symbol>, TM_TransitionFunction, State, set<State>)
10      type TM_Configuration = (State, seq<Symbol>, Symbol, seq<Symbol>)
11
12      const BLANK_SQUARE: Symbol := ' '
13
14      function TM_K(M: TuringMachine): set<State> { M.0 }
15      function TM_Sigma(M: TuringMachine): set<Symbol> { M.1 }
16      function TM_Gamma(M: TuringMachine): set<Symbol> { M.2 }
17      function TM_Delta(M: TuringMachine): TM_TransitionFunction { M.3 }
18      function TM_s(M: TuringMachine): State { M.4 }
19      function TM_H(M: TuringMachine): set<State> { M.5 }
20
21      // TM properties:
22      ghost predicate ValidTM(M: TuringMachine) {
23      TM_s(M) in TM_K(M) && TM_H(M) <= TM_K(M) &&
24      TM_Sigma(M) <= TM_Gamma(M) && BLANK_SQUARE in TM_Gamma(M)-TM_Sigma(M) &&
25      forall k, c :: k in TM_K(M)-TM_H(M) && c in TM_Gamma(M) ==>
26          TM_Delta(M).requires(k, c) &&
27          TM_Delta(M)(k, c).0 in TM_K(M) &&
28          TM_Delta(M)(k, c).1 in TM_Gamma(M)
29      }
```

```
3   module Turing_Machine_Simulation {
31      predicate NoLeadingBlanks(str: String) {
32          |str| == 0 || str[0] != BLANK_SQUARE
33      }
34
35      predicate NoTrailingBlanks(str: String) {
36          |str| == 0 || str[|str|-1] != BLANK_SQUARE
37      }
38
39      ghost predicate Valid_TM_Configuration(C: TM_Configuration, M: TuringMachine) {
40          ValidTM(M) &&
41          C.0 in TM_K(M) &&
42          (forall c :: c in C.1 + [C.2] + C.3 ==> c in TM_Gamma(M)) &&
43          NoLeadingBlanks(C.1) &&
44          NoTrailingBlanks(C.3)
45      }
46
47      predicate Halting_TM_Configuration(C: TM_Configuration, M: TuringMachine)
48          requires Valid_TM_Configuration(C, M)
49      {
50          C.0 in TM_H(M)
51      }
```

```
3   module Turing_Machine_Simulation {
76      method {:verify false} tmsimulate(M: TuringMachine, w: String) returns (output: TM_Configuration, steps: nat)
77          requires ValidTM(M) && ValidString(w, TM_Sigma(M))
78          ensures Halting_TM_Configuration(output, M)
79          decreases * // could be non-terminating
80      {
81          print "Run a TM on input string ", w;
82          steps := 0;
83          var st := TM_s(M);
84          var tape_left, c, tape_right := "", BLANK_SQUARE, w;
85          var blanks := "";
86          while !Halting_TM_Configuration((st, tape_left, c, tape_right), M)
87              invariant Valid_TM_Configuration((st, tape_left, c, tape_right), M)
88              decreases *
89          {…
112         }
113         output := (st, tape_left, c, tape_right);
114         print "\n ", output.1 + [output.2] + output.3, "\t\tConfiguration #", steps, ": <", output, ">";
115         print "\nTM halts after ", steps, " steps.";
116         assert Valid_TM_Configuration(output, M);
117         assert Halting_TM_Configuration(output, M);
118     }
```

```
 3  module Turing_Machine_Simulation {
76      method {:verify false} tmsimulate(M: TuringMachine, w: String) returns (output: TM_Configuration, steps: nat)
86          while !Halting_TM_Configuration((st, tape_left, c, tape_right), M)
87              invariant Valid_TM_Configuration((st, tape_left, c, tape_right), M)
88              decreases *
89          {
90              print "\n ", tape_left + [c] + tape_right, "\t\tConfiguration #", steps, ": <", (st, tape_left, c, tape_right), ">";
91              assert st in TM_K(M) && c in TM_Gamma(M);
92              var action := TM_Delta(M)(st, c);
93              st := action.0;
94              var tape := tape_left + [action.1] + tape_right;
95              var c_index := if action.2 == Left then |tape_left| - 1 else |tape_left| + 1;
96              if c_index < 0 {
97                  tape_left, c, tape_right := "", BLANK_SQUARE, tape;
98              }
99              else if c_index >= |tape| {
100                 tape_left, c, tape_right := tape, BLANK_SQUARE, "";
101             }
102             else {
103                 tape_left, c, tape_right := tape[..c_index], tape[c_index], tape[c_index+1..];
104             }
105             if |tape_left| == 1 && tape_left  == [BLANK_SQUARE] {
106                 tape_left := [];
107             }
108             if |tape_right| == 1 && tape_right  == [BLANK_SQUARE] {
109                 tape_right := [];
110             }
111             steps := steps + 1;
112         }
```

```
3   module Turing_Machine_Simulation {
53      // example TM from slide 9:
54      const K_9 := {1, 2, 3, 4, 5, 6, 7} // adding 7 as a dummy halting state: it should never be reached when running on valid input
55      const Sigma_9 := {'a', 'b'}
56      const Gamma_9 := Sigma_9 + {BLANK_SQUARE, '$', '#'}
57      function delta_9(k: State, c: Symbol): (State, Symbol, Direction)
58      requires k in K_9-H_9 && c in Gamma_9
59      {
60      if k == 1 then
61          if c == BLANK_SQUARE then (2, c, Right) else (7, c, Right)
62      else if k == 2 then
63          if c == 'a' then (3, '$', Right) else if c == BLANK_SQUARE then (6, c, Left) else (7, c, Right)
64      else if k == 3 then
65          if c == 'a' || c == '$' || c == '#' then (3, c, Right) else if c == 'b' || c == BLANK_SQUARE then (4, '#', Left) else (7, c, Right)
66      else if k == 4 then
67          if c == 'a' then (3, '$', Right) else if c == '$' || c == '#' then (4, c, Left) else if c == BLANK_SQUARE then (5, c, Right) else (7, c, Right)
68      else
69          assert k == 5;
70          if c == '$' then (5, 'a', Right) else if c == '#' then (5, 'b', Right) else if c == BLANK_SQUARE then (6, c, Left) else (7, c, Right)
71      }
72      const s_9 := 1
73      const H_9 := {6, 7}
74      const M_9 := (K_9, Sigma_9, Gamma_9, delta_9, s_9, H_9)
```


1
2
3
4
5
6
7
▫/▫/→
a/$/→
#/#/→
$/$/→
a/a/→
b/#/←
▫/#/←
#/#/←
$/$/←
▫/▫/→
#/b/→
$/a/→
▫/▫/←
a/a/→
b/b/→
b/b/→
▫/▫/←
#/#/→
$/$/→
b/b/→
#/#/→
$/$/→
a/a/→
b/b/→

Machine: M__9

Zoom −

Zoom +

Zoom: 60%

Input string

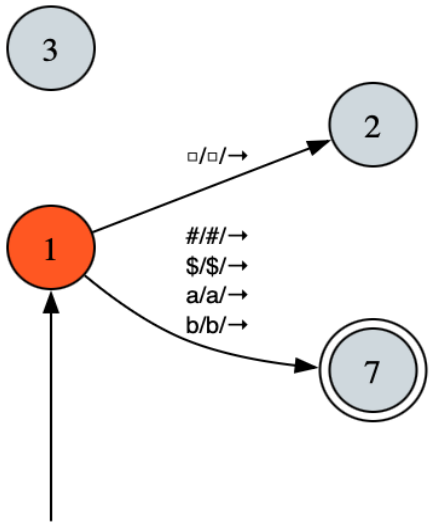
aaab

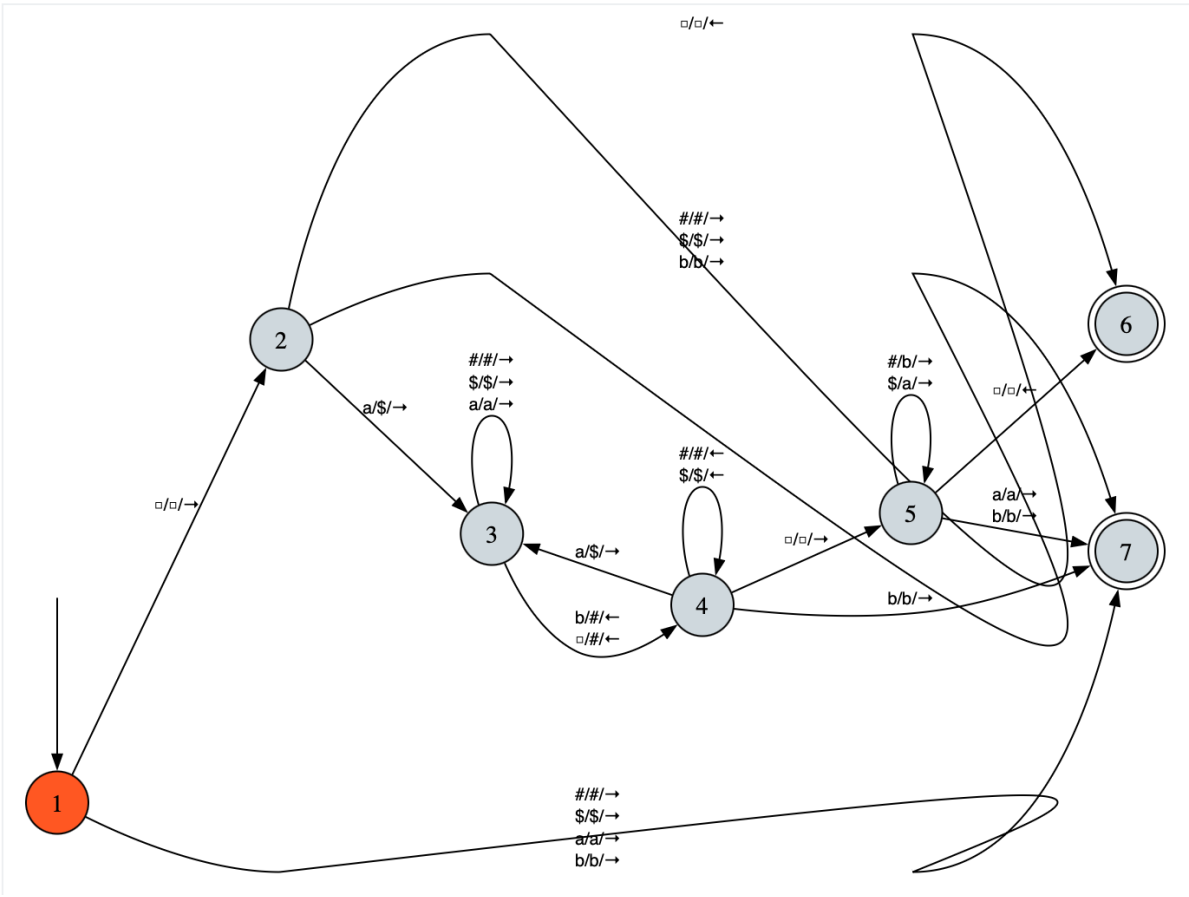
◄ Prev

Next ►

Configuration 0

aaab





Machine: M__9

Zoom −

Zoom +

Zoom: 60%

Input string

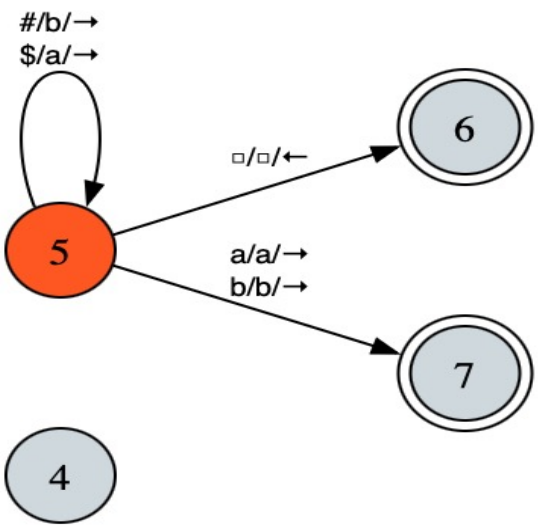
aaab

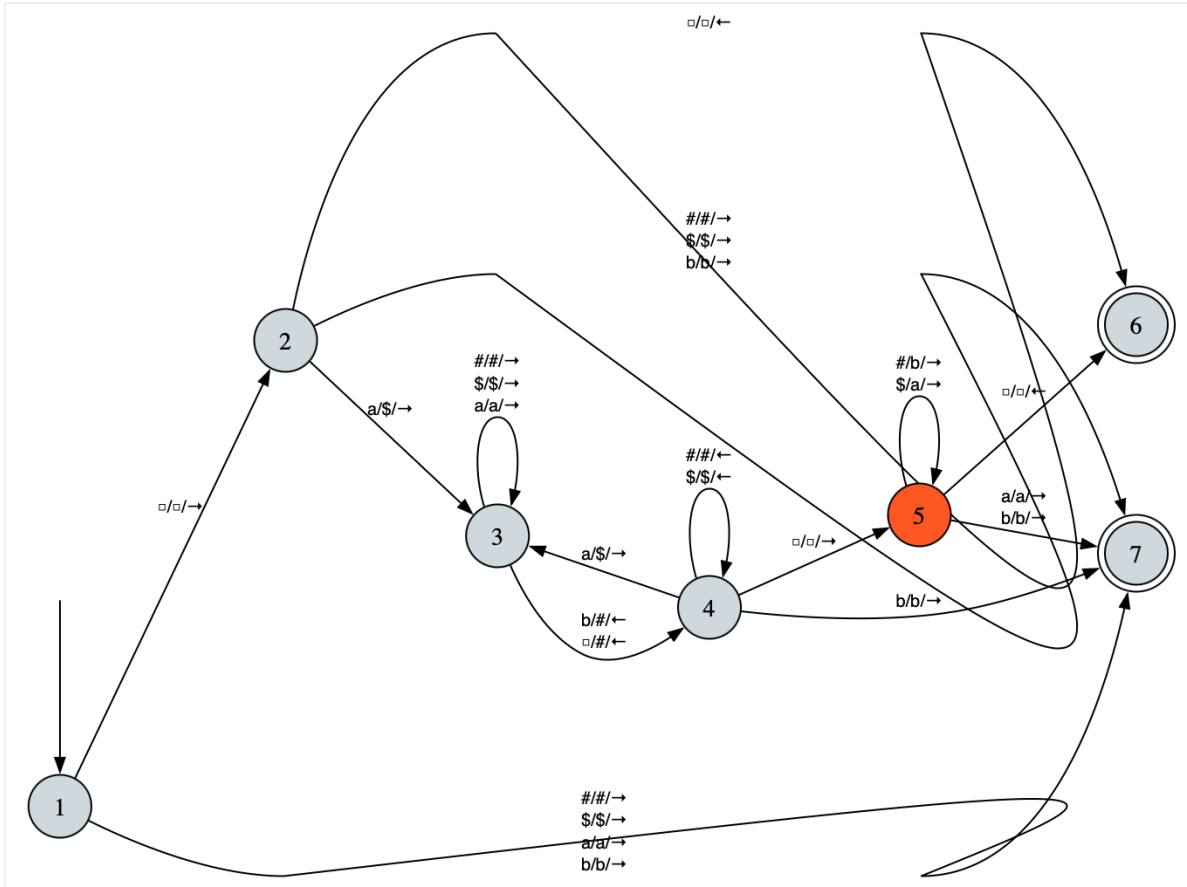
◄ Prev

Next ►

Configuration 21

$$$###





Machine: M__9

Zoom − | Zoom + | Zoom: 60%

Input string

aaab

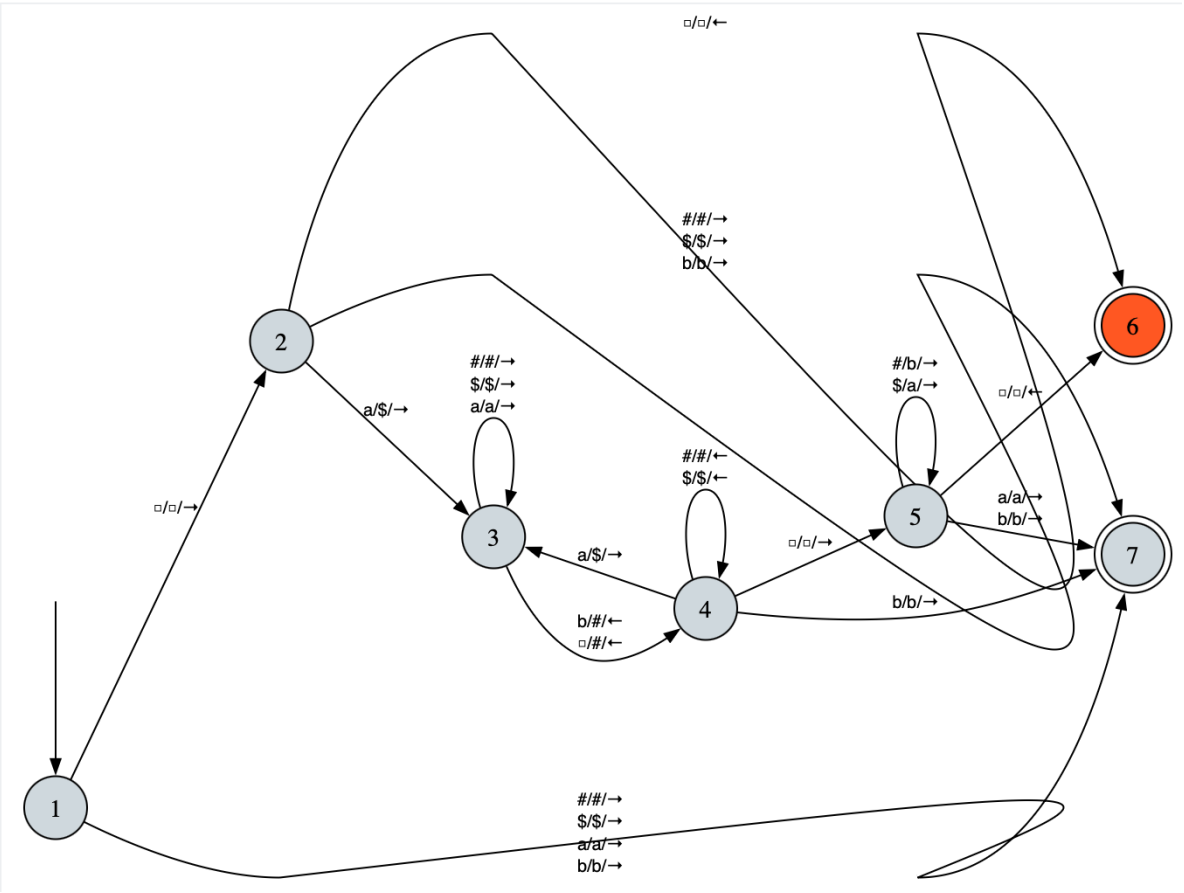
◄ Prev | Next ►

Configuration 28

aaabbb

Result: TM halted.

Show only transitions from current state



END OF SLIDE DECK