CS:314 Fall 2024

Section **04**

Recitation **1**

chris.tu@rutgers.edu

Office hours: 2-3pm Thursday CoRE 335

Today's Topics

Today we'll be reviewing examples of the following:

- Rewrite Systems
- Regular Expressions
- Finite State Machines
- Homework 1

Rewrite System

- A rewrite system is a set of rules which we can apply to transform a string.
- In lecture, we discussed a rewrite system with these characteristics:
  - Input is a binary string, bracketed by $ and # characters. (e.g. $0101#)
  - There are six rewrite rules:
    1. $1 → 1&(AMPERSAND)
    2. $0 → 0$
    3. &1 → 1$
    4. &0 → 0&
    5. $# → A
    6. &# → B

Applying Our Rewrite System

- Given input **$01001#** let's apply the rewrite system from lecture.
- $01001#
- → ?

<u>Our rules:</u>

1. $1 → 1&
2. $0 → 0$
3. &1 → 1$
4. &0 → 0&
5. $# → A
6. &# → B

Applying Our Rewrite System

- Given input **$01001#** let's apply the rewrite system from lecture.

- $01001#
- → 0$1001#   by rule 2
- → ?

Our rules:

1. $1 → 1&
2. $0 → 0$
3. &1 → 1$
4. &0 → 0&
5. $# → A
6. &# → B

Applying Our Rewrite System

- Given input **$01001#** let's apply the rewrite system from lecture.

- $01001#
- → 0$1001# by rule 2
- → 01&001# by rule 1
- → ?

<u>Our rules:</u>

1. $1 → 1&
2. $0 → 0$
3. &1 → 1$
4. &0 → 0&
5. $# → A
6. &# → B

Applying Our Rewrite System

- Given input **$01001#** let's apply the rewrite system from lecture.

- $01001#
- → 0$1001# by rule 2
- → 01&001# by rule 1
- → 010&01# by rule 4
- → ?

Our rules:

1. $1 → 1&
2. $0 → 0$
3. &1 → 1$
4. &0 → 0&
5. $# → A
6. &# → B

Applying Our Rewrite System

- Given input **$01001#** let's apply the rewrite system from lecture.

- $01001#
- → 0$1001# by rule 2
- → 01&001# by rule 1
- → 010&01# by rule 4
- → 0100&1# by rule 4
- → ?

<u>Our rules:</u>

1. $1 → 1&
2. $0 → 0$
3. &1 → 1$
4. &0 → 0&
5. $# → A
6. &# → B

Applying Our Rewrite System

- Given input **$01001#** let's apply the rewrite system from lecture.

- $01001#
- → 0$1001# by rule 2
- → 01&001# by rule 1
- → 010&01# by rule 4
- → 0100&1# by rule 4
- → 01001$# by rule 3
- → ?

<u>Our rules:</u>

1. $1 → 1&
2. $0 → 0$
3. &1 → 1$
4. &0 → 0&
5. $# → A
6. &# → B

Applying Our Rewrite System

- Given input **$01001#** let's apply the rewrite system from lecture.

- $01001#
- → 0$1001# by rule 2
- → 01&001# by rule 1
- → 010&01# by rule 4
- → 0100&1# by rule 4
- → 01001$# by rule 3
- → 01001A by rule 5

<u>Our rules:</u>

1. $1 → 1&
2. $0 → 0$
3. &1 → 1$
4. &0 → 0&
5. $# → A
6. &# → B

Another Rewrite System

- Say we want to make a rewrite system to decide if a number is even or odd.
- Assume the input will be a **nonempty binary string,** bracketed by $ and #.
- Let's design our system to produce "O" if the number is odd, and to produce "E" if the number is even.
- Note that a binary number is odd if and only if it ends with **1**.

Another Rewrite System

- Say we want to make a rewrite system to decide if a number is even or odd.
- First, let's create rewrite rules that will get rid of unnecessary digits:

Another Rewrite System

- Say we want to make a rewrite system to decide if a number is even or odd.
- First, let's create rewrite rules that will get rid of unnecessary digits:
  - Rule 1: $10 \to 0$
  - Rule 2: $00 \to 0$
  - Rule 3: $01 \to 1$
  - Rule 4: $11 \to 1$
- Note that “one rule at a time” doesn’t apply
- Applying these rules will leave us with either $0# or $1#.

Another Rewrite System

- Say we want to make a rewrite system to decide if a number is even or odd.
- First, let's create rewrite rules that will get rid of unnecessary digits:
  - Rule 1: $10 \to 0$
  - Rule 2: $00 \to 0$
  - Rule 3: $01 \to 1$
  - Rule 4: $11 \to 1$
- Finally, let's create rules to produce "E" for even or "O" for odd:
  - Rule 5: $\$0\# \to E$
  - Rule 6: $\$1\# \to O$

Regular Expressions

- Regular expressions are a way of specifying regular languages.
- They use these rules:
  - **(x)** accepts **x**
  - **x*** accepts 0 or more copies of **x**
  - **x^+** accepts 1 or more copies of **x**
  - **xy** accepts **x** followed by **y**
  - **x|y** accepts either **x** or **y**
- The above rules are listed in descending order of precedence.
  - For example: a|bc accepts either "a" or "bc", **not** "ac"
- Note that it's common to use **a-z** as shorthand for (a|b|...|z)

Precedence of Regex Rules

1. Kleene * or Positive +
2. Concatenation
3. Union

“Clean your Cat on Union Street.”

“Kleene*” conCATenation Union Street

Parentheses precedes all.

Regular Expressions

- Some regular expressions' languages can be simply described in english.
- How can we describe the languages of the following regular expressions?

1. (0|1)*
2. 1(0|1)*
3. (a-z|A-Z)+
4. (a-z|A-Z)(a-z|A-Z|0-9)*

Regular Expressions

- Some regular expressions' languages can be simply described in english.
- How can we describe the languages of the following regular expressions?

1. (0|1)*

   Answer: The set of all binary strings. The Kleene * refers to 0 or more copies of whatever is inside of the parenthesis.

2. 1(0|1)*

3. (a-z|A-Z)+

4. (a-z|A-Z)(a-z|A-Z|0-9)*

Regular Expressions

- Some regular expressions' languages can be simply described in english.
- How can we describe the languages of the following regular expressions?

1. (0|1)*

   Answer: The set of all binary strings. The Kleene * refers to 0 or more copies of whatever is inside of the parenthesis.
2. 1(0|1)*

   Answer: The set of all binary strings that start with a 1.
3. (a-z|A-Z)+
4. (a-z|A-Z)(a-z|A-Z|0-9)*

Regular Expressions

- Some regular expressions' languages can be simply described in english.
- How can we describe the languages of the following regular expressions?

1. (0|1)*

   Answer: The set of all binary strings.
2. 1(0|1)*

   Answer: The set of all binary strings that start with a 1.
3. (a-z|A-Z)+

   Answer: The set of all non-empty alphabetic strings.
4. (a-z|A-Z)(a-z|A-Z|0-9)*

Regular Expressions

- Some regular expressions' languages can be simply described in english.
- How can we describe the languages of the following regular expressions?

1. (0|1)*

   Answer: The set of all binary strings.
2. 1(0|1)*

   Answer: The set of all binary strings that start with a 1.
3. (a-z|A-Z)+

   Answer: The set of all non-empty alphabetic strings.
4. (a-z|A-Z)(a-z|A-Z|0-9)*

   Answer: The set of all alphanumeric strings that start with a letter.

Regular Expressions

- Let's write regular expressions that produce the following languages.
- Note that these may have multiple solutions.

1. All binary strings that are divisible by 4.
2. All binary strings that contain **exactly** one 0.
3. All binary strings that contain **at least** one 0.
4. All alphanumeric strings that do **not** have consecutive digits.

Regular Expressions

- Let's write regular expressions that produce the following languages.
- Note that these may have multiple solutions.

1. All binary strings that are divisible by 4.

   Answer: (0|1)*00|0 - Every power of 2 after 2^2 (4) is divisible by 4. Therefore either the last two bits need to be zero, or the binary number is zero itself.
2. All binary strings that contain **exactly** one 0.
3. All binary strings that contain **at least** one 0.
4. All alphanumeric strings that do **not** have consecutive digits.

Regular Expressions

- Let's write regular expressions that produce the following languages.
- Note that these may have multiple solutions.

1. All binary strings that are divisible by 4.
   Answer: (0|1)*00|0
2. All binary strings that contain **exactly** one 0.
   Answer: 1*01*
3. All binary strings that contain **at least** one 0.

4. All alphanumeric strings that do **not** have consecutive digits.

Regular Expressions

- Let's write regular expressions that produce the following languages.
- Note that these may have multiple solutions.

1. All binary strings that are divisible by 4.

   Answer: (0|1)*00|0
2. All binary strings that contain **exactly** one 0.

   Answer: 1*01*
3. All binary strings that contain **at least** one 0.

   Answer: (0|1)*0(0|1)*
4. All alphanumeric strings that do **not** have consecutive digits.

Regular Expressions

- Let's write regular expressions that produce the following languages.
- Note that these may have multiple solutions.

1. All binary strings that are divisible by 4.
   Answer: (0|1)*00|0
2. All binary strings that contain **exactly** one 0.
   Answer: 1*01*
3. All binary strings that contain **at least** one 0.
   Answer: (0|1)*0(0|1)*
4. All alphanumeric strings that do **not** have consecutive digits.
   Answer: (0-9|ε)((a-z|A-Z)(0-9|ε))*

Start with a number or not. Then you pick 0 or more combinations of a substring of fixed size that either is a letter number pair, or just letter(s)

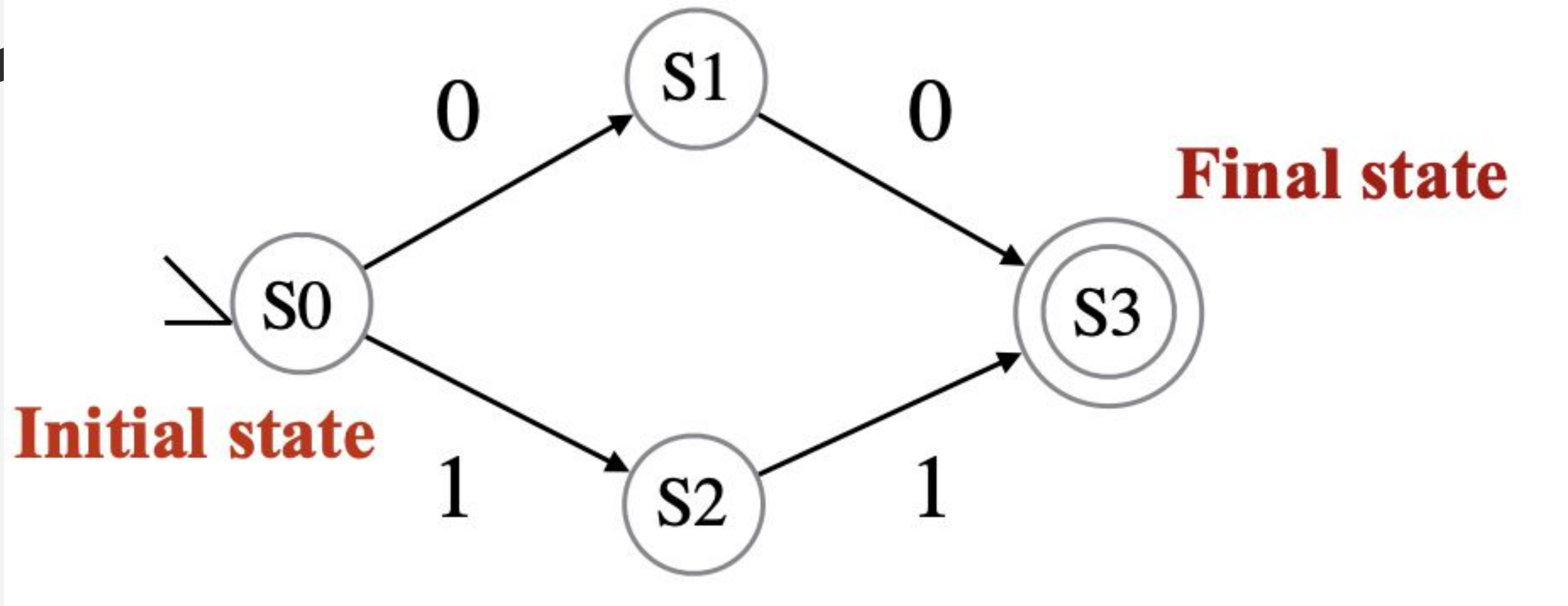
Regular Expressions

- Not all languages are regular!
- Here are some languages which regular expressions cannot recognize:
  - The set of binary strings with the same number of 0s as 1s.
  - The set of binary strings that are palindromes.
  - The set of prime numbers.
  - The set of correctly formatted Java methods.
- Most of the above examples are context-free languages, which you'll learn more about later in this course.

Regular Expressions

- Regular Expressions are not good at remembering the state.

Finite State Machines (a.k.a. Finite State



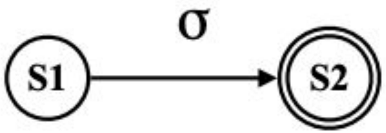
formally,

$S \times \Sigma \to S$

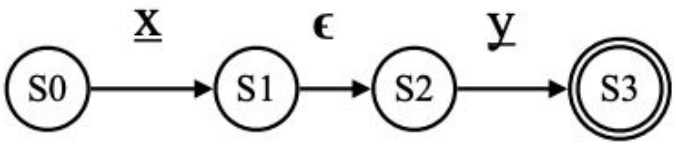
Finite State Machines (a.k.a. Finite State Automata)

- There are two types of finite state machines: **deterministic finite automata (DFAs)** and **non-deterministic finite automata (NFAs)**.
- Technically speaking, every FSA is an NFA.
- **An FSA is also a DFA only if the following conditions are true:**
  - Given any <*state*, *input*> tuple, there is no more than one possible transition.
  - The empty string (epsilon) is not used for any transitions.

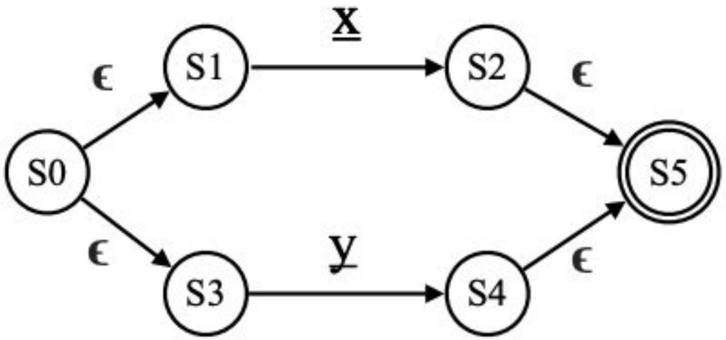
DFA for σ



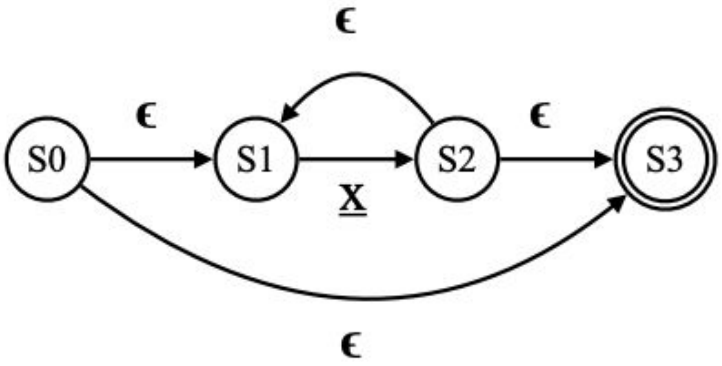
NFA for xy



NFA for x|y



NFA for x*

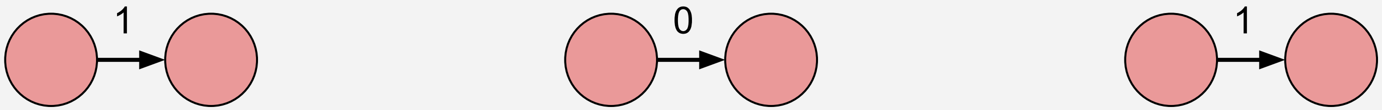


Thompson's Construction

- Let's build an NFA for this regular expression from earlier: 1*01*

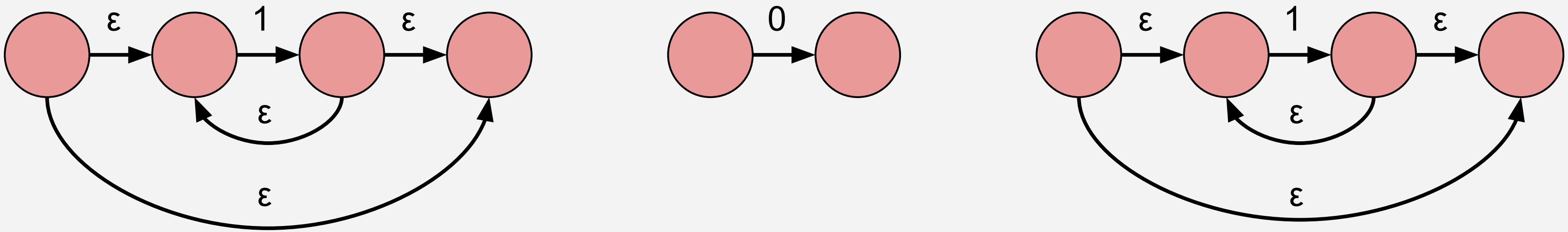
Thompson's Construction

- Let's build an NFA for this regular expression from earlier: 1*01*
- Let's start by making state transitions for each of the symbols:



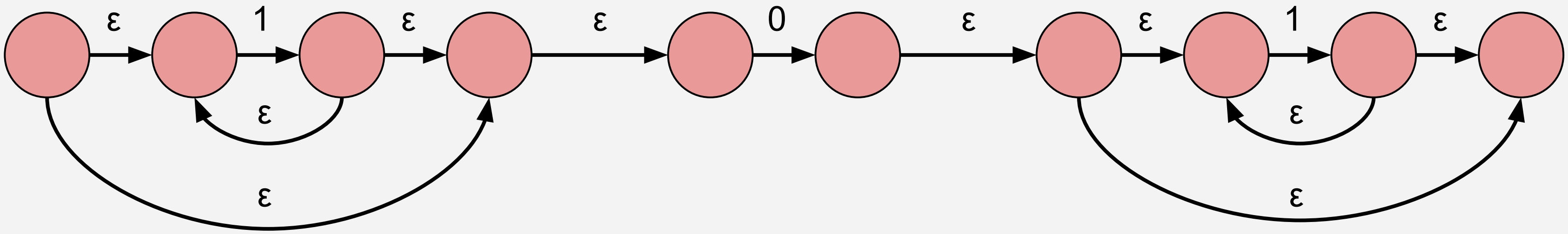
Thompson's Construction

- Let's build an NFA for this regular expression from earlier: 1*01*
- Since Kleene star has highest precedence here, let's handle that now:



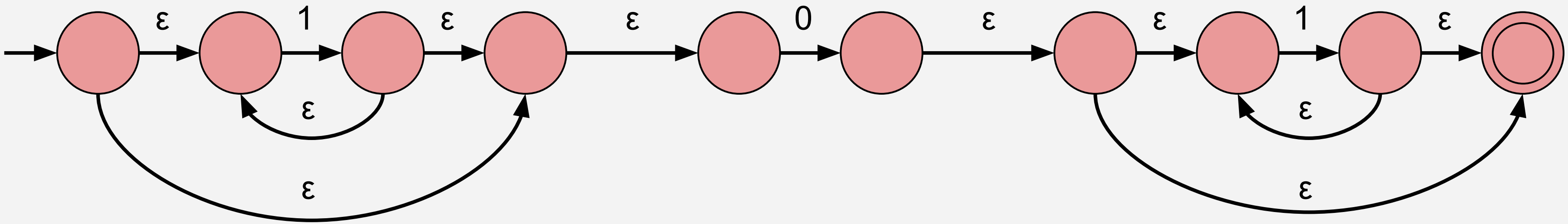
Thompson's Construction

- Let's build an NFA for this regular expression from earlier: 1*01*
- Now that we have three FSAs for the subexpressions 1*, 0, and 1*, we can just connect them via concatenation:



Thompson's Construction

- Let's build an NFA for this regular expression from earlier: 1*01*
- Finally, we mark the start state and the final states:



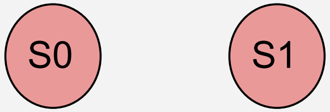
Another FSA Example: Modulo

Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
    (Reminder: "x modulo y = z" means that the remainder after dividing x by y is z)
- What states will we need?
  - I.e. what do we need to remember about the parts of the input that we've seen so far?

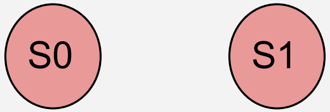
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
    (Reminder: "x modulo y = z" means that the remainder after dividing x by y is z)
- What states will we need?
- State S0, for the case that the number modulo 2 is 0.
- State S1, for the case that the number modulo 2 is 1.
  - The states' names are arbitrary, but this pattern is convenient.



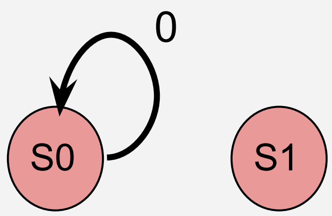
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- If we're in state S0 and the next input is a 0, what state do we transition to?
  - Hint: appending a 0 to a binary number multiplies its value by 2.



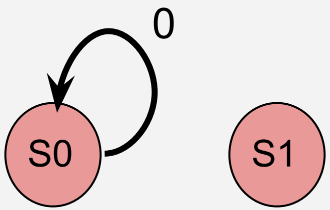
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- If we're in state S0 and the next input is a 0, what state do we transition to?
- Note that if (x mod 2) = 0, then (2*x mod 2) = 0.
- Therefore <S0, 0> → S0



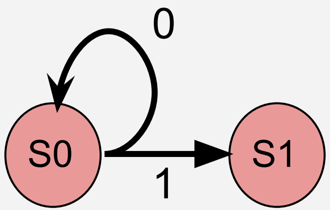
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- If we're in state S0 and the next input is a 1, what state do we transition to?
  - Hint: appending a 1 to a binary number multiplies its value by 2 and adds 1.



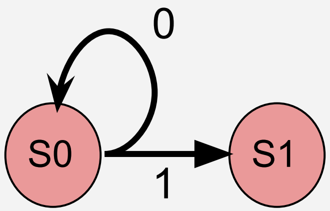
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- If we're in state S0 and the next input is a 1, what state do we transition to?
- Note that if (x mod 2) = 0, then (2*x+1 mod 2) = 1.
- Therefore <S0, 1> → S1



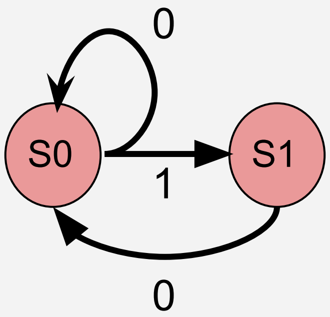
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
    - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- What transition should we have from state S1 on input 0?



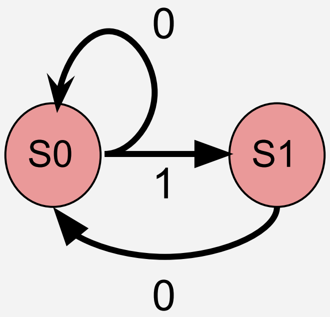
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- What transition should we have from state S1 on input 0?
- <S1, 0> → S0



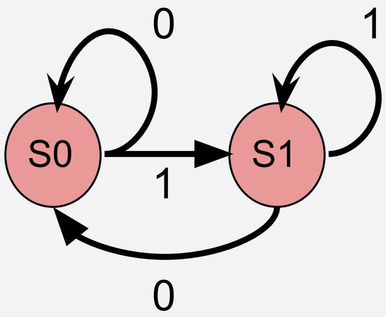
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- What transition should we have from state S1 on input 1?



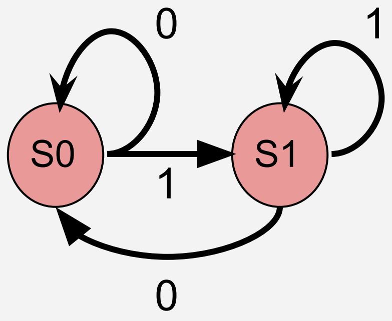
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- What transition should we have from state S1 on input 1?
- <S1, 1> → S1



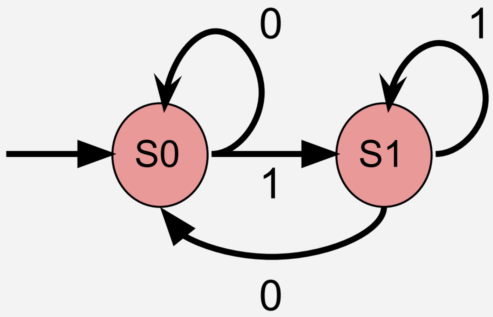
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- What should our start state be?



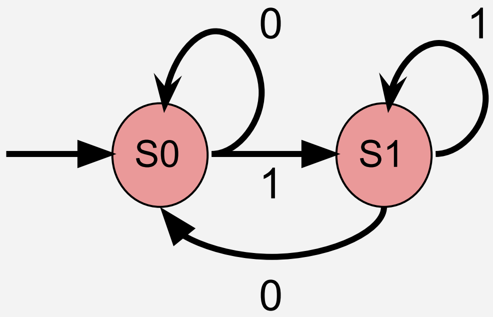
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- What should our **start state** s be?
- An empty binary string can be considered to have value 0.
- Therefore s = S0



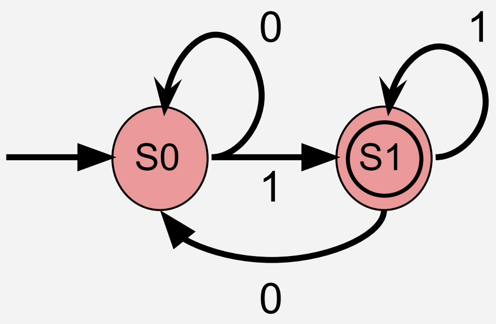
Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- Finally, what should our set of final states contain?



Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
    - In other words, the number modulo 2 is 1.
- S0 means the number mod 2 is 0; S1 means the number mod 2 is 1.
- Finally, what should our set of final states contain?
- F = {S1}



Finite State Machine For Modulo

- Say we want an FSA that recognizes odd binary numbers.
  - In other words, the number modulo 2 is 1.
- Each state here has only one outgoing edge per symbol, and none of the transitions use ε, so this FSA is actually a DFA.
- A similar strategy as shown in these slides can be used to construct a larger DFA that performs modulo for a larger number than 2.