# P, NP, NP-Completeness

P, NP, NP-Completeness, and Reductions

# Introduction

- Some problems can be solved in polynomial time.
  - as most of the problems we have seen in the previous lectures
- You've heard some other problems are "NP-hard" or "NP-complete".
- This lecture:
  - Learn what exactly do we mean by NP-hardness, or NP-completeness.
  - Understand why people believe these problems are hard.

# Let’s first see some famous NP-hard problems

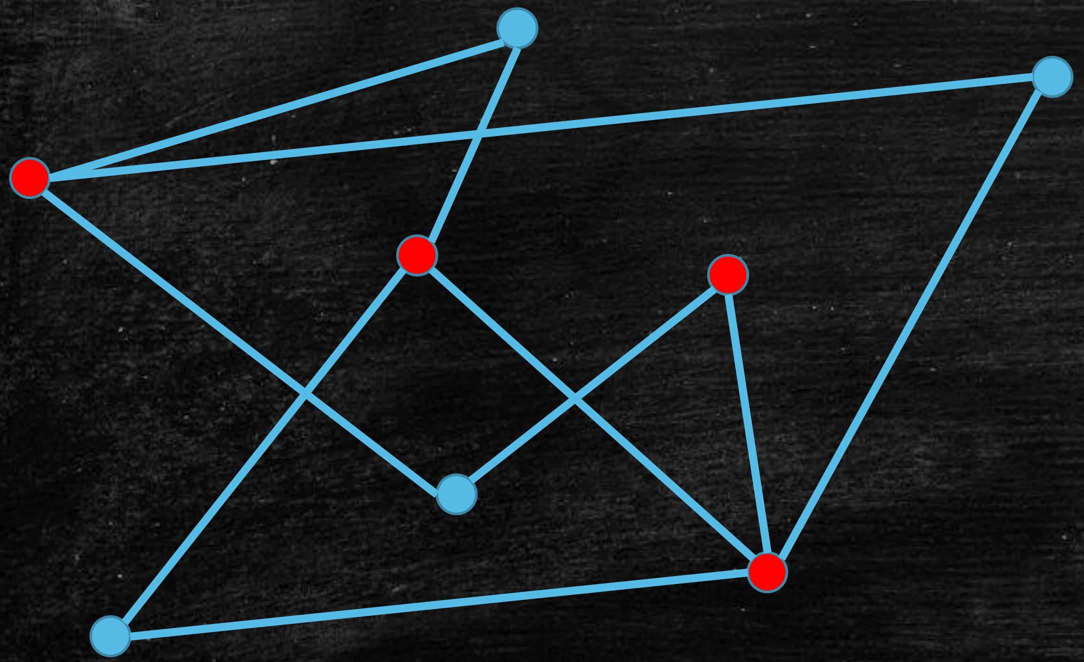
- SAT
- Vertex Cover
- Independent Set
- Subset Sum
- Hamiltonian Path

# SAT (Boolean Satisfiability Problem)

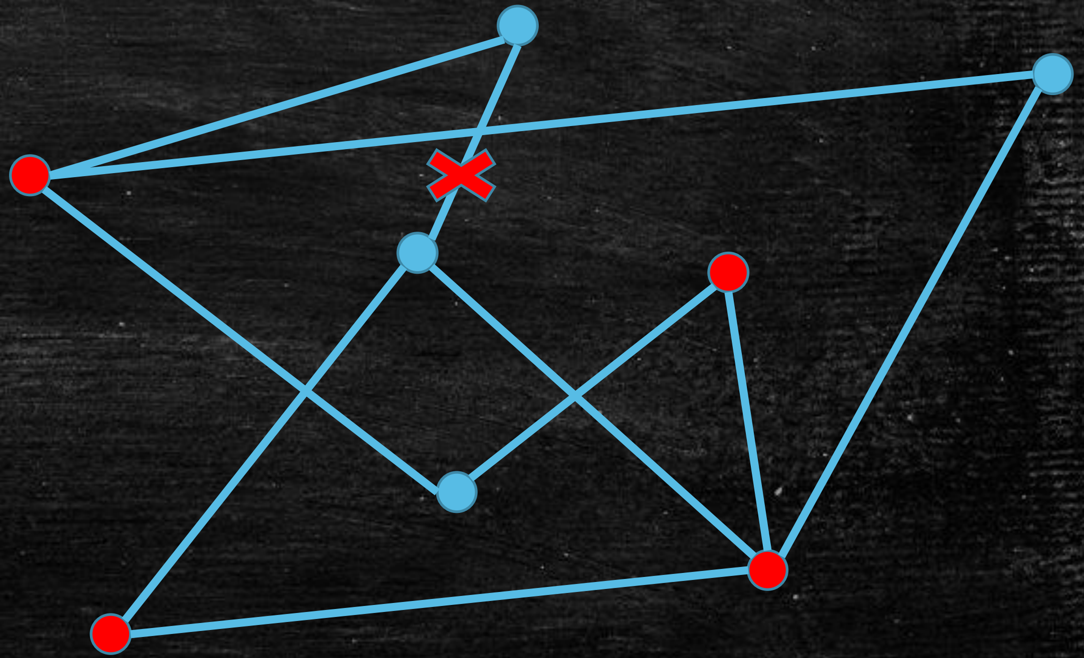
- A Boolean formula is built from variables, operators AND ($\wedge$), OR ($\vee$), NOT ($\neg$), and parentheses.
  - Example: $(x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$
- A Boolean formula is in conjunctive normal form (CNF) if it is an "AND" of many clauses:
  - Each clause contains "OR" of literals:
    - A literal is a variable $x_i$ or its negation $\neg x_i$
  - The example is in CNF; it has three clauses: $(x_1 \vee x_3 \vee \neg x_4)$, $(x_2 \vee \neg x_3)$ and $(\neg x_1 \vee \neg x_2)$
- [SAT Problem] Given a CNF formula $\phi$, decide if there is a value assignment to the variables to make $\phi$ $\mathrm{true}$.
  - This is true for the example above: $x_1 = \mathrm{true}, x_2 = \mathrm{false}, x_3 = \mathrm{false}$.

# Vertex Cover

- Given an undirected graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a vertex cover if $S$ contains at least one endpoint of every vertex.



a vertex cover

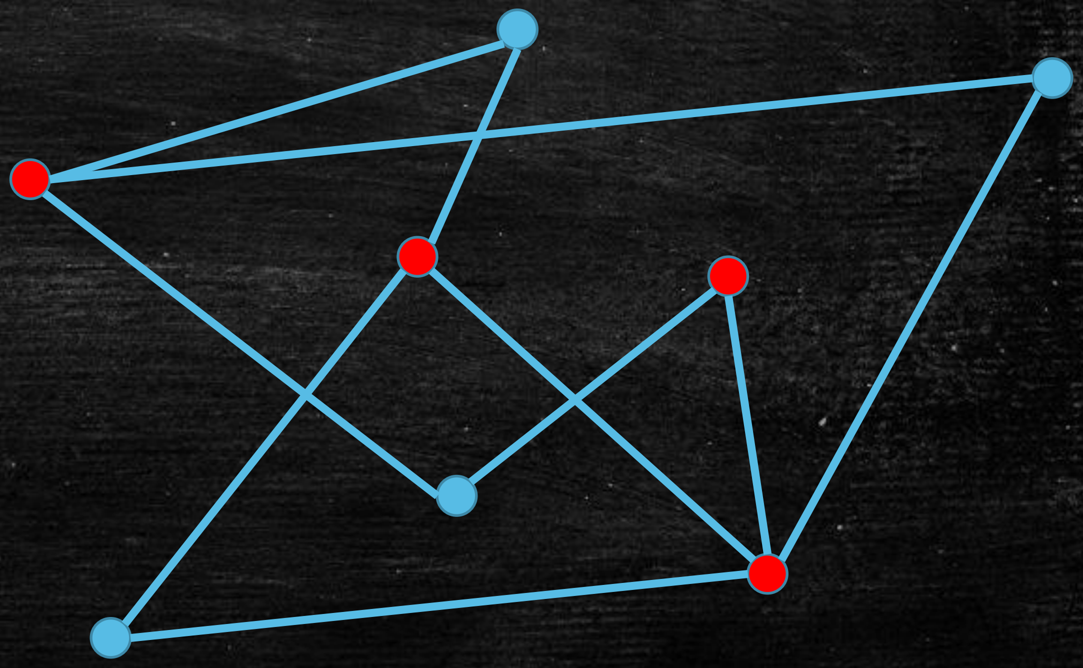


not a vertex cover

# Vertex Cover Problem

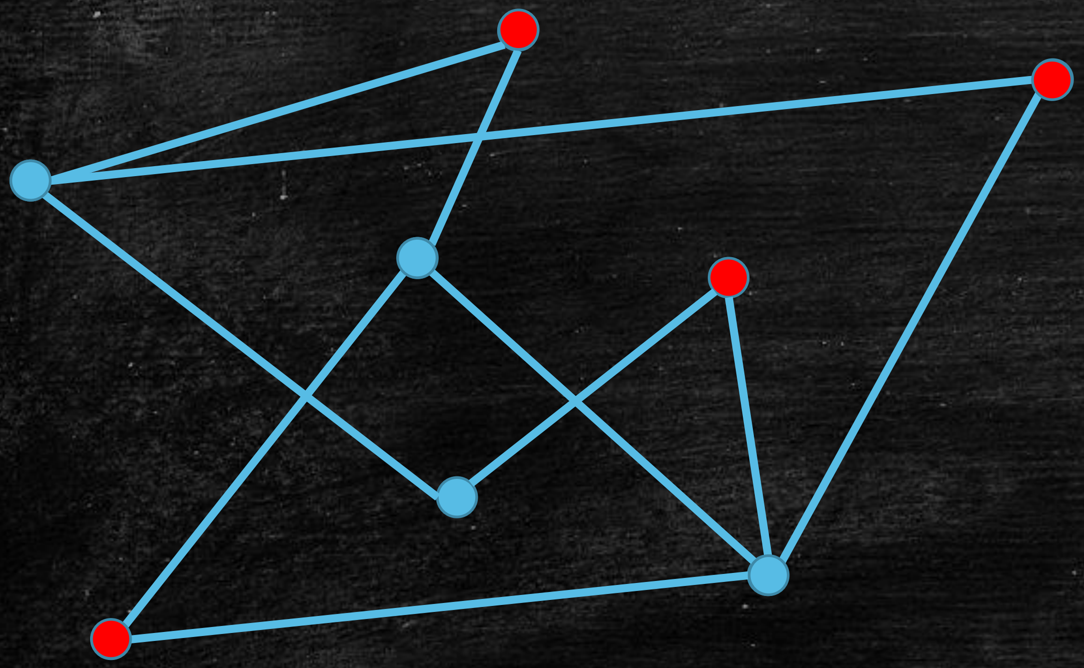
- [Vertex Cover Problem] Given an undirected graph $G = (V, E)$ and $k \in \mathbb{Z}^+$, decide if the graph has a vertex cover of size $k$.

For this graph and $k = 4$, the output should be yes.

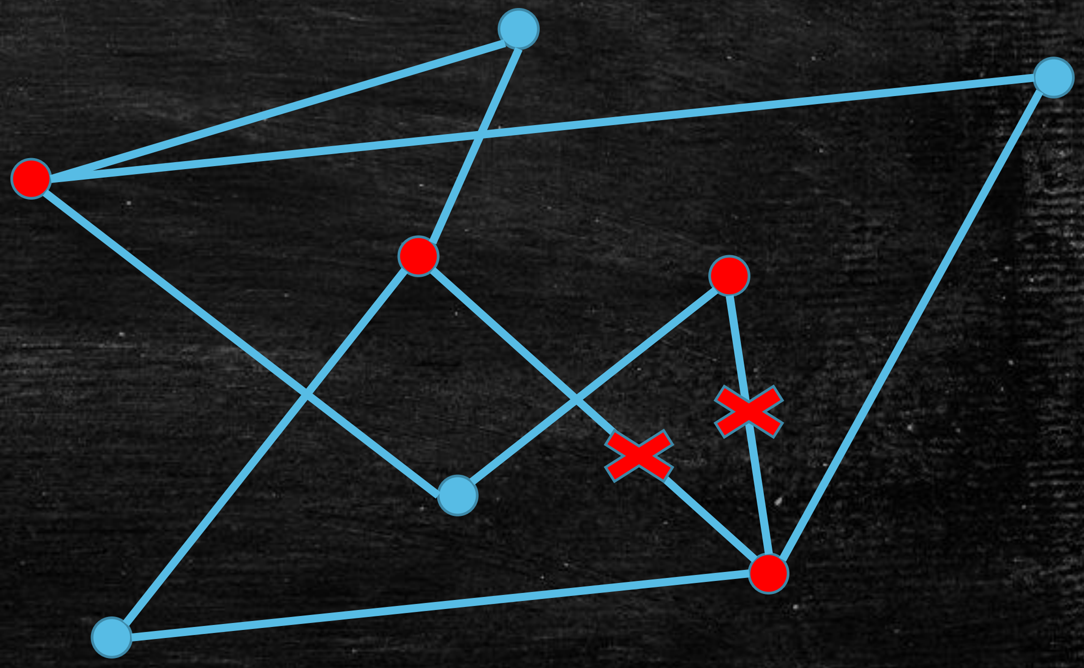


# Independent Set

- Given an undirected graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is an independent set if there is no edge between any two vertices in $S$.



an independent set

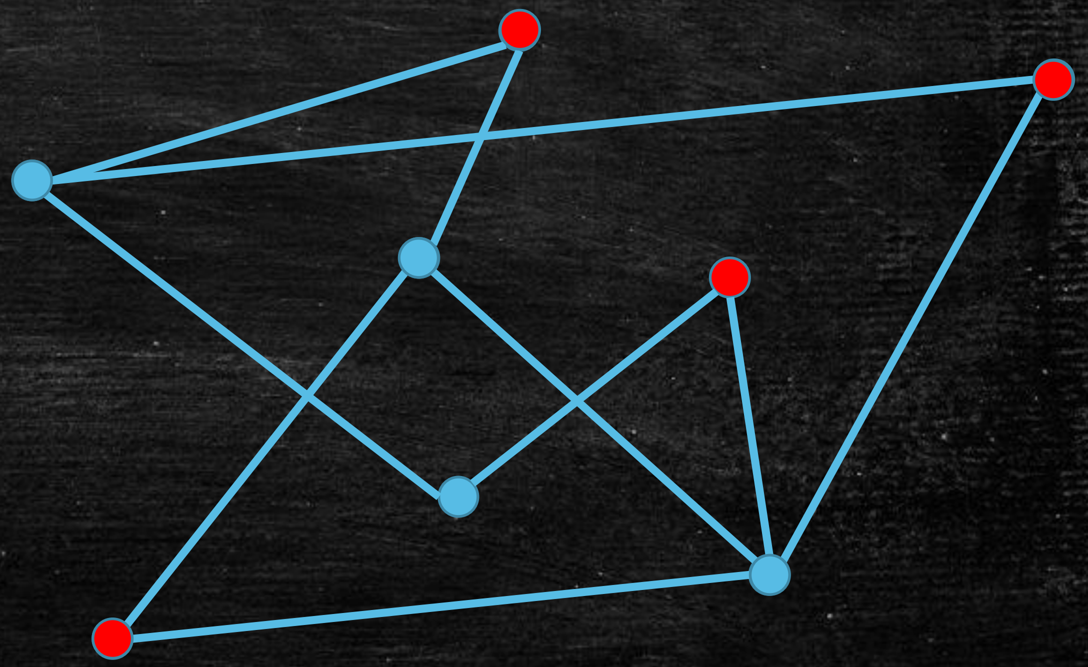


not an independent set

# Independent Set Problem

- [Independent Set Problem] Given an undirected graph $G = (V, E)$ and $k \in \mathbb{Z}^+$, decide if the graph has an independent set of size $k$.

For this graph and $k = 4$, the output should be yes.

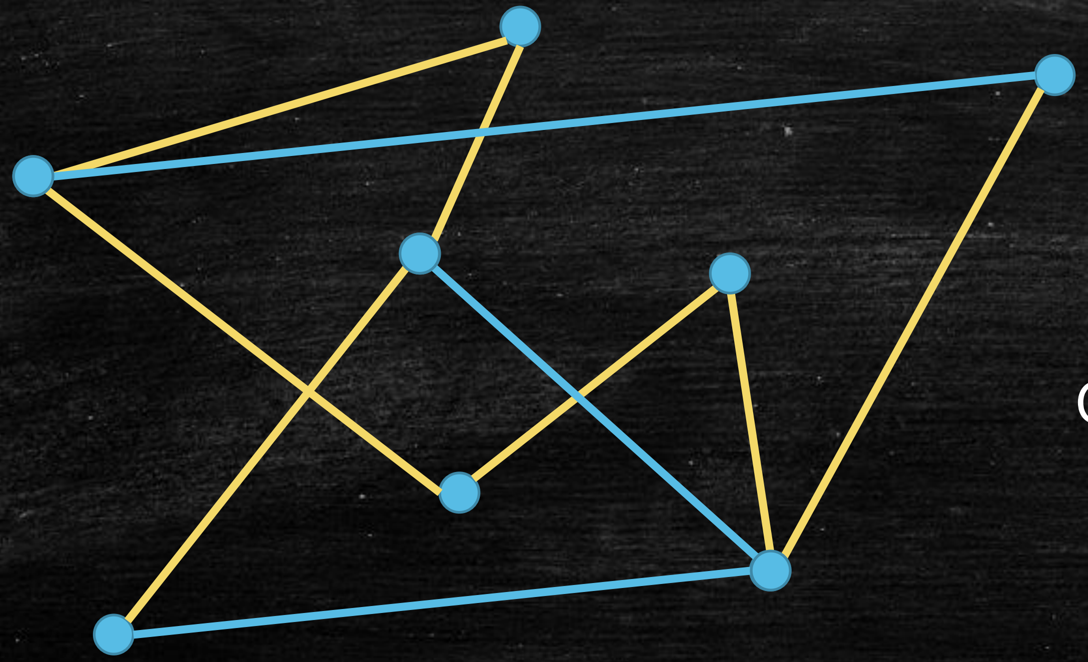


# Subset Sum Problem

- [Subset Sum Problem] Given a collection of integers $S = \{a_1, \ldots, a_n\}$ and $k \in \mathbb{Z}^+$, decide if there is a sub-collection $T \subseteq S$ such that $\sum_{a_i \in T} a_i = k$.
- The output should be $\mathrm{yes}$ for $S = \{1,1,6,13,27\}$ and $k = 21$, as $1 + 1 + 6 + 13 = 21$.
- The output should be $\mathrm{no}$ for $S = \{1,1,6,13,27\}$ and $k = 22$.

# Hamiltonian Path Problem

- Given an undirected graph $G = (V, E)$, a Hamiltonian path is a path containing each vertex exactly once.
- [Hamiltonian Path Problem] Given an undirected graph $G = (V, E)$, decide if it contains a Hamiltonian path.



Output should be yes

# In this lecture, we will only focus on…

- Decision Problems: those with output yes or no.
- Polynomial Time vs Not Polynomial Time
  - E.g., we will not care about $O(n)$ or $O(n^2)$
  - "Easy" Problems: those can be solved in polynomial time
  - "Hard" problems: those for which people believe cannot be solved in polynomial time

# Decision Problem – Formal Definition

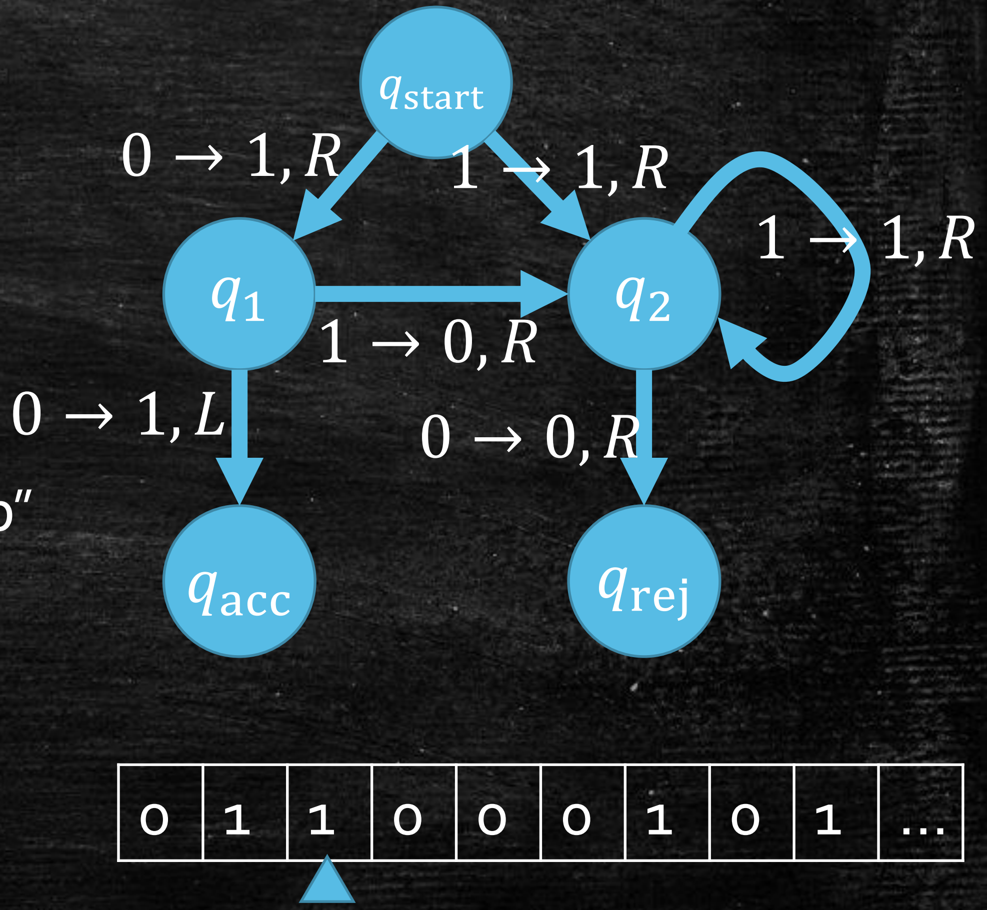
- A decision problem is a function $f : \Sigma^* \to \{0,1\}$
- $\Sigma$ - set of alphabets: for example, binary alphabets $\Sigma = \{0,1\}$
- $\Sigma^n$ - set of strings using alphabets in $\Sigma$ with length $n$
- $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$ - set of all strings with any lengths
- $x \in \Sigma^*$ - an instance
- $f(x) = 1$: $x$ is a yes instance
  - E.g., $x$ encodes $G$ and $k$ where $G$ has a $k$-vertex cover
- $f(x) = 0$: $x$ is a no instance
  - E.g., $x$ encodes $G$ and $k$ where $G$ does not have a $k$-vertex cover
  - Or $x$ is not a valid encoding of $G$ and $k$

# Problems That Are "Easy"

- A decision problem $f: \Sigma^* \to \{0,1\}$ is "easy" if there is a polynomial time algorithm $\mathcal{A}$ that computes it.
- That is, $\mathcal{A}(x) = f(x)$ always holds.
- Polynomial time: $\mathcal{A}(x)$ terminates in $|x|^{O(1)}$ steps.
- But wait! What exactly is an algorithm??

# Turing Machine (TM)

- An abstract machine that is a prototype of modern computers.
- A Turing Machine is a triple $(Q, \Sigma, \delta)$
  - one tape: contains infinitely many cells
    - Each cell can store an alphabet
  - A moving head pointing at a cell of the tape
  - $\Sigma$: set of alphabets
  - $Q$: set of states, each state specifying "the current step"
  - Transition function $\delta: Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$
    - instructions on how to move to the next step
    - Input: current state, current alphabet the head is reading
    - Output: next state, new alphabet written on the current position of the head, move to left ($L$) or right ($R$) by one cell

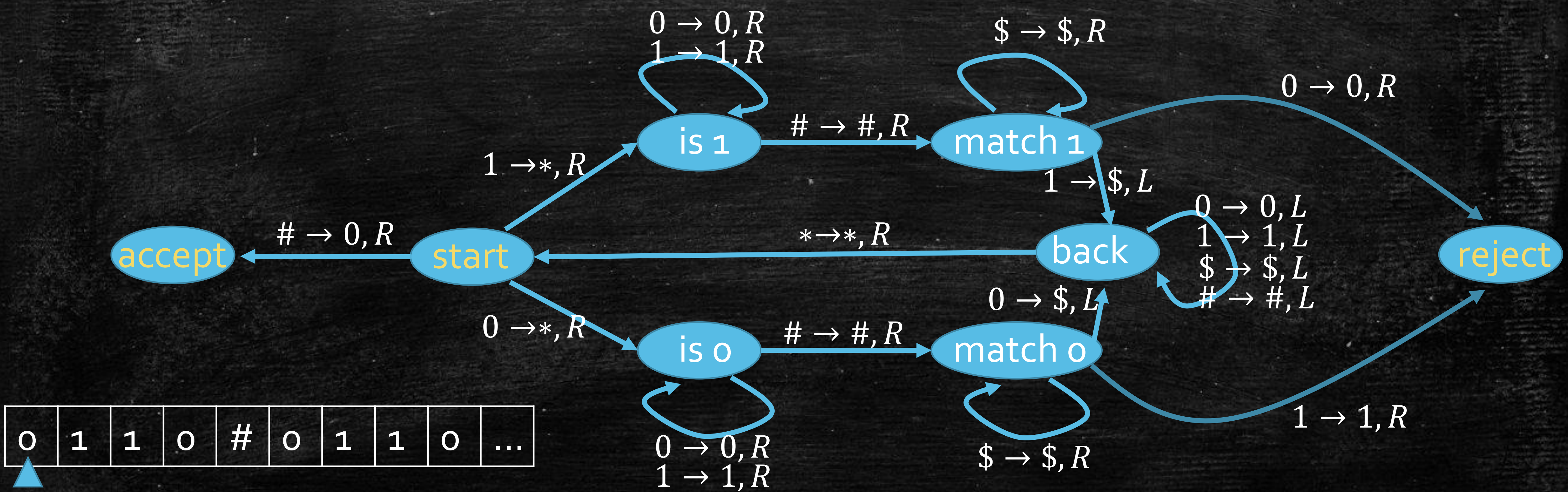


# Turing Machine: Start and Terminate

- Start:
  - At a special state called starting state: $q_{\text{start}} \in Q$
  - Input is loaded to the tape
  - Moving Head is pointing at the first cell
- Terminate:
  - Two special state called halting states: $q_{\text{acc}}$ and $q_{\text{rej}}$
  - TM terminates when reaching a halting state
  - TM accepts a string if $q_{\text{acc}}$ is reached
  - TM rejects a string if $q_{\text{rej}}$ is reached
  - TM's output is the content on the tape when TM terminates

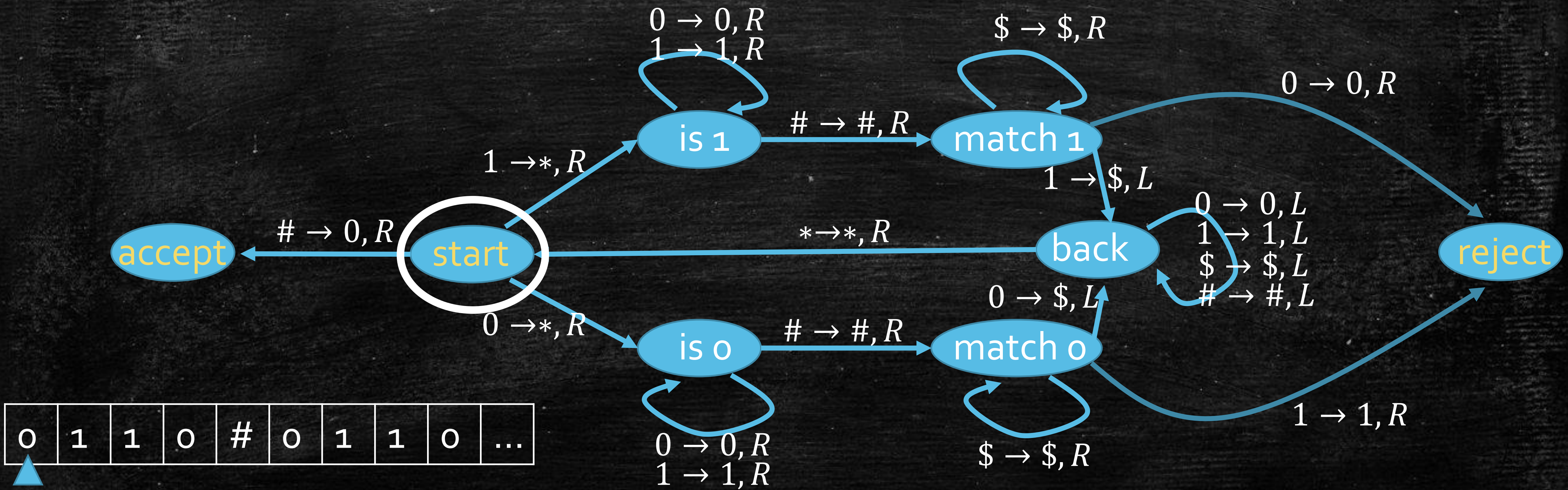
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



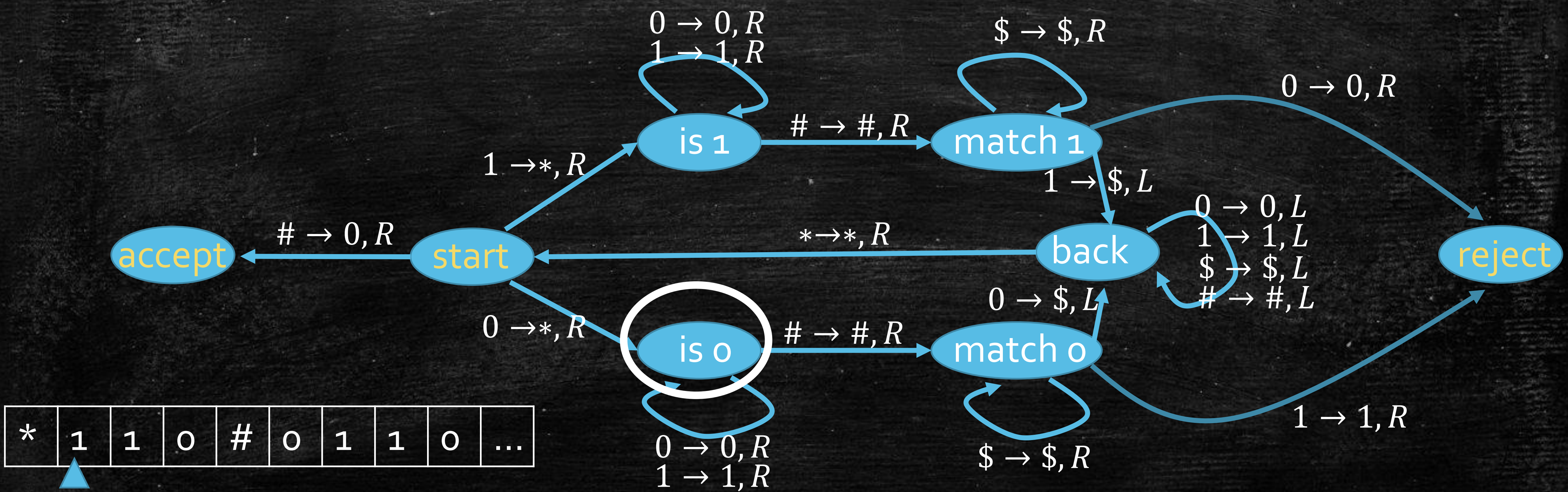
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



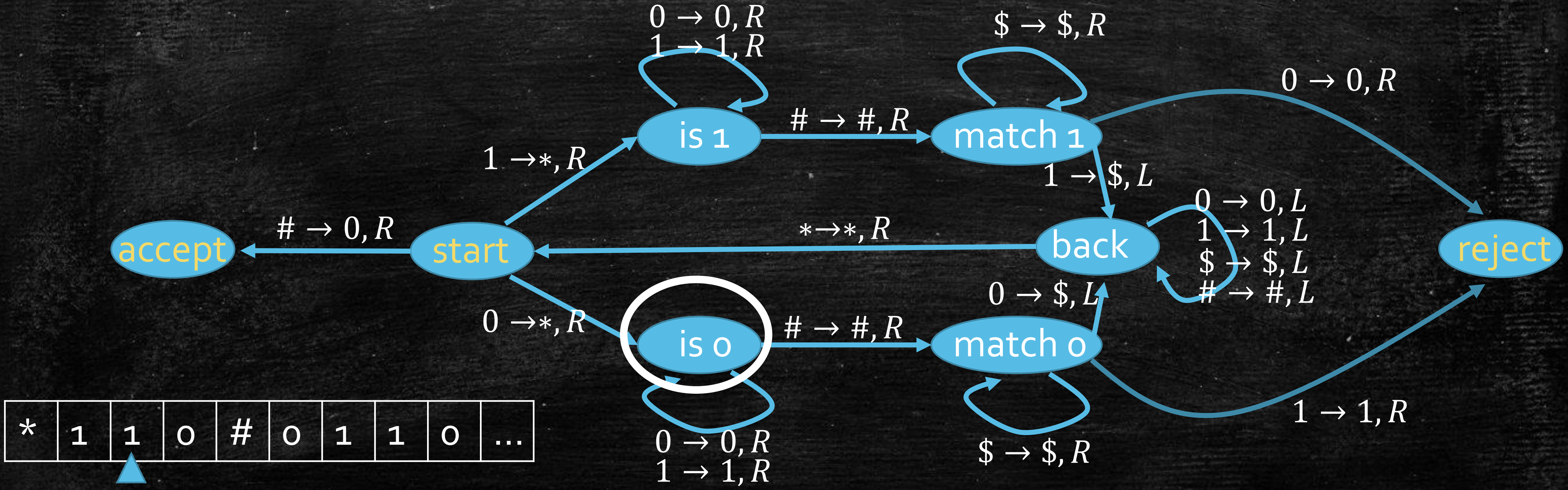
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



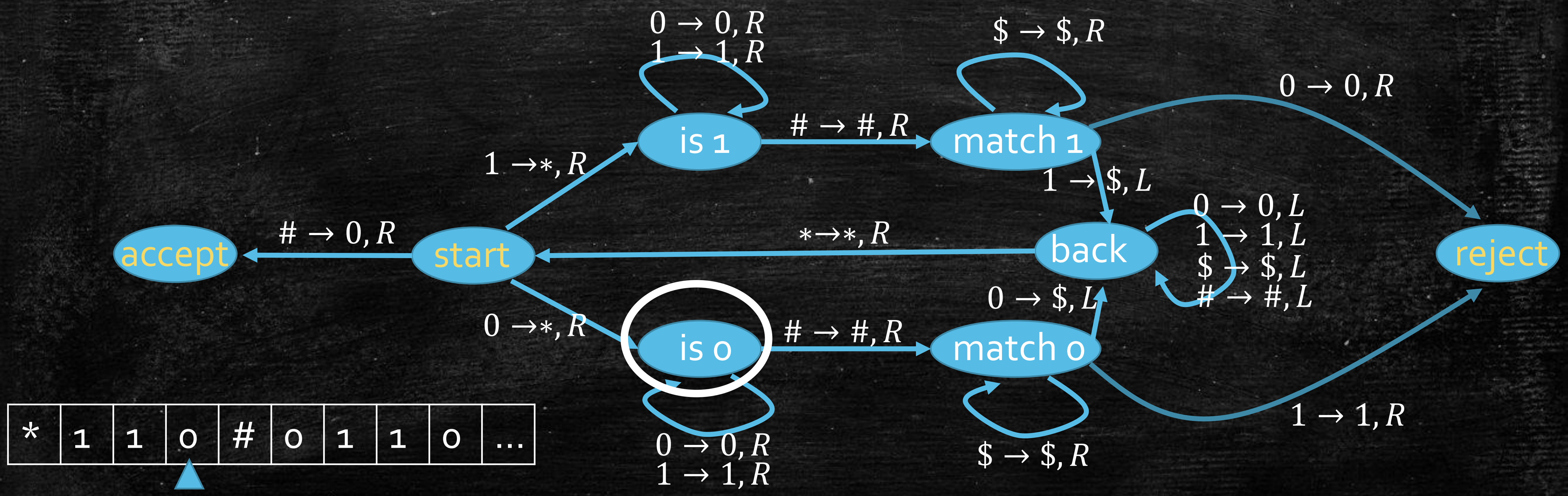
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



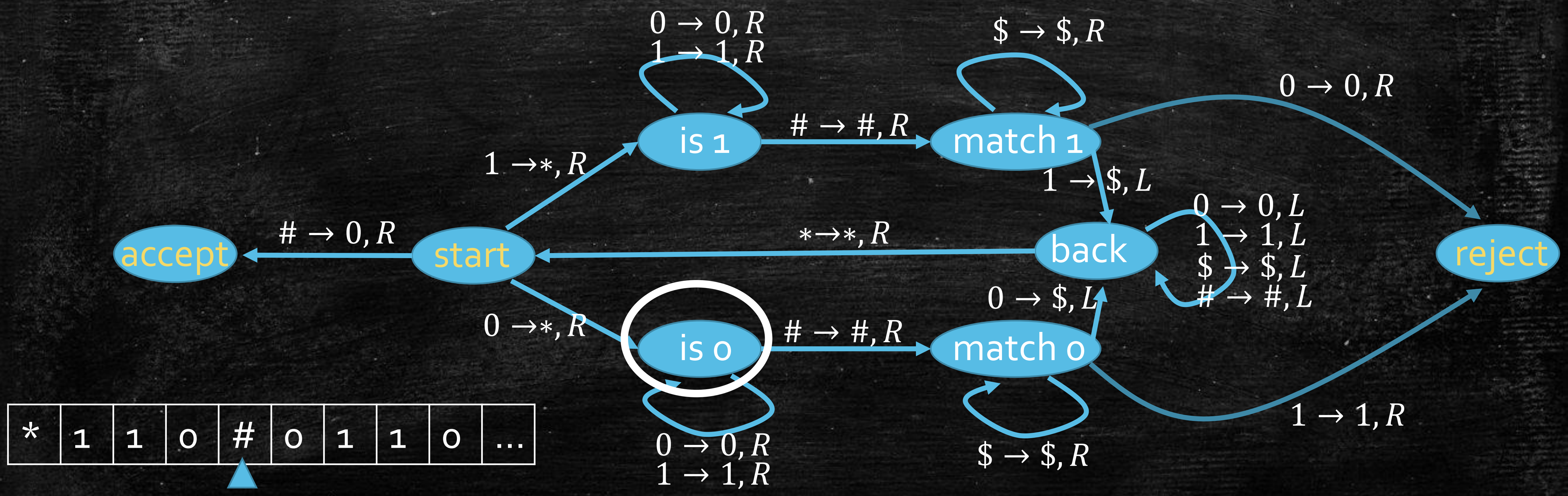
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



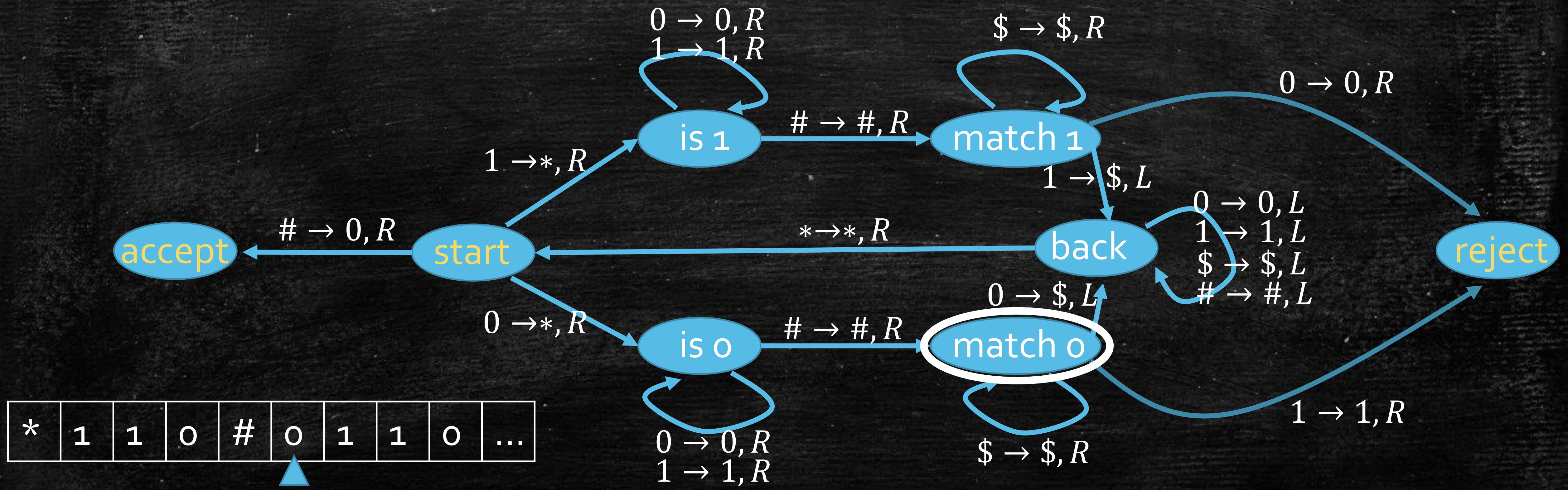
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



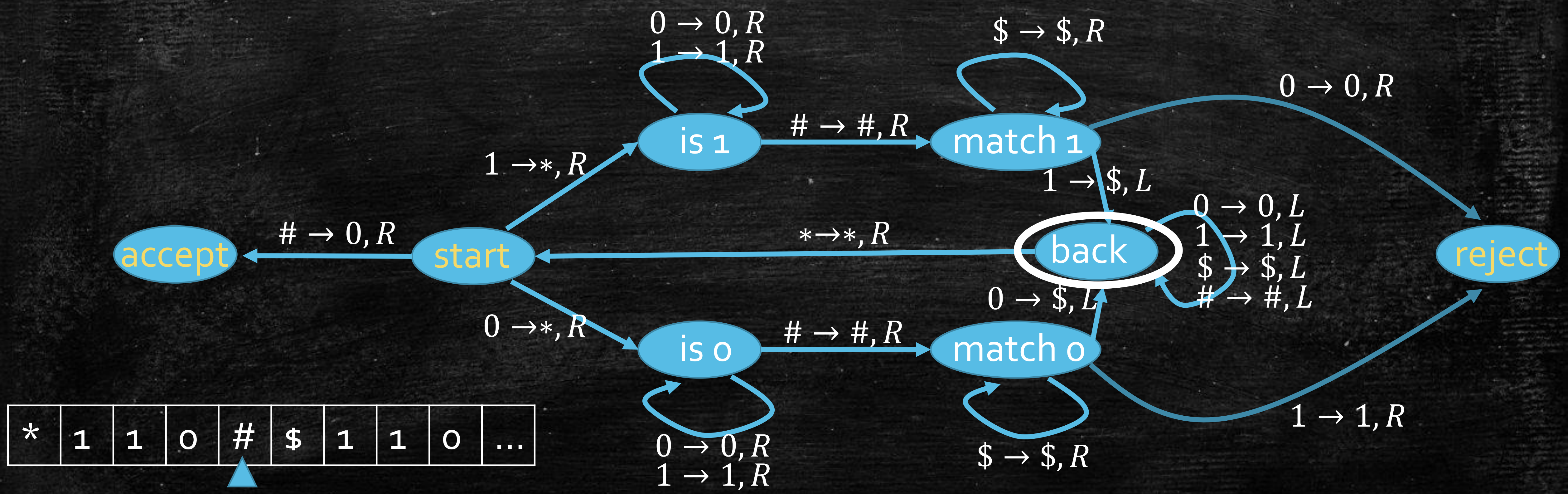
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



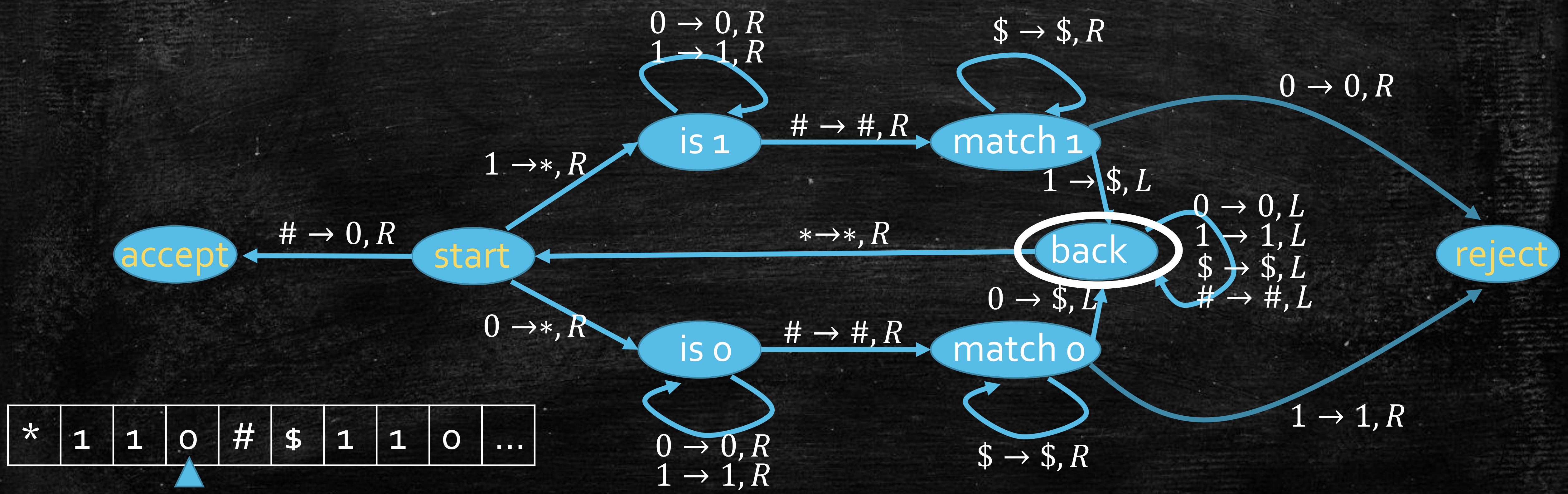
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



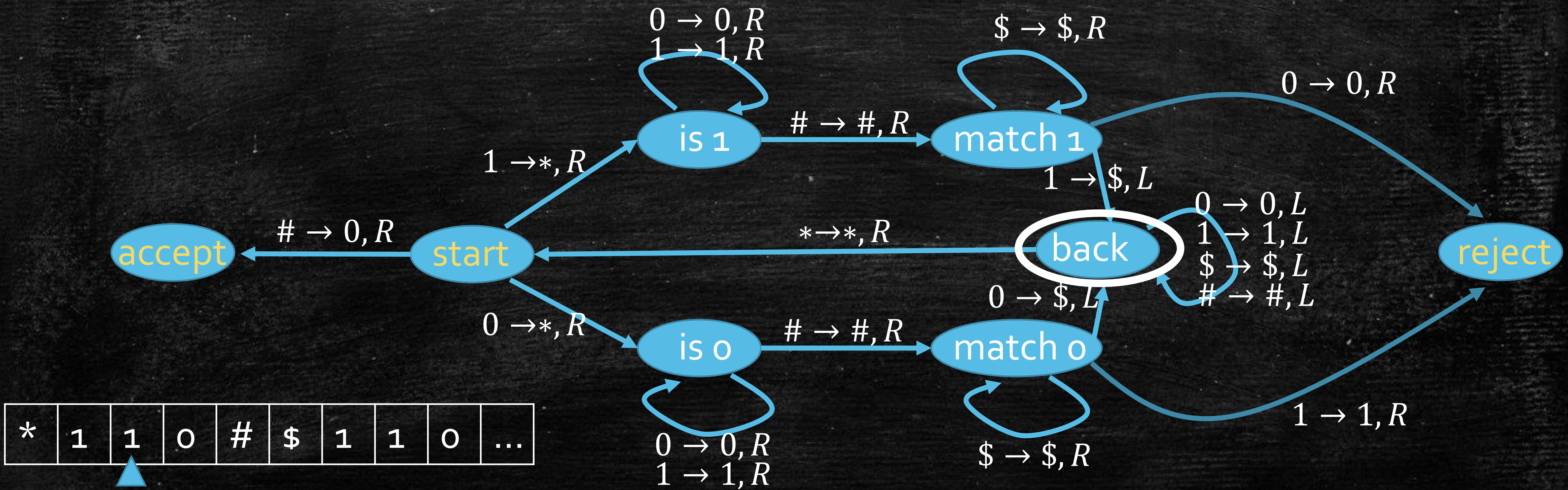
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



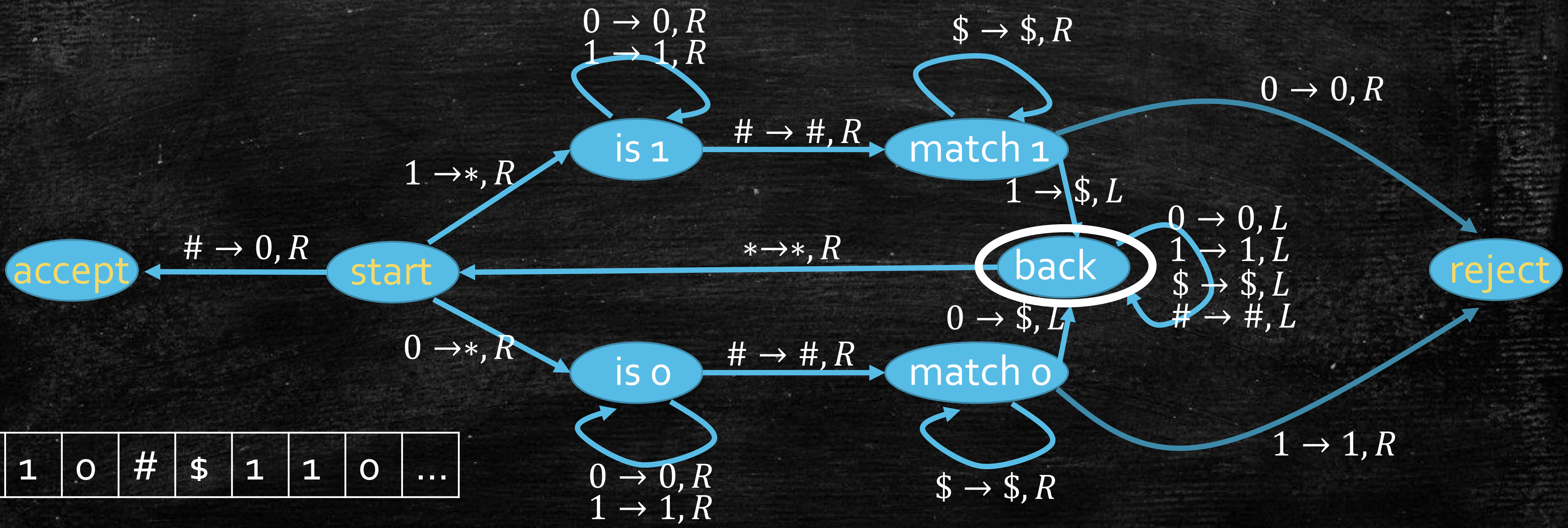
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

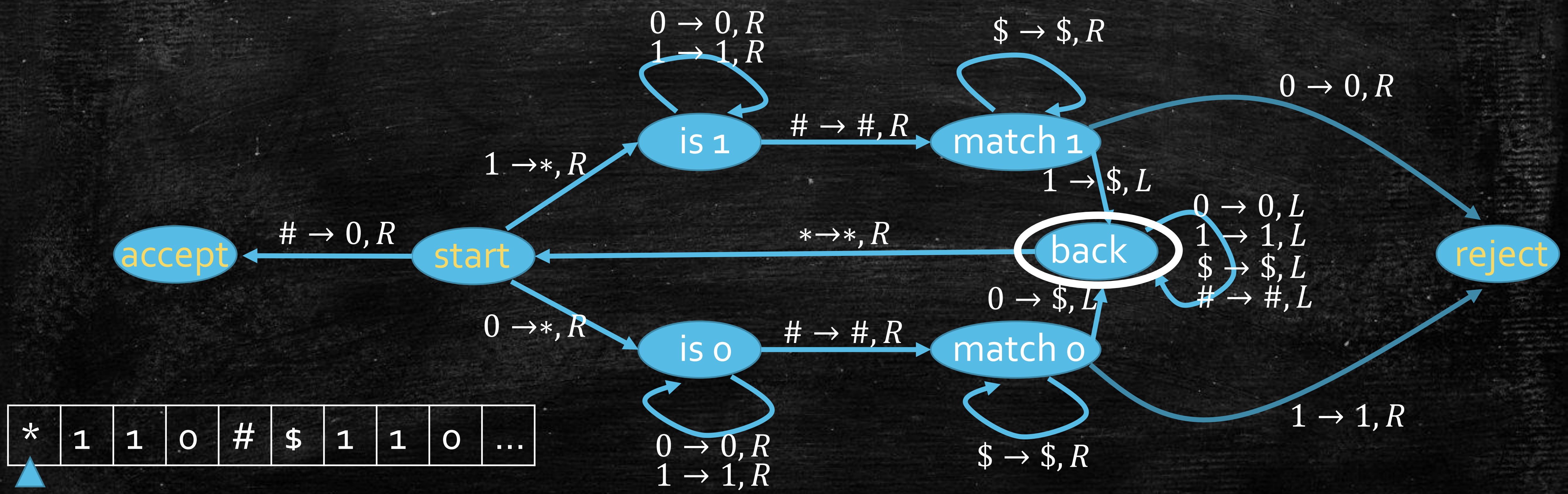
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical




| * | 1 | 1 | 0 | # | $ | 1 | 1 | 0 | ... |
|---|---|---|---|---|---|---|---|---|---|

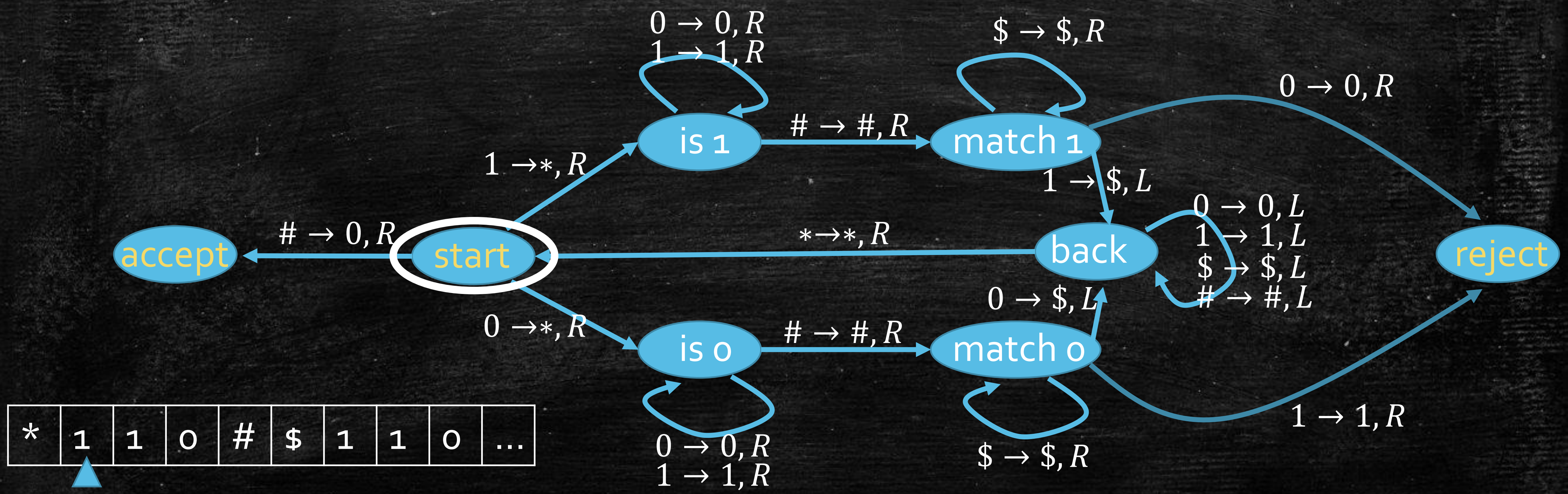
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



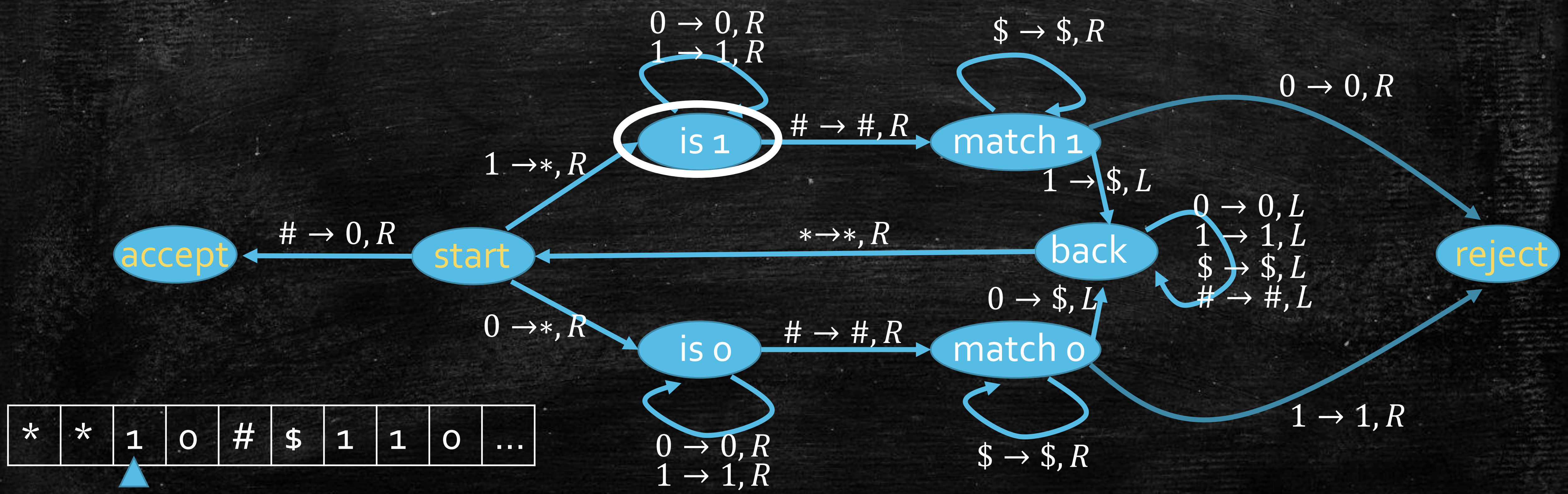
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



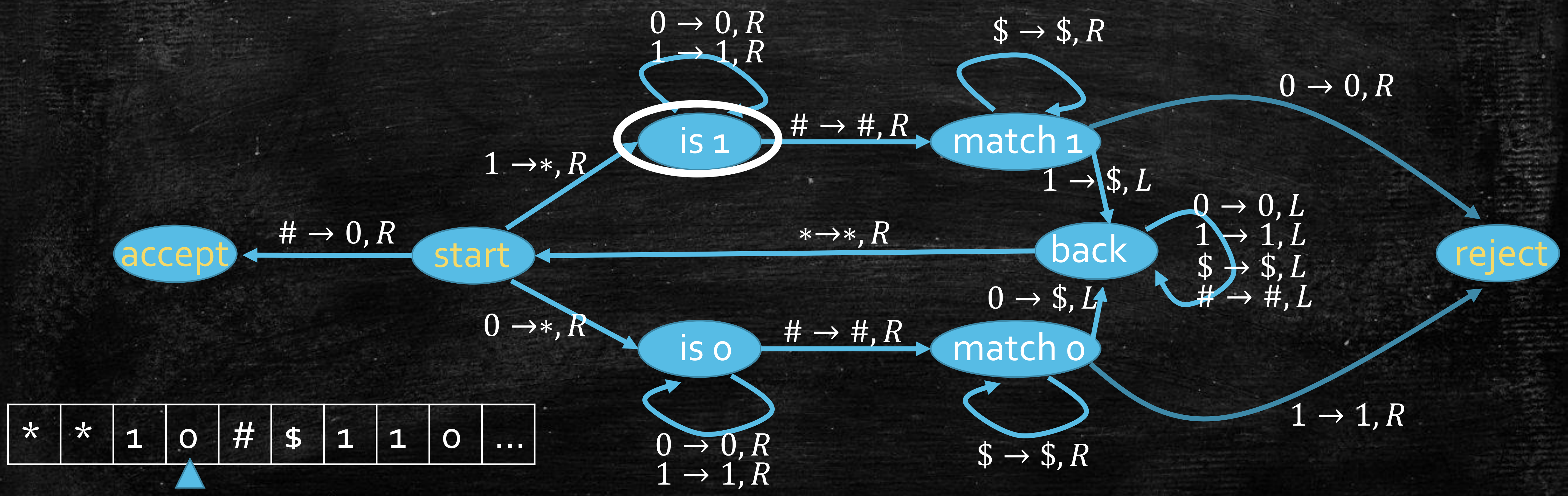
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



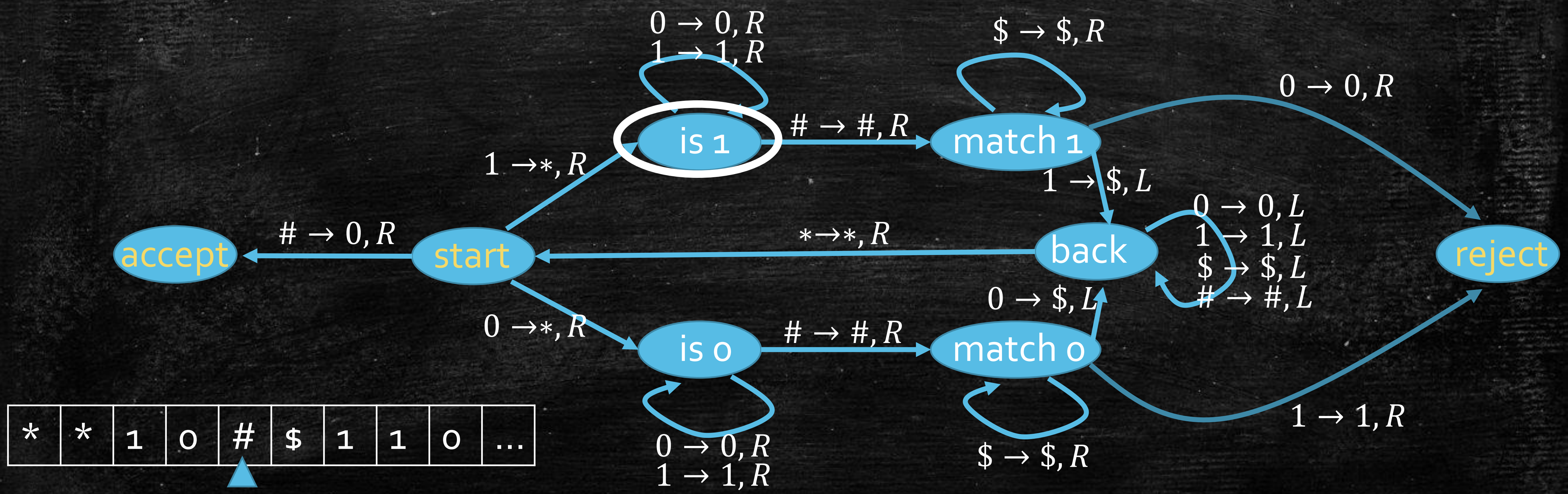
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



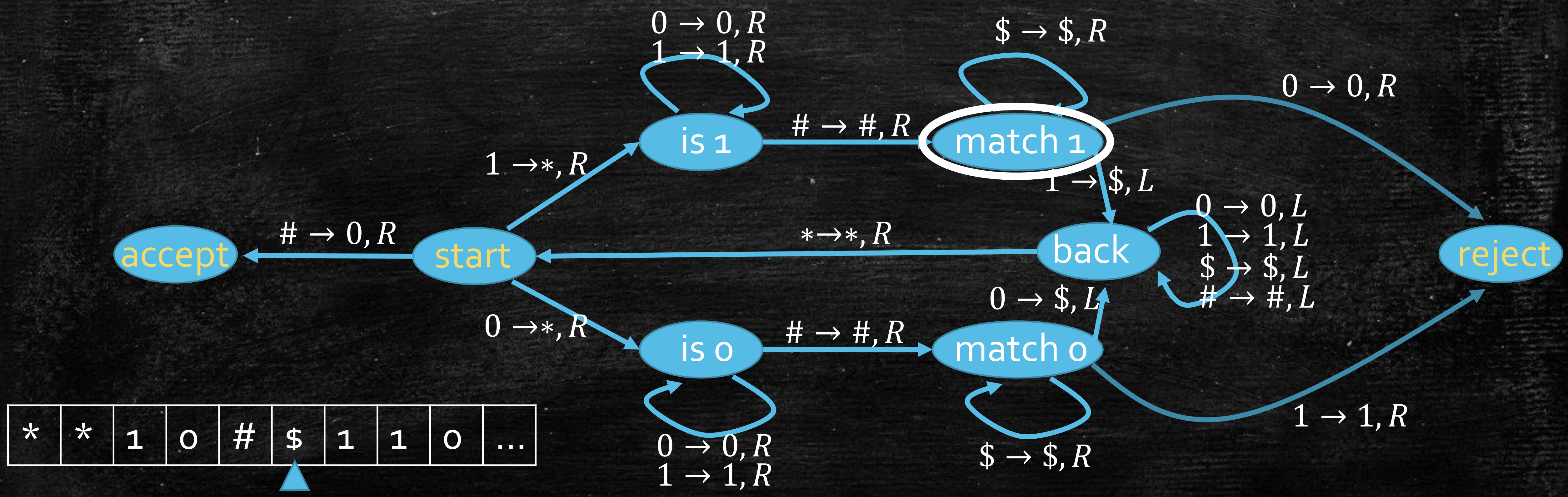
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



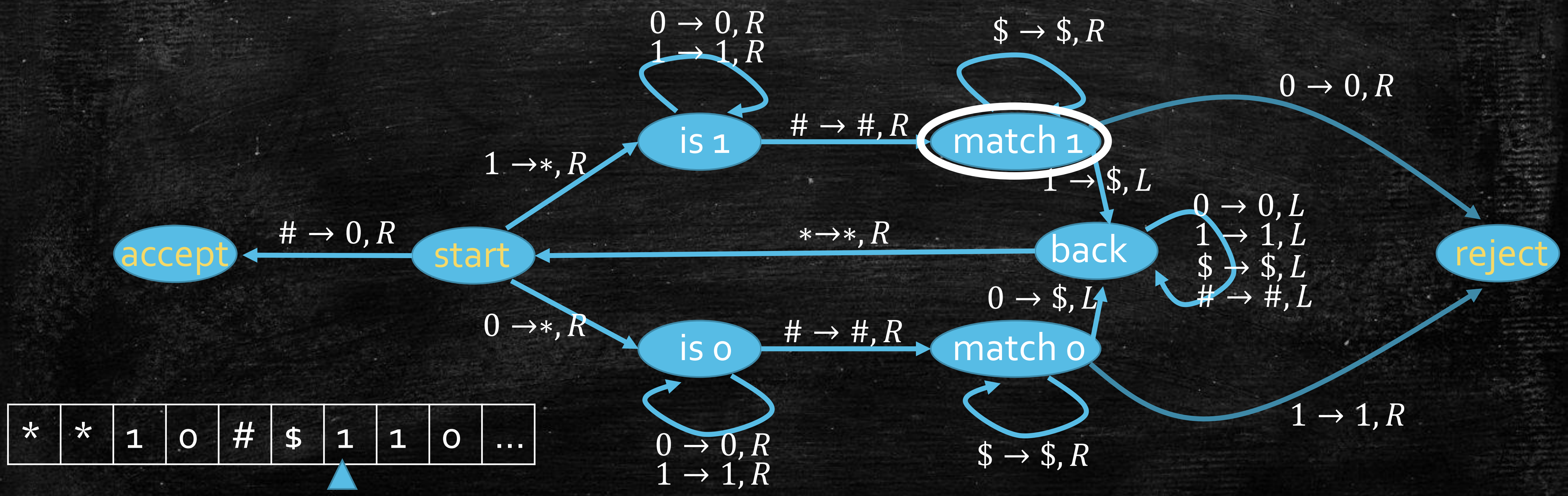
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



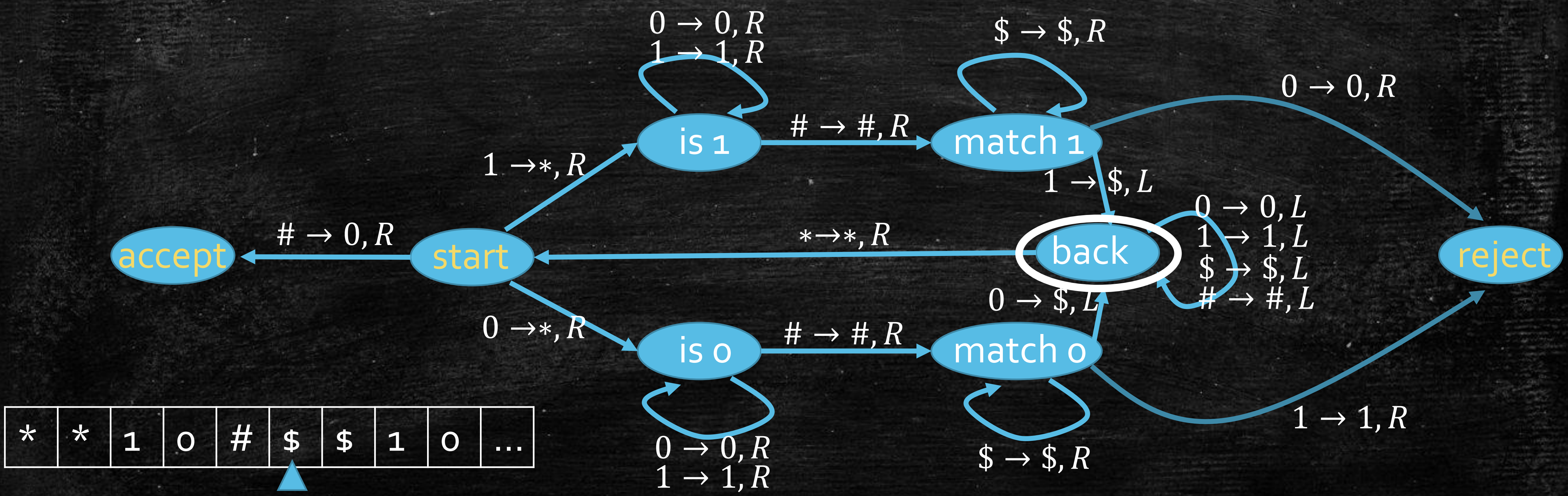
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



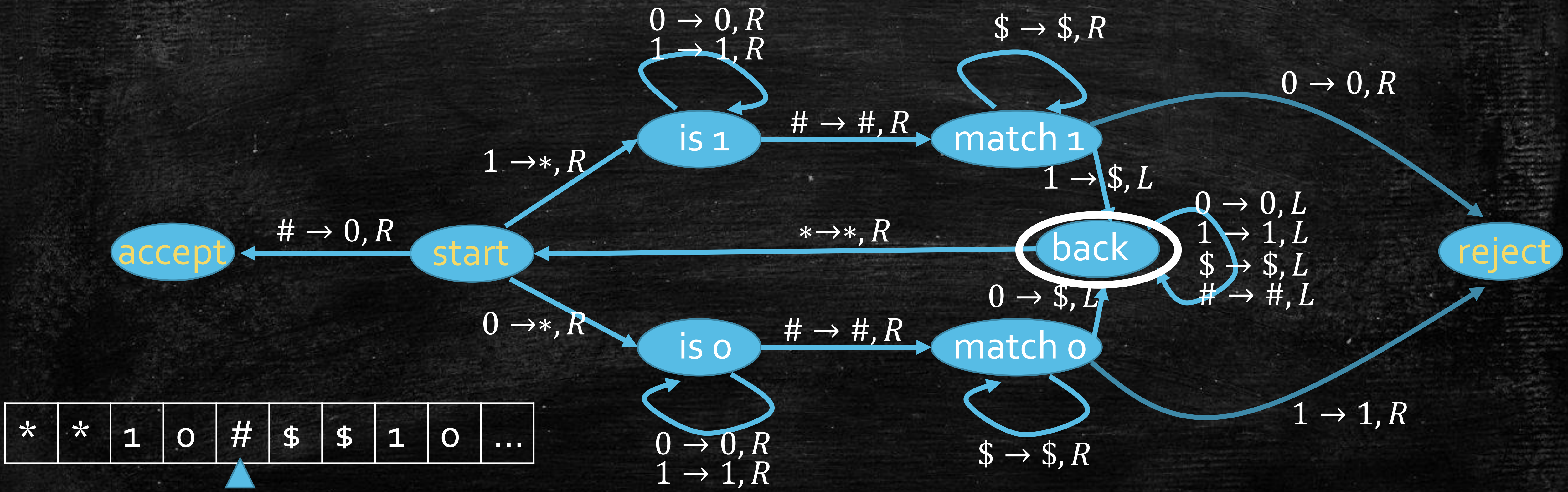
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



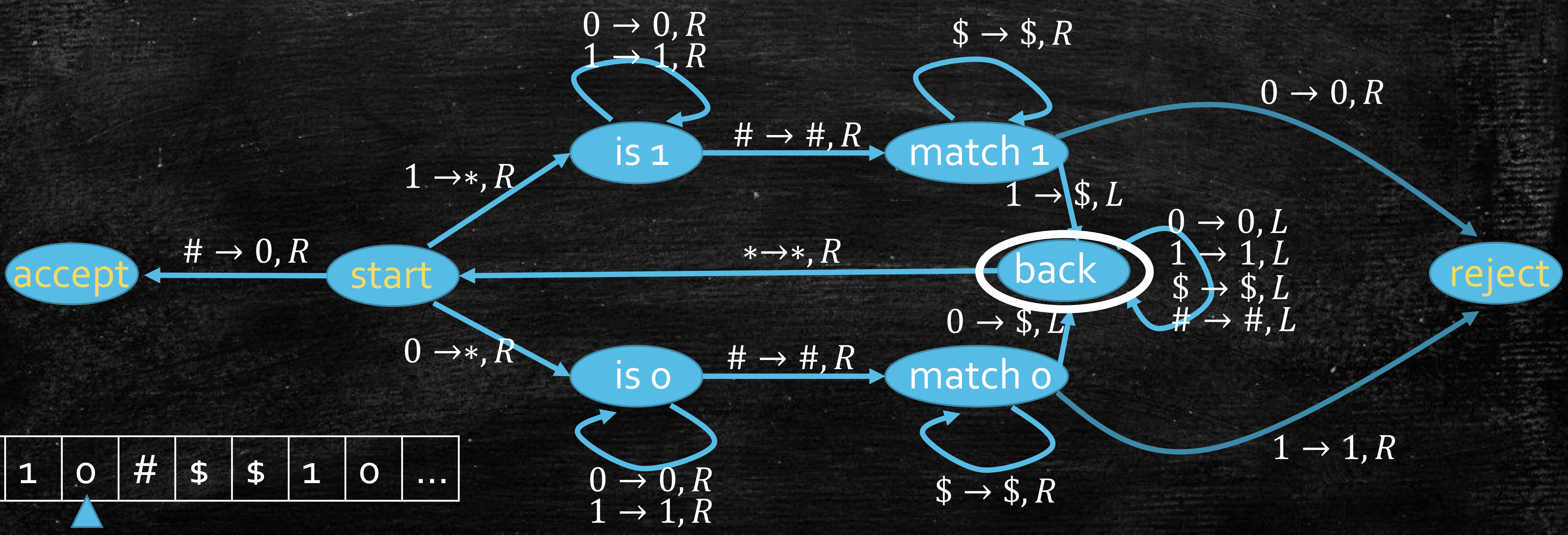
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

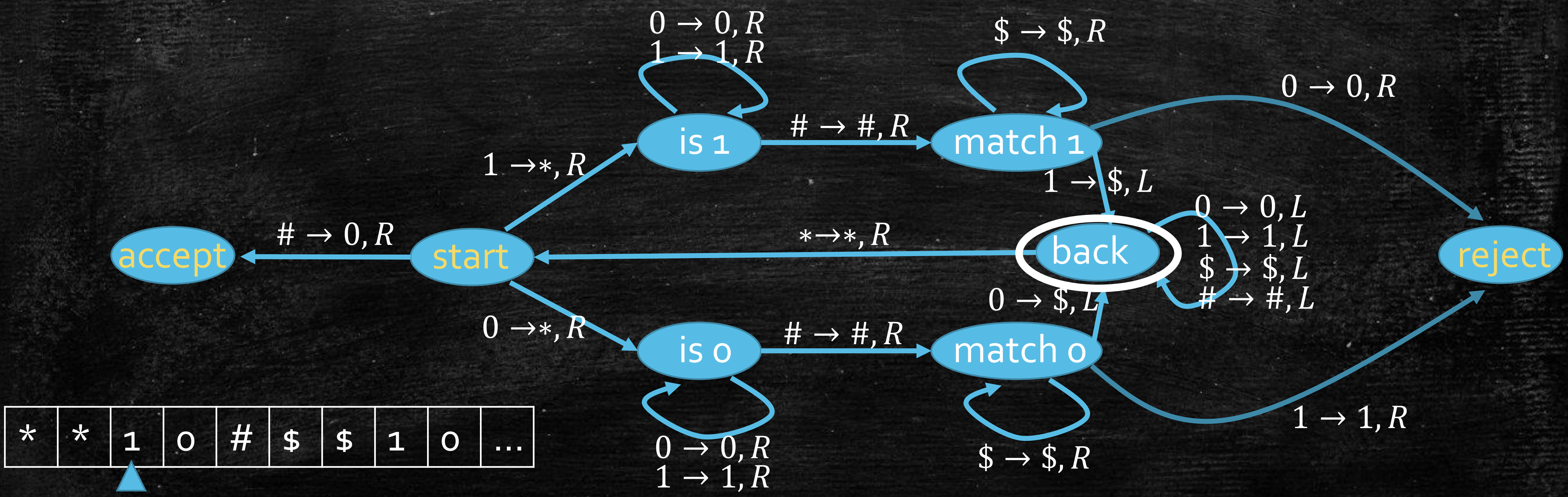
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical




| * | * | 1 | 0 | # | \$ | \$ | 1 | 0 | … |
|---|---|---|---|---|---|---|---|---|---|

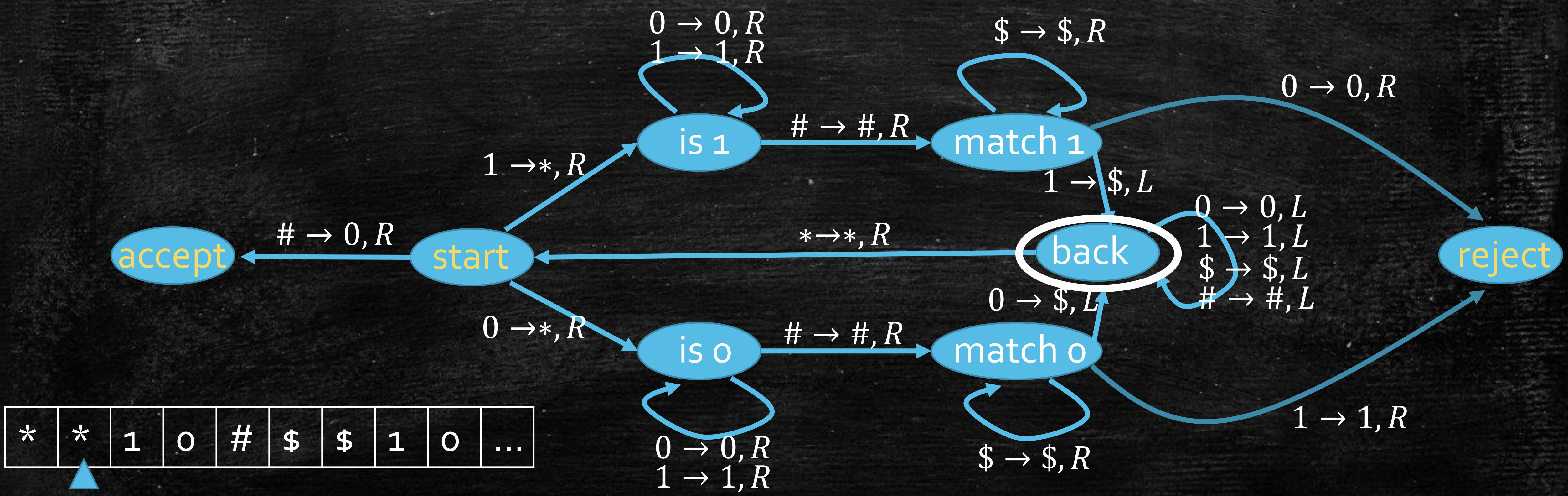
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



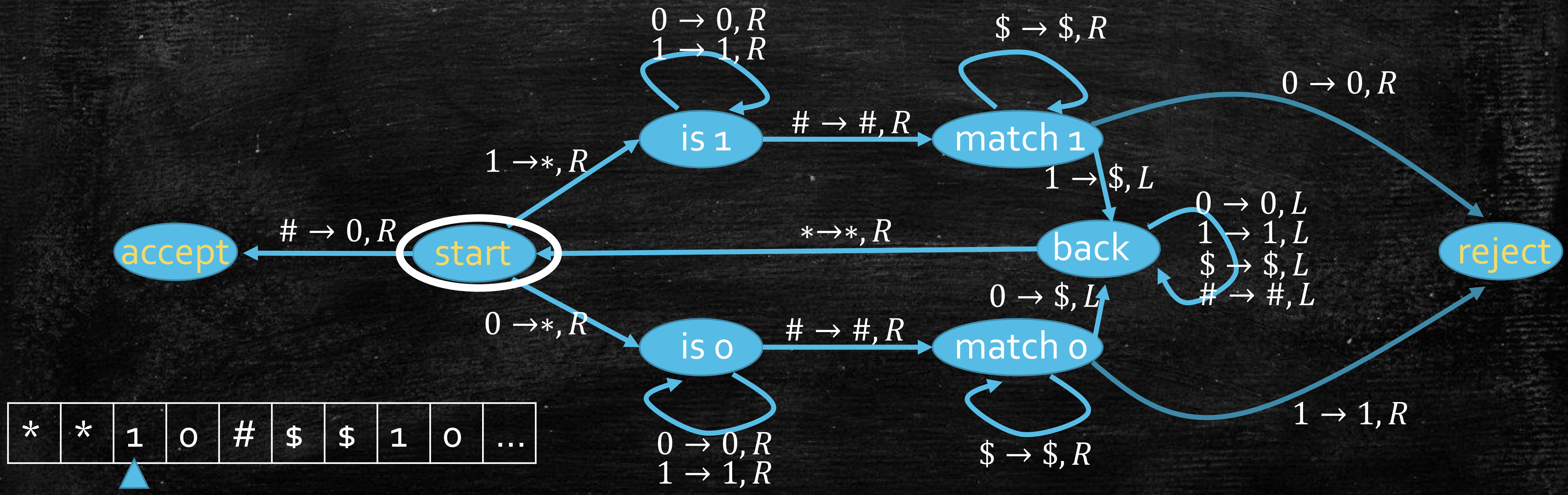
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



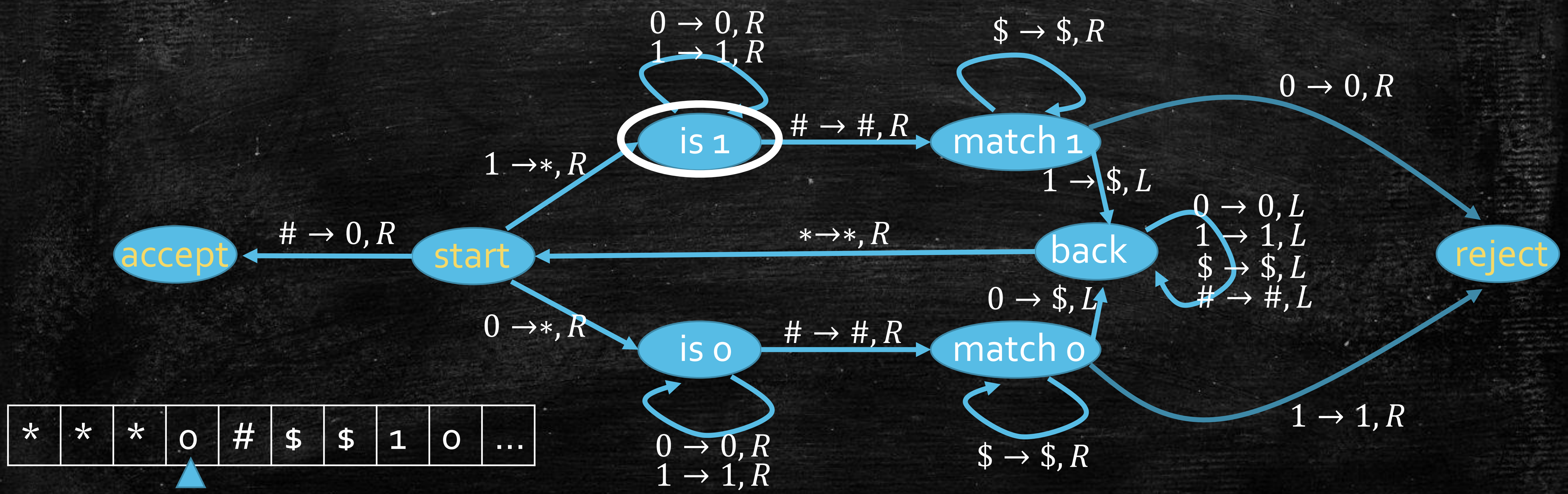
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



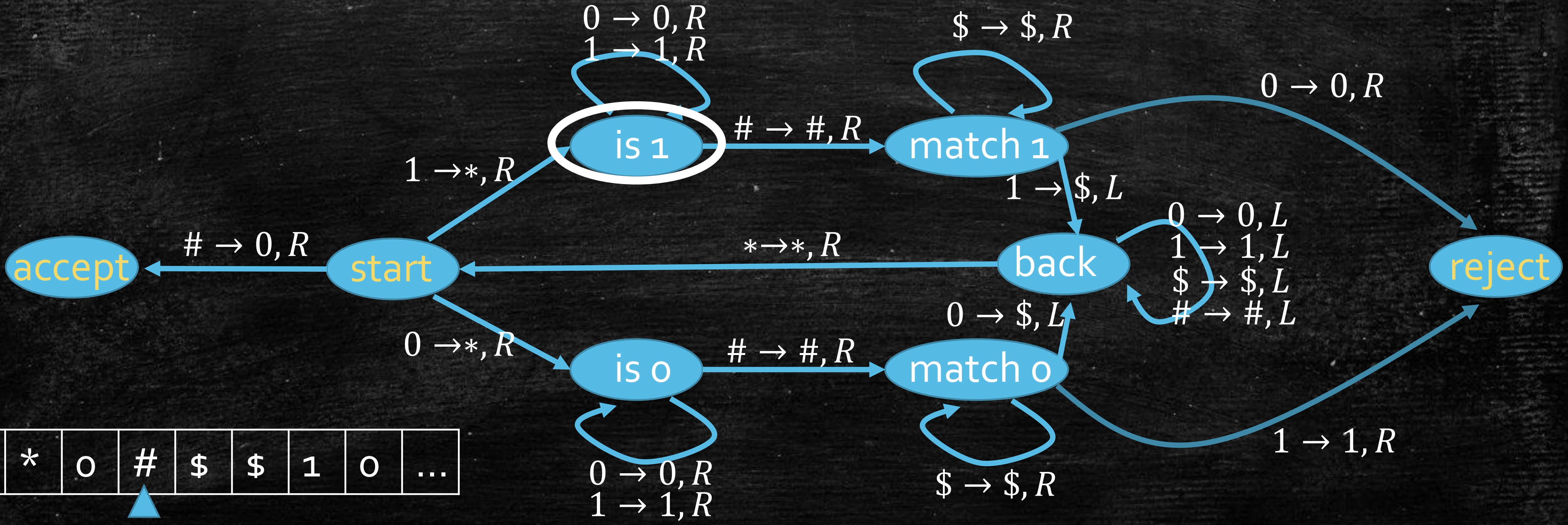
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

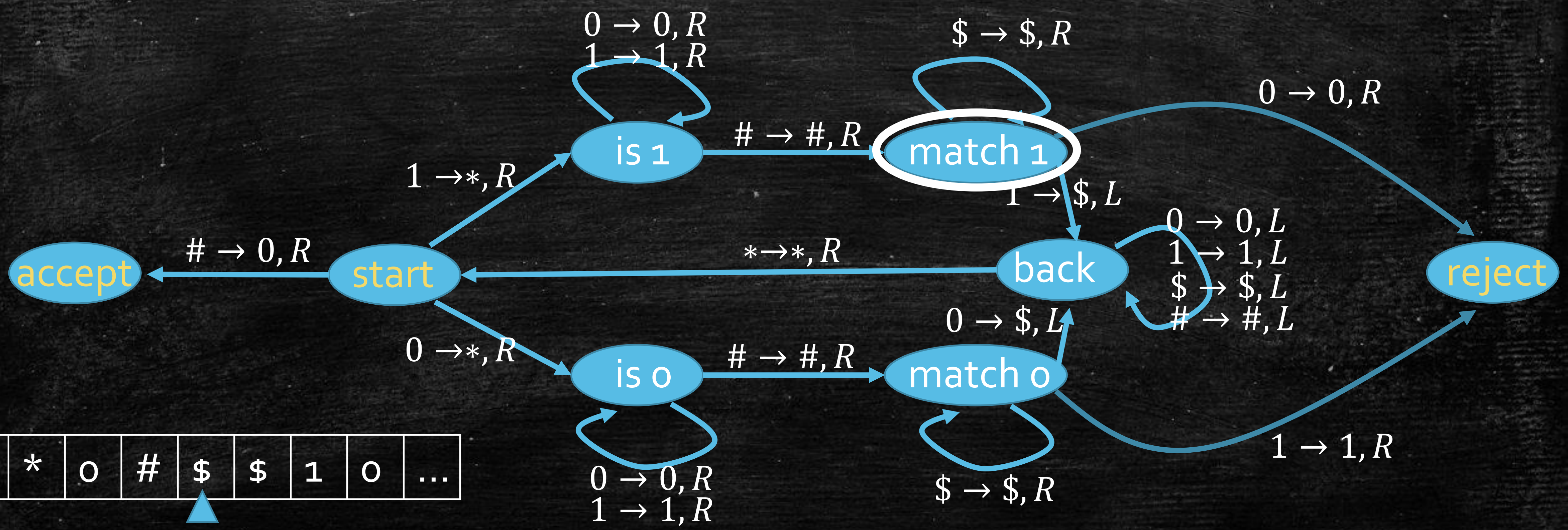
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| * | * | * | 0 | # | \$ | \$ | 1 | 0 | ... |

# TM Example: Check if two strings are identical

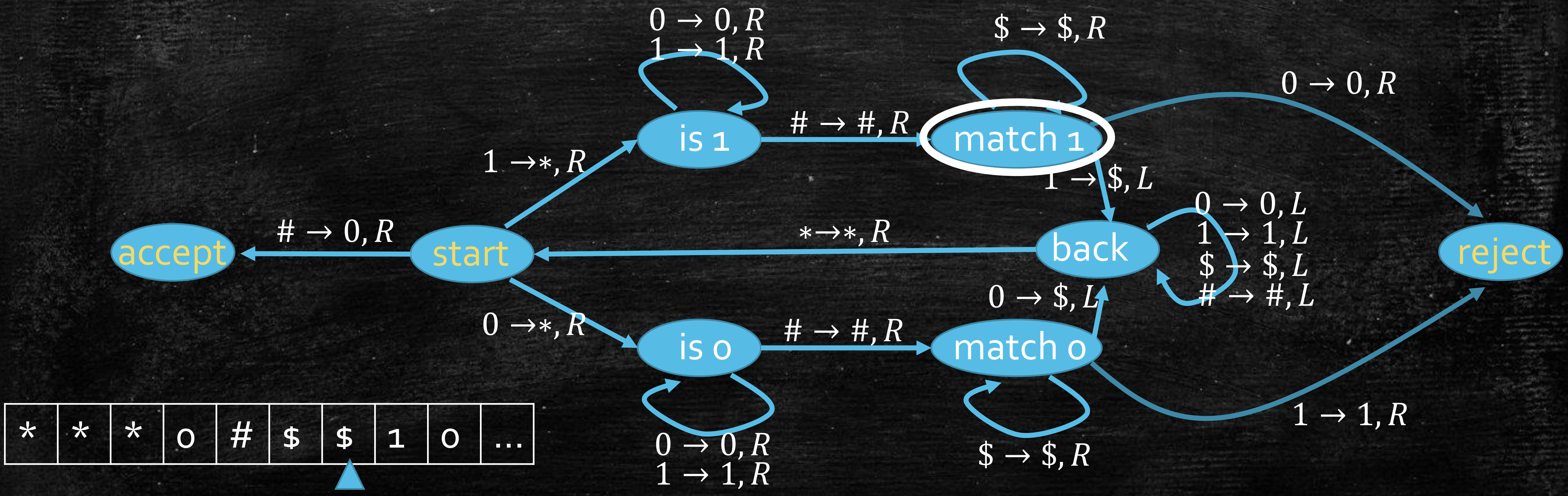
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical




| * | * | * | 0 | # | $ | $ | 1 | 0 | … |
|---|---|---|---|---|---|---|---|---|---|

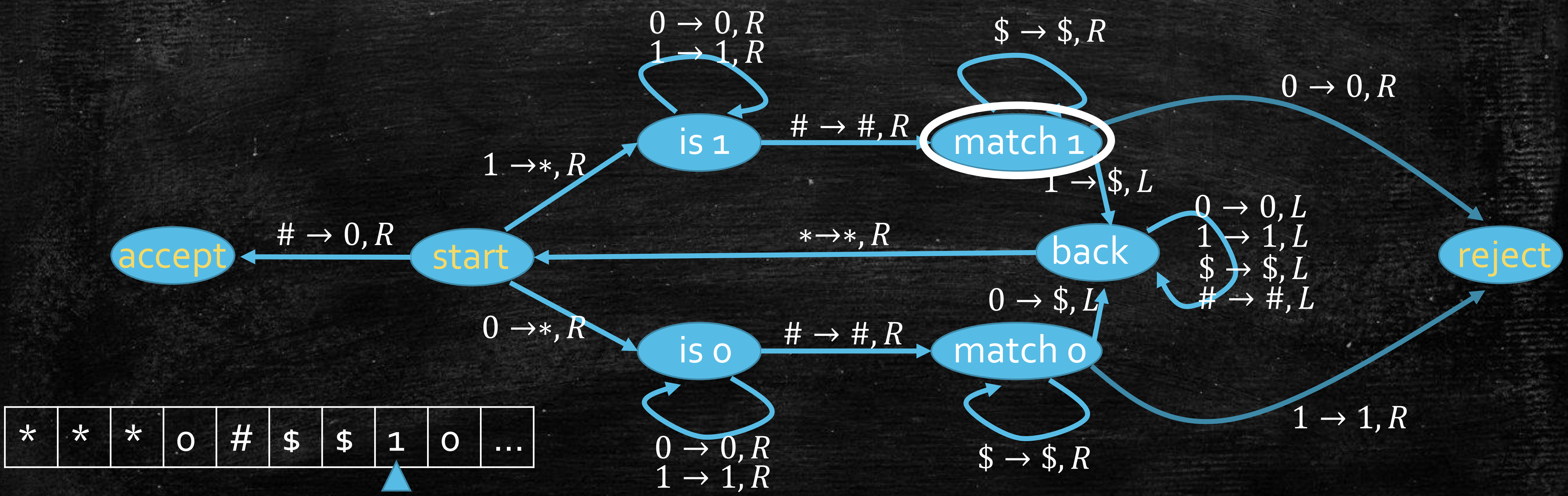
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



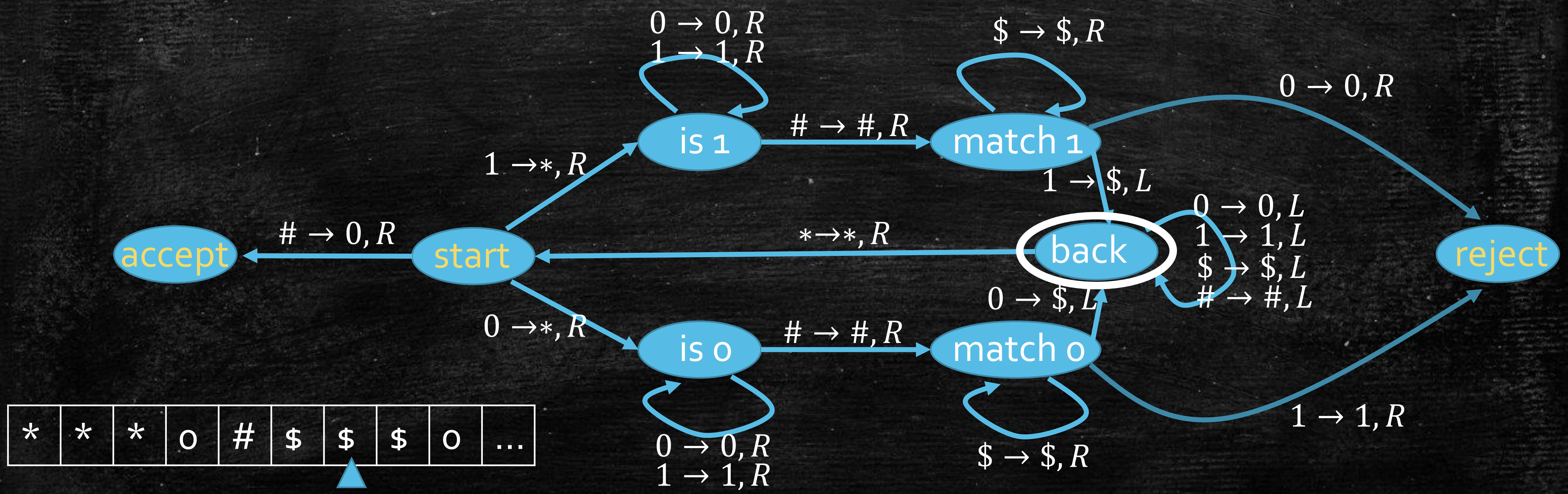
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



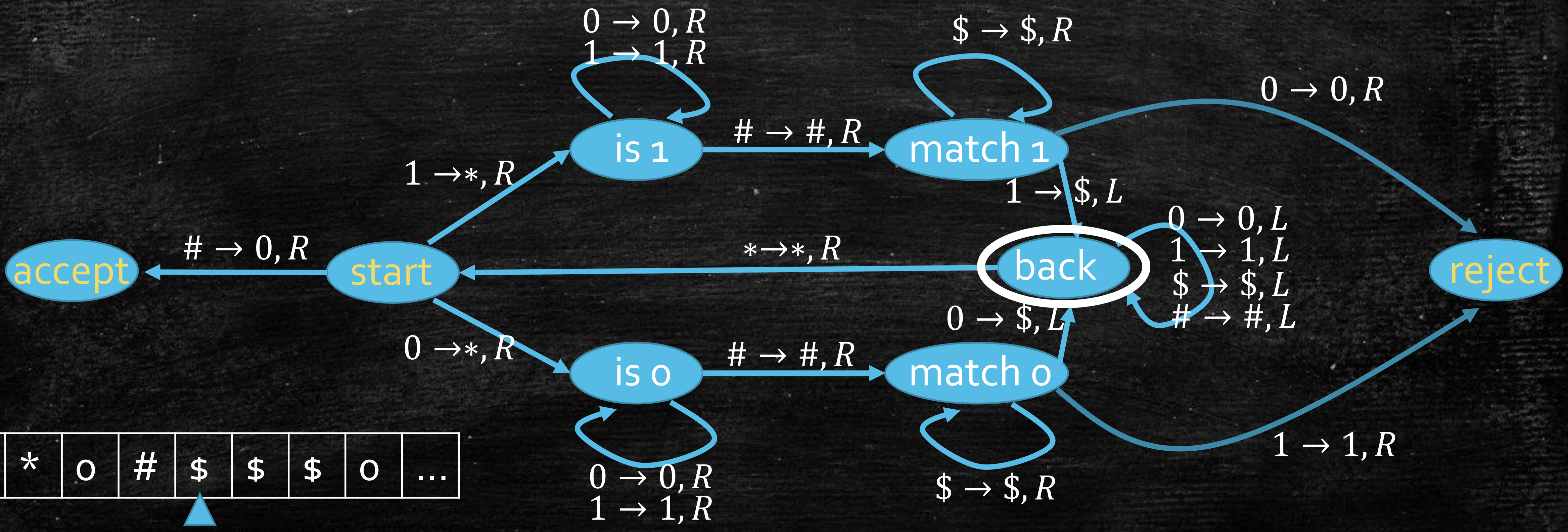
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

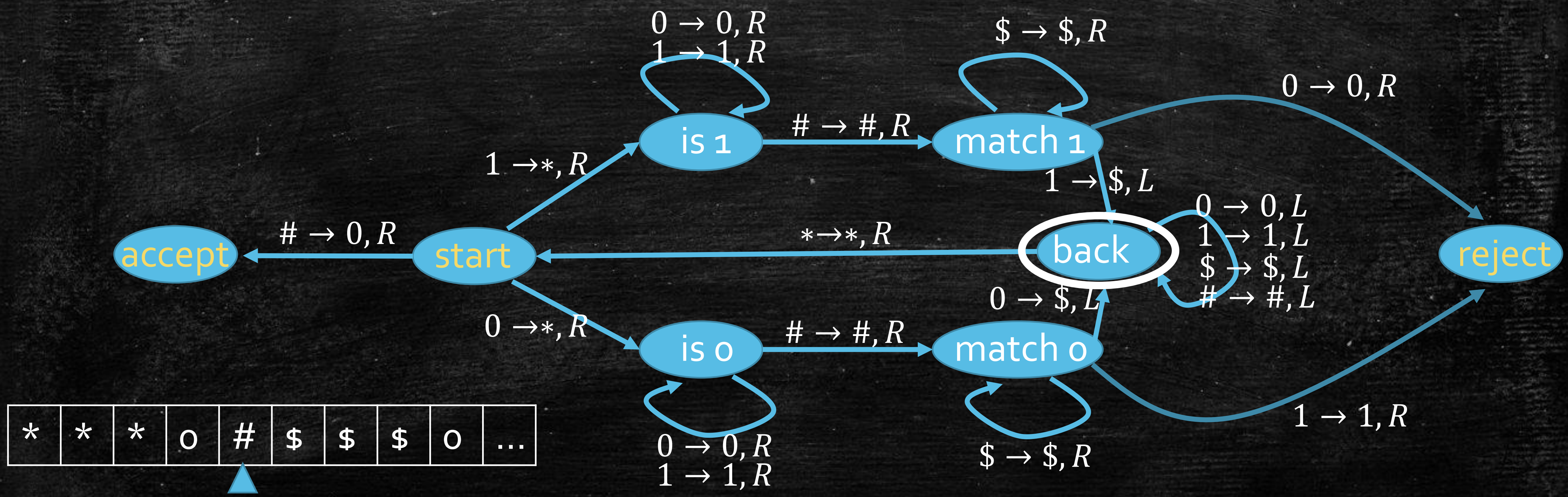
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| * | * | * | 0 | # | \$ | \$ | \$ | 0 | ... |

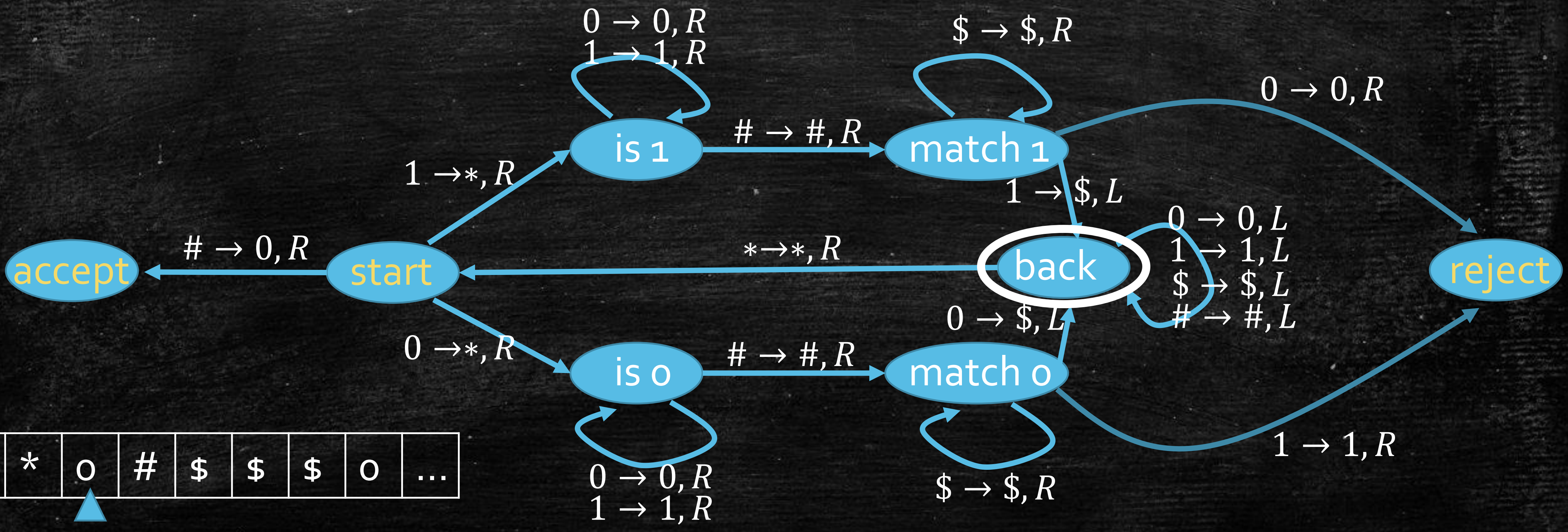
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

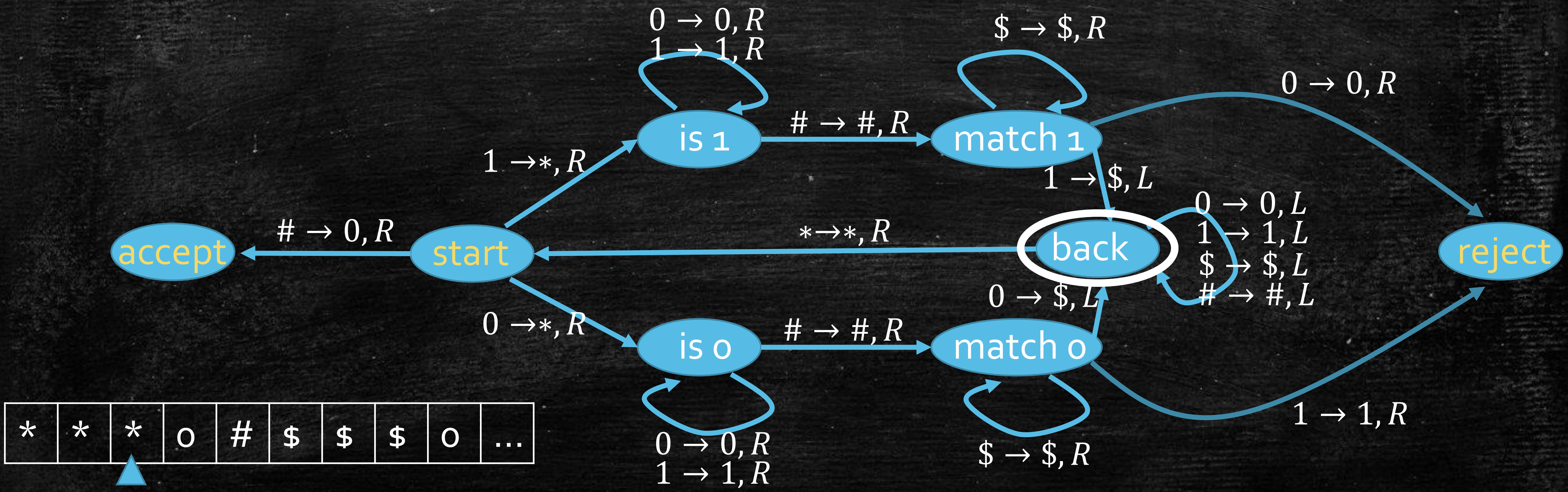
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| * | * | * | 0 | # | $ | $ | $ | 0 | … |

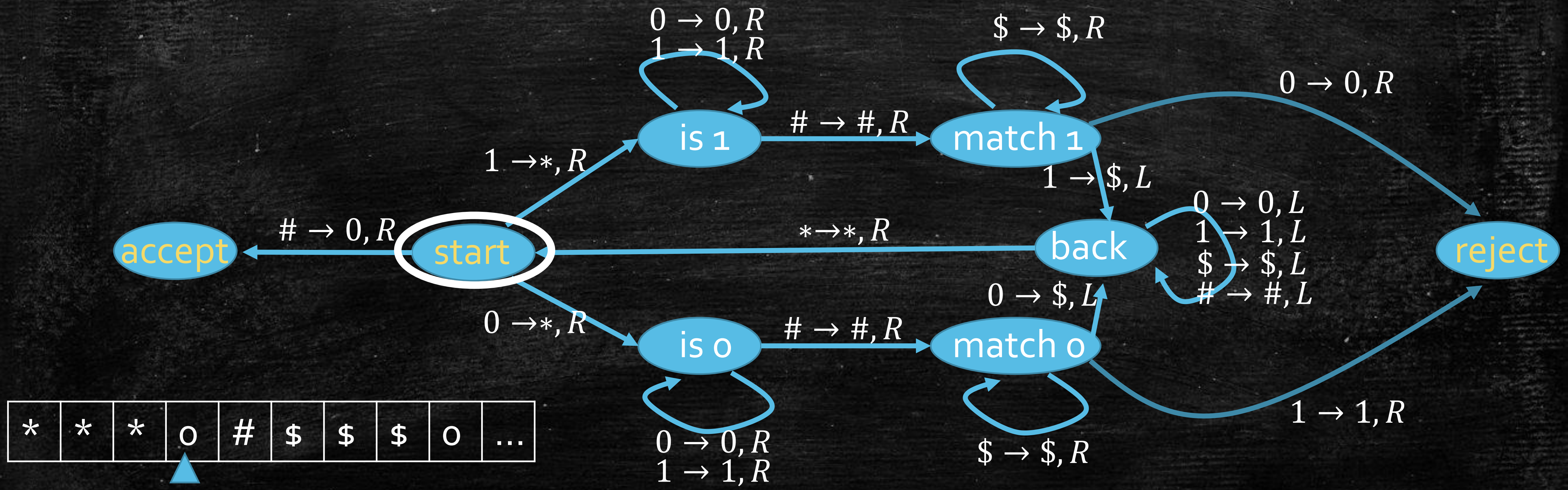
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



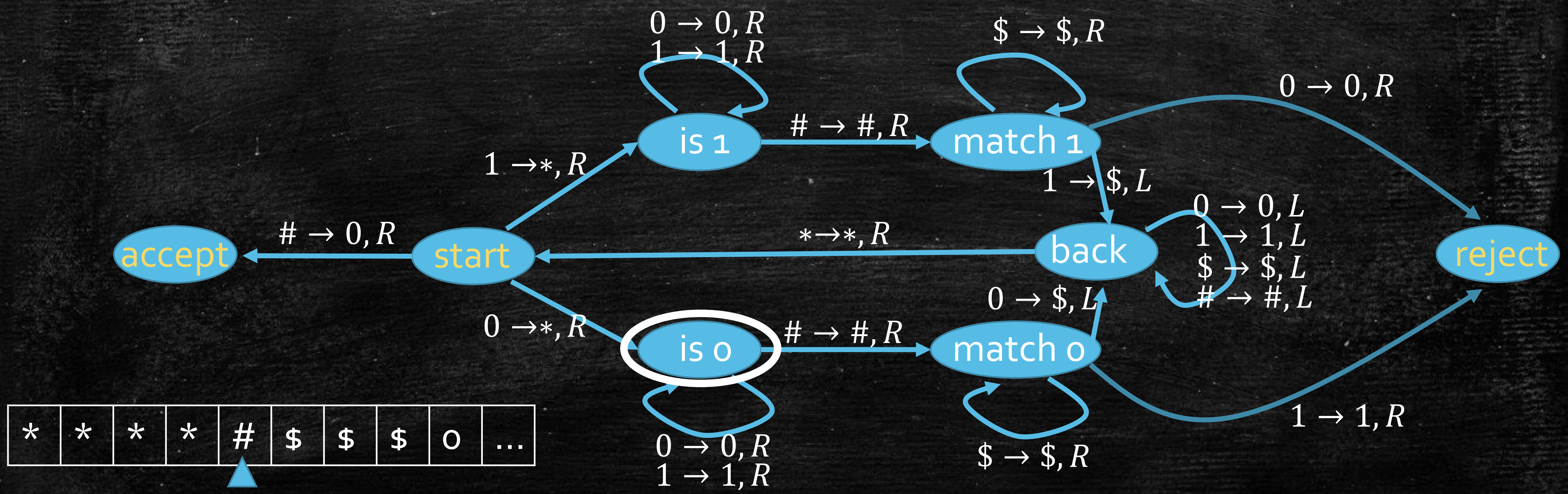
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

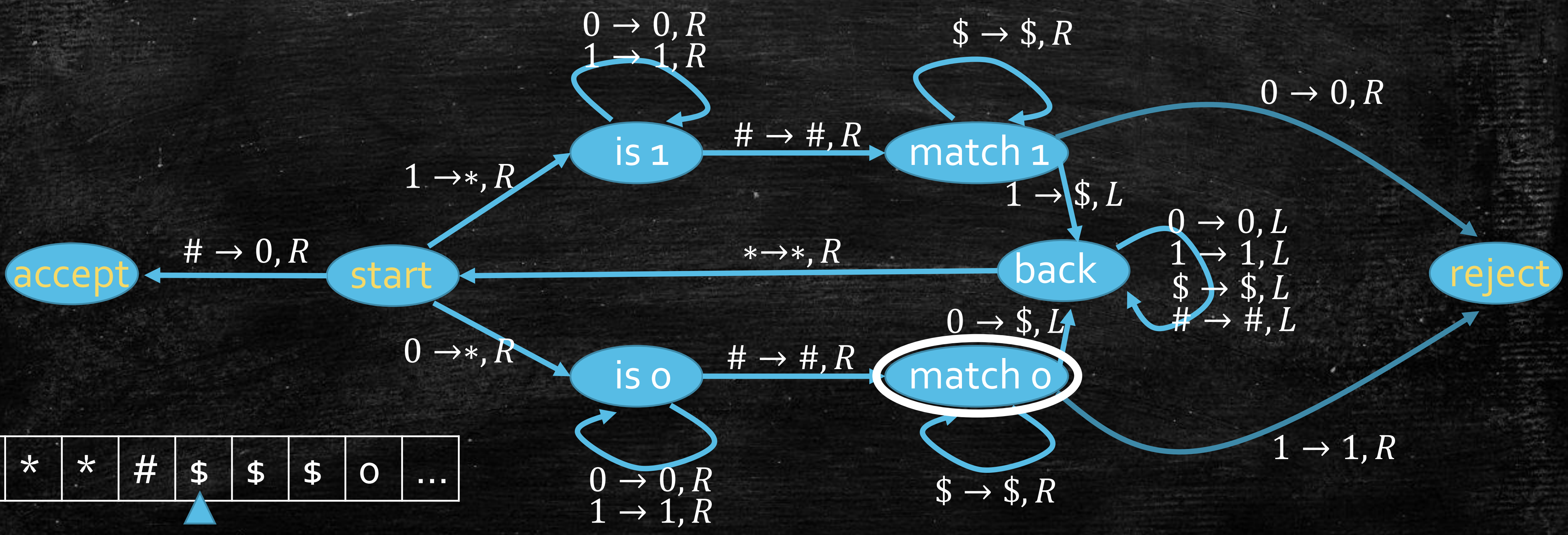
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet

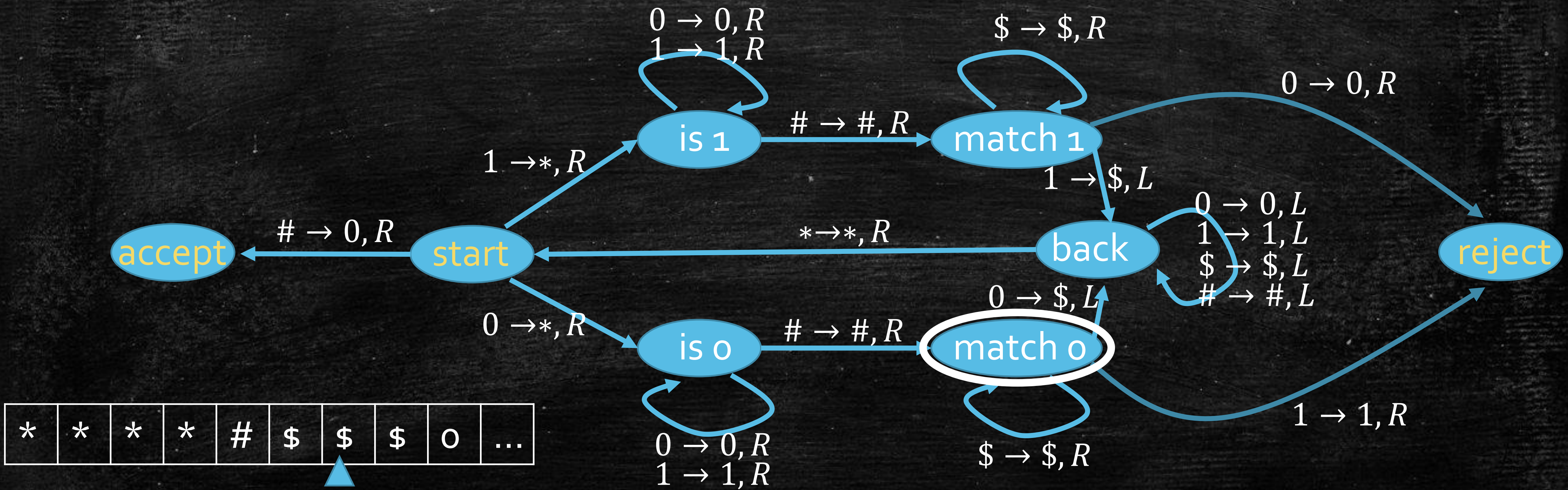
- Decide if the binary strings $x$ and $y$ is identical




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| * | * | * | * | # | \$ | \$ | \$ | 0 | … |

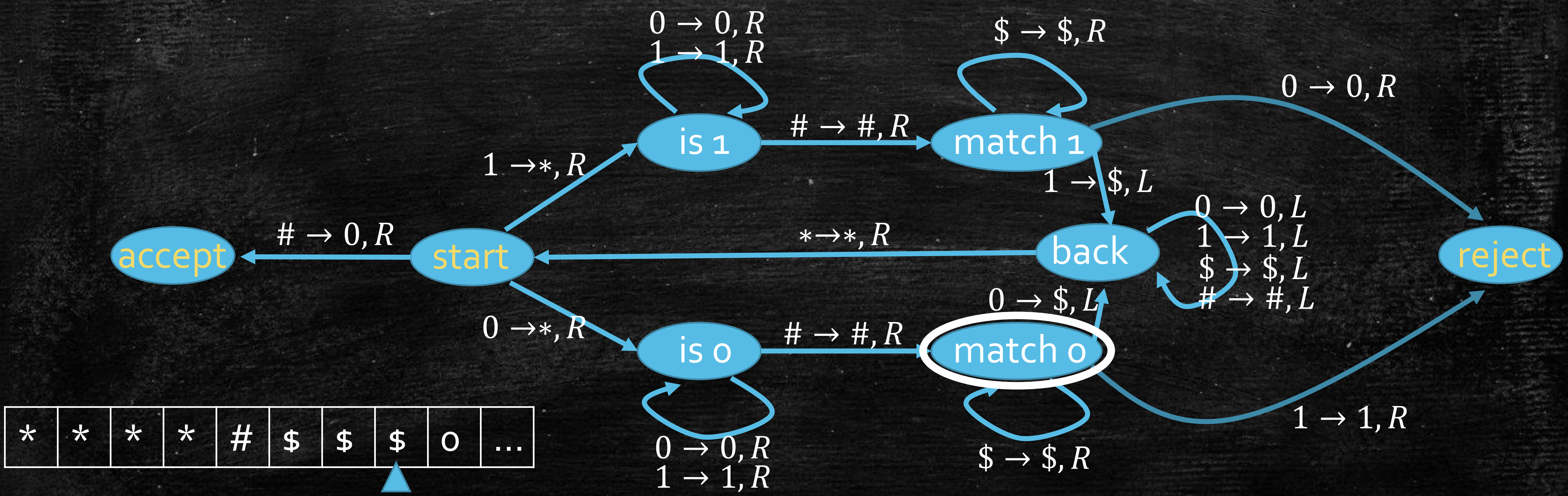
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



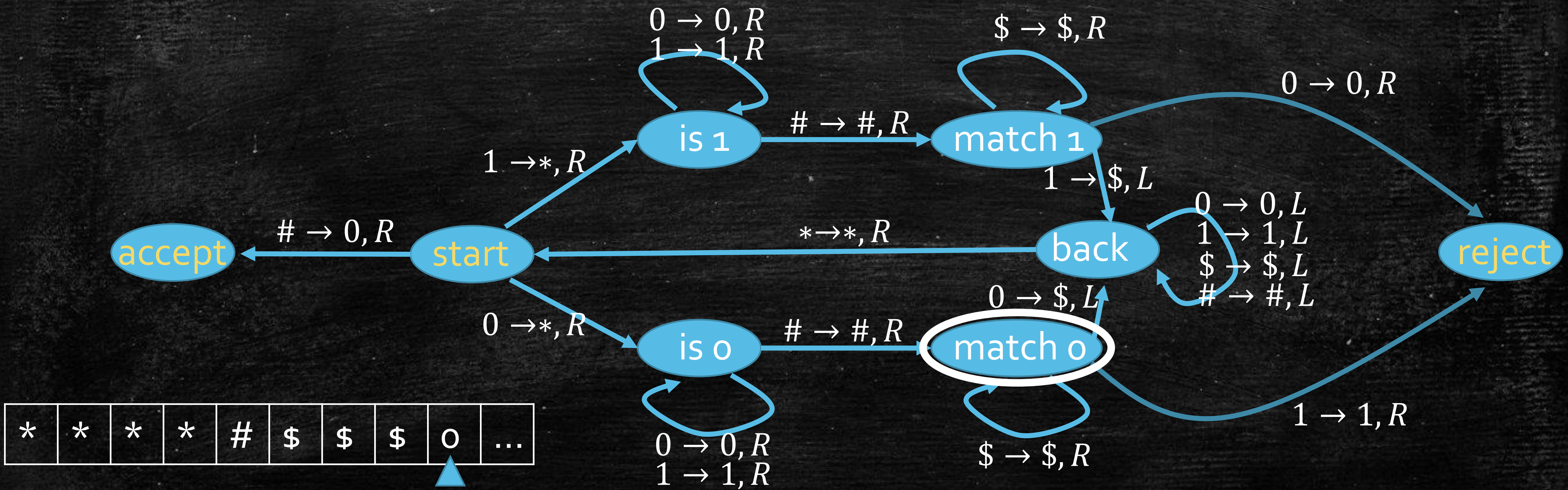
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



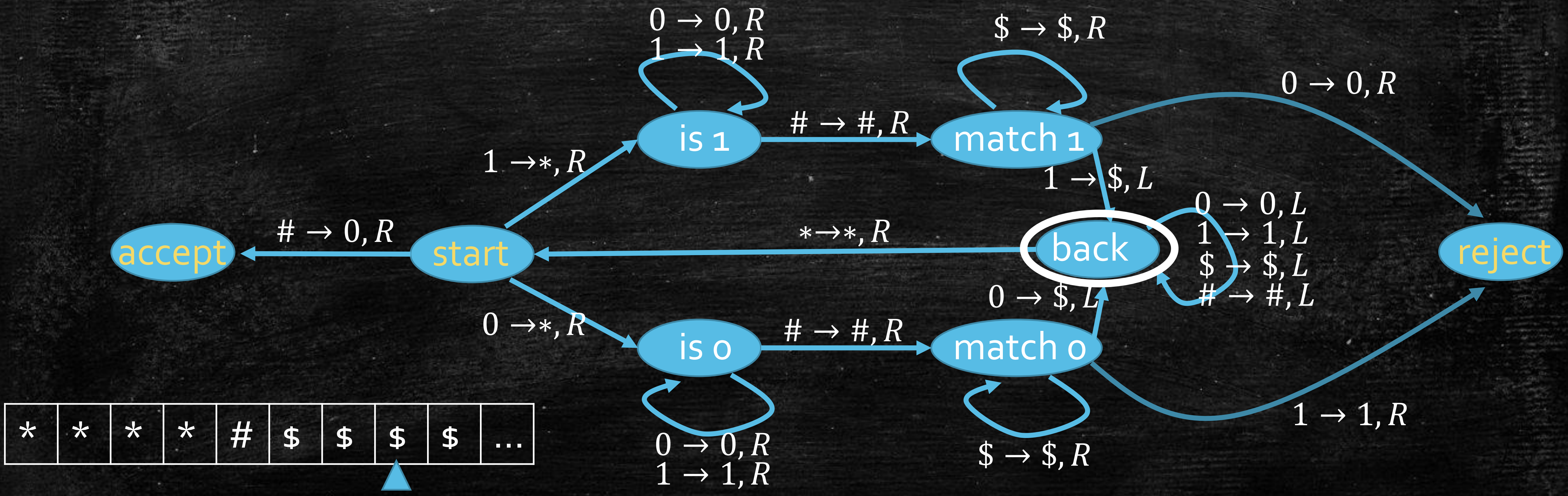
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



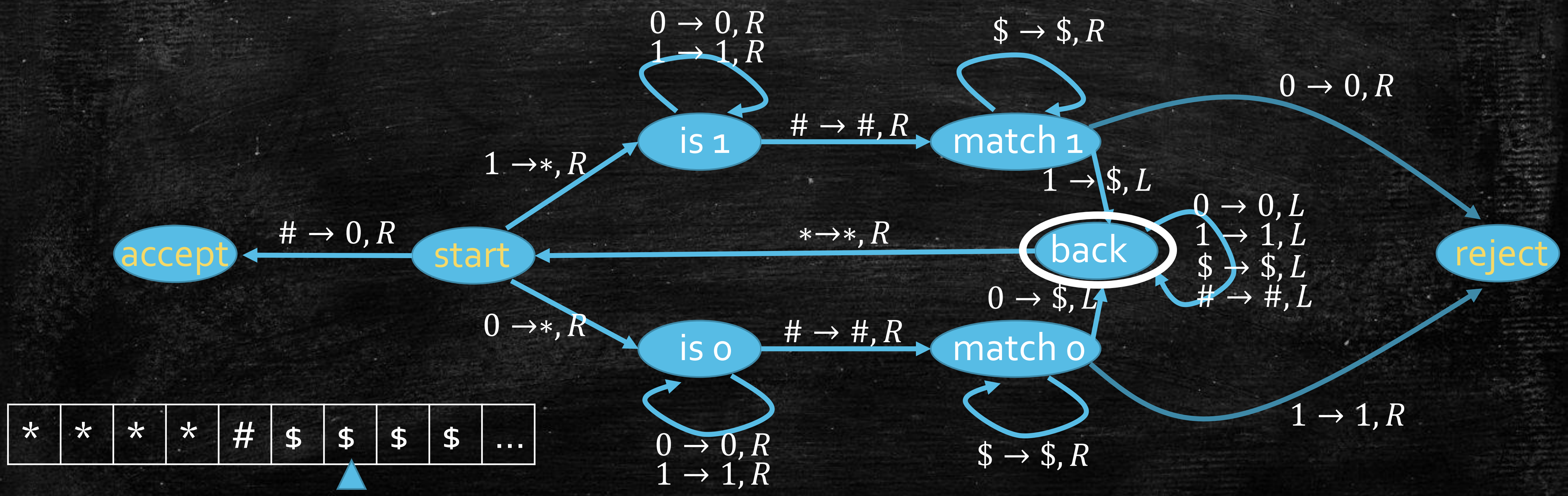
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



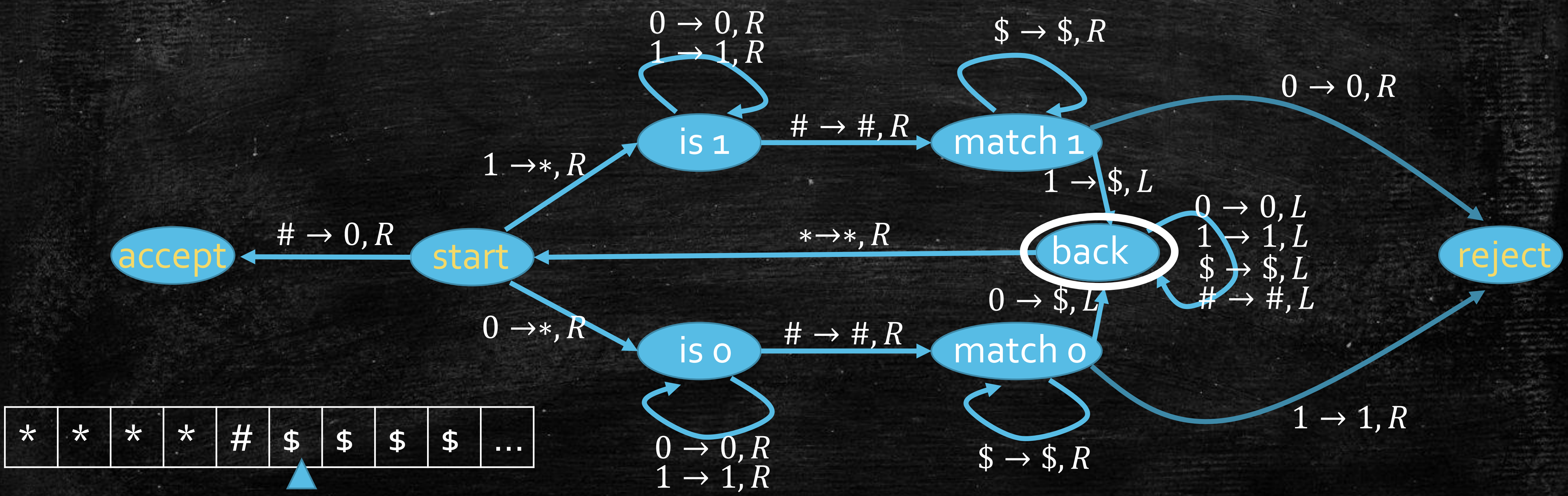
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



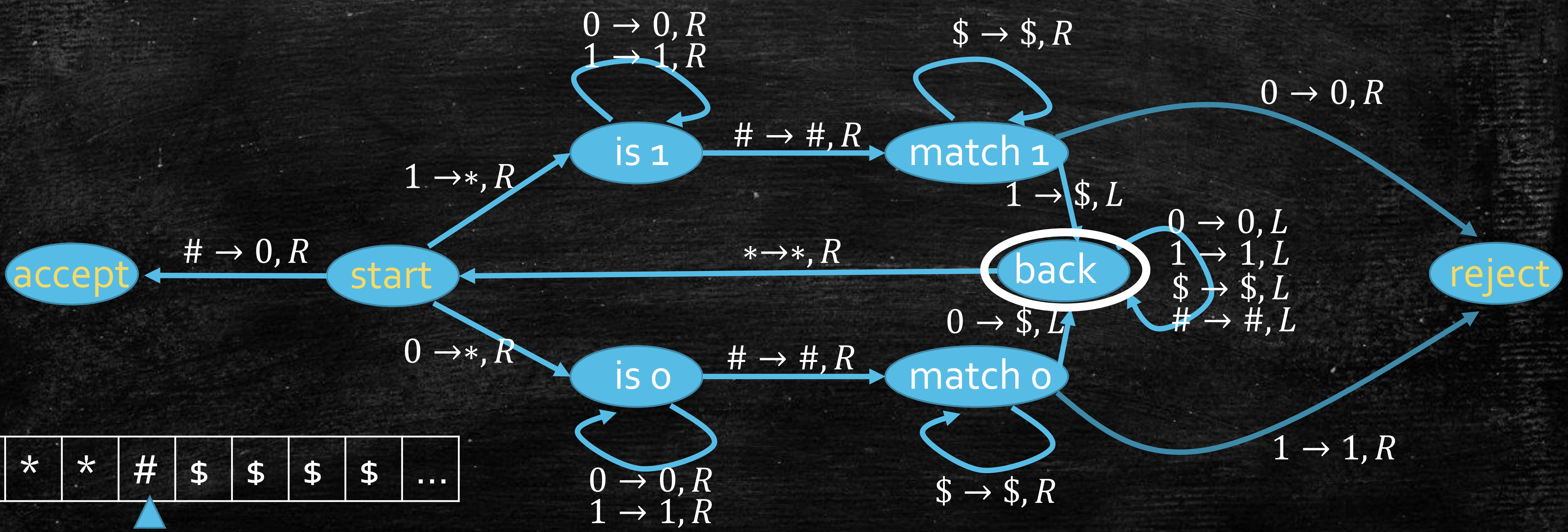
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

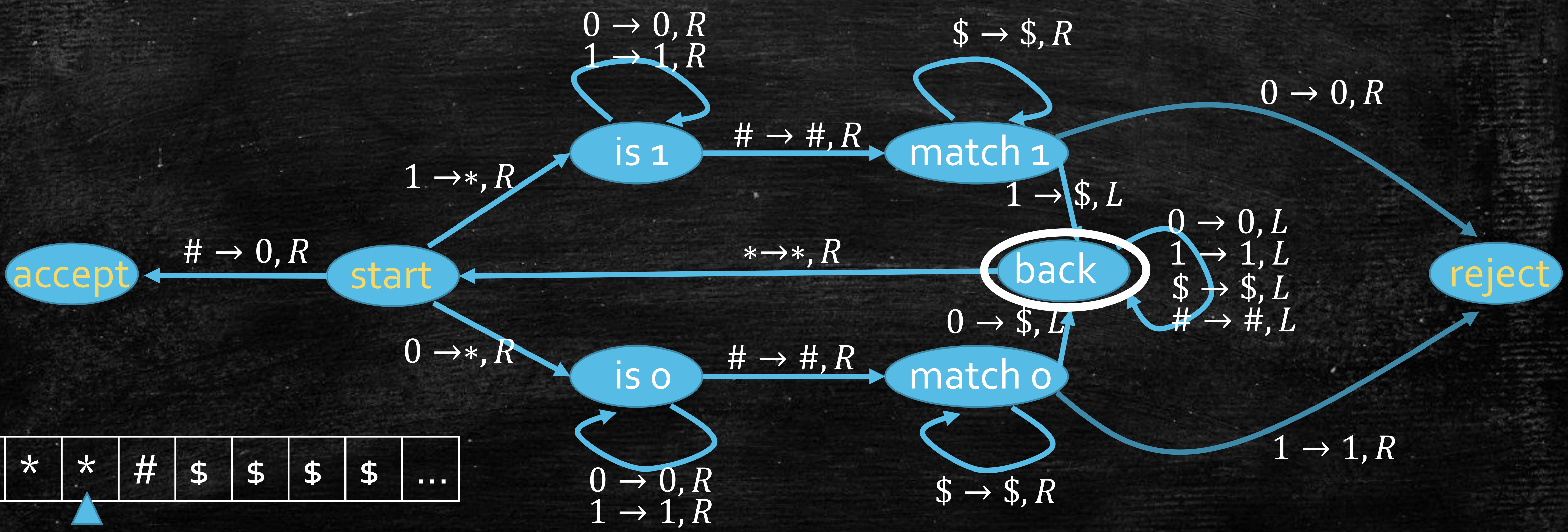
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| * | * | * | * | # | \$ | \$ | \$ | \$ | … |

# TM Example: Check if two strings are identical

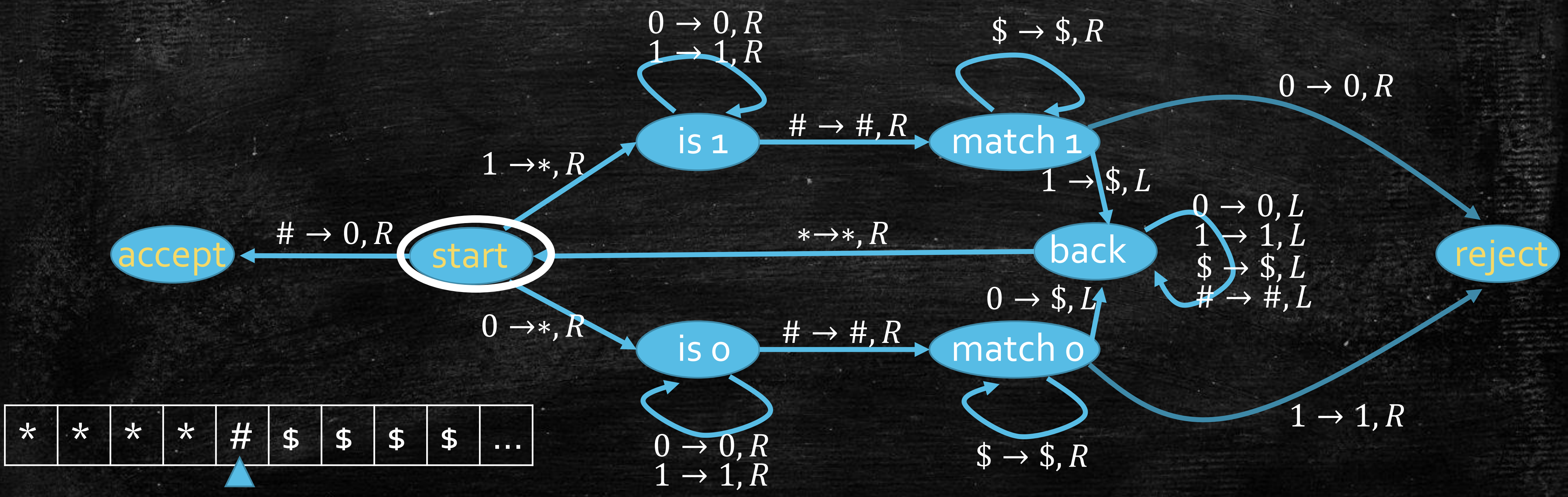
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical




| * | * | * | * | # | \$ | \$ | \$ | \$ | … |
|---|---|---|---|---|---|---|---|---|---|

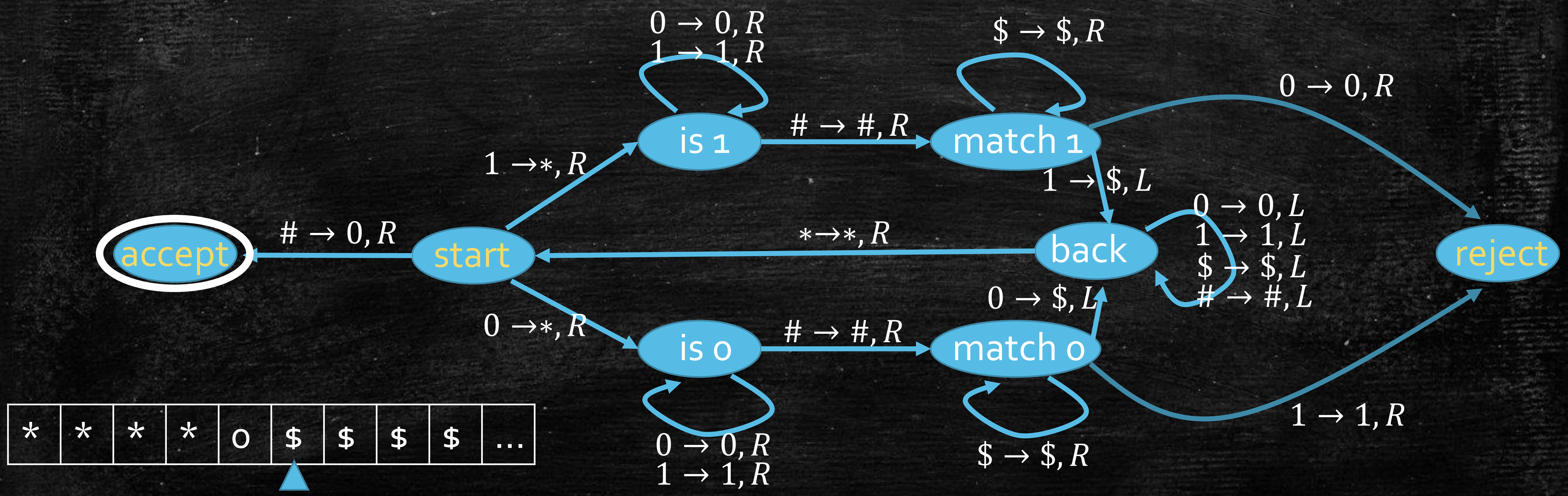
# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# TM Example: Check if two strings are identical

- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings $x$ and $y$ is identical



# Turing Machine

- If you do not appreciate a Turing machine, in this course, just treat it as a computer program or an algorithm (that outputs "accept" or "reject" as well as an output string)…
- Turing machine has the same power as a computer program or an algorithm, in the following sense:
- Whatever can be computed in polynomial time by a computer program or an algorithm can also be computed in polynomial time by a Turing machine.

# Polynomial Time TM

- **Definition.** A Turing Machine $\mathcal{A}$ is a polynomial time TM if there exists a polynomial $p$ such that $\mathcal{A}$ always terminates within $p(|x|)$ steps on input $x$.

# The Complexity Class **P**

- A decision problem $f: \Sigma^* \to \{0,1\}$ is in **P**, if there exists a polynomial time TM $\mathcal{A}$ such that
  - $\mathcal{A}$ accepts $x$ if $f(x) = 1$
  - $\mathcal{A}$ rejects $x$ if $f(x) = 0$
- Problems in **P** are those "easy" problems that can be solved in polynomial time.

# Examples for Problems in **P**

- [PATH] Given a graph $G = (V, E)$ and $s, t \in V$, decide if there is a path from $s$ to $t$.
  - Build a TM that runs BFS or DFS at $s$; accept if $t$ is reached; reject if the search terminates without reaching $t$.
  - PATH ∈ **P**
- [k-FLOW] Given a directed graph $G = (V, E)$, $s, t \in V$, a capacity function $c: E \to \mathbb{R}^+$, and $k \in \mathbb{R}^+$, decide if there is a flow with value at least $k$.
  - Build a TM that implements Edmonds-Karp, Dinic's, or other algorithms.
  - k-FLOW ∈ **P**
- [PRIME] Given $k \in \mathbb{Z}^+$ encoded in binary string, decide if $k$ is a prime number.
  - [Agrawal, Kayal & Saxena, 2004] PRIME ∈ **P**

# The Complexity Class **NP**

- A commonality with SAT, VertexCover, IndependentSet, SubsetSum, HamiltonianPath:
  - For a yes instance, it can be easily verified if a hint is given.
- SAT: a hint can be a valid assignment to the variables
- VertexCover/IndependentSet: a hint can be a valid set of $k$ vertices
- SubsetSum: a hint can be a sub-collection with sum $k$
- HamiltonianPath: a hint can be an encoding of a valid path

# The Complexity Class **NP**

- **NP**: Problems whose yes instances can be efficiently verified if hints are given.
- Formal Definition. A decision problem $f:\Sigma^* \to \{0,1\}$ is in **NP** if there exist a polynomial $q$ and a polynomial time TM $\mathcal{A}$ such that
  - If $x$ is a yes instance ($f(x)=1$), there exists $y \in \Sigma^*$ with $|y| \le q(|x|)$ such that $\mathcal{A}$ accepts the input $(x,y)$
  - If $x$ is a no instance ($f(x)=0$), for all $y \in \Sigma^*$ with $|y| \le q(|x|)$ such that $\mathcal{A}$ rejects the input $(x,y)$
- The string $y$ is called a certificate.
- SAT, VertexCover, IndependentSet, SubsetSum, HamiltonianPath are all in **NP**.

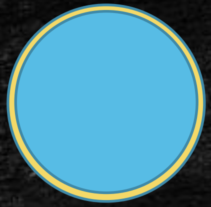
# $\mathbf{P} \subseteq \mathbf{NP}$

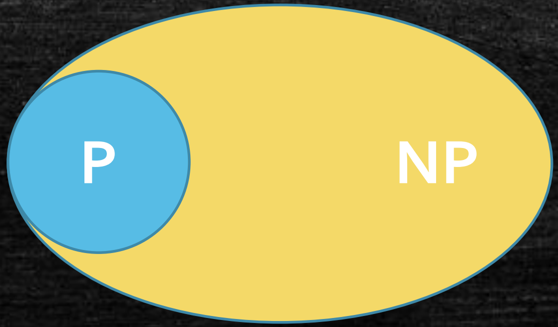
- Proof. If a decision problem $f: \Sigma^* \to \{0,1\}$ is in **P**, we will show it is in **NP**.
- By definition of **P**, there exists a polynomial time TM $\mathcal{A}$ such that $\mathcal{A}$ accepts $x$ if and only if $f(x) = 1$.
- Let $\mathcal{A}'$ be a TM such that it outputs $\mathcal{A}(x)$ on input $(x, y)$. That is, $\mathcal{A}'$ implements $\mathcal{A}$ and ignore $y$.
- If $f(x) = 1$, there exists $y$, say, $y = \emptyset$, such that $\mathcal{A}'$ accepts $(x, y)$.
- If $f(x) = 0$, for all $y$, $\mathcal{A}'$ rejects $(x, y)$.
- Thus, $f \in \mathbf{NP}$.

# Central Open Problem: **P** vs. **NP**

- Central Open Problem: Does **P** equals **NP**?
- Most research believes no…
  - If **P** = **NP**, we do not need the certificate: we can just "guess" it correctly and efficiently… This doesn't seem possible.
  - Given an exam question, do you believe solving the question is much harder than checking if someone's solution to the question is correct? **P** = **NP** would suggest they are equally easy…

**P** = **NP**






**P** $\subsetneq$ **NP**

# **NP** Problems

- We have seen many **NP** problems not known in **P**
  - SAT
  - VertexCover
  - IndependentSet
  - SubsetSum
  - HamiltonianPath
- Are some of these problems “more difficult” than the others?

# 3SAT

- A 3-CNF formula is a CNF formula where each clause contains at most three literals:
  - a 3-CNF formula: $(x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$
  - Not a 3-CNF formula: $(x_1 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4)$
- [3SAT] Given a 3-CNF formula, decide if there is a value assignment to the variables to make the formula $\mathrm{true}$.
- Clearly, 3SAT is at most as hard as SAT, as it is a special case.
- We will prove 3SAT is also at least as hard as SAT.
  - so that SAT and 3SAT are "equally hard"

# Proving 3SAT is "weakly harder than" SAT

- Idea: given a CNF formula $\phi$, construct a 3-CNF formula $\phi'$ such that $\phi$ is a $\text{yes}$ SAT instance if and only if $\phi'$ is a $\text{yes}$ 3SAT instance.
- If converting $\phi$ to $\phi'$ can be done in polynomial time, being able to solve 3SAT in polynomial time implies being able to solve SAT in polynomial time.
  - That is, 3SAT is weakly harder than SAT.

# Proving 3SAT is "weakly harder than" SAT

- We can "break" a long clause in $\phi$ to shorter clauses by introducing new variables:
- $(x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4) = (x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee \neg x_3 \vee \neg x_4)$
  - For example, if $x_2 = \text{true}$ is the one making LHS true, we can set $x_2 = \text{true}$, $y_1 = \text{false}$ to make RHS true.
  - If $x_1 = x_2 = \text{false}$ and $x_3 = x_4 = \text{true}$ so that LHS is false, at least one of the two clauses on RHS is false.
- We can "break" a even longer clause to clauses with at most three literals:
- $(x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4 \vee x_5 \vee x_6) = (x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee \neg x_3 \vee y_2) \wedge (\neg y_2 \vee \neg x_4 \vee y_3) \wedge (\neg y_3 \vee x_5 \vee x_6)$
  - For example, if $x_4 = \text{false}$ is the one making LHS true, we can set $y_3 = \text{false}$, $y_2 = \text{true}$, $y_1 = \text{true}$ to guarantee RHS is true.

# Proving 3SAT is "weakly harder than" SAT

In general:

- $(\ell_1 \vee \cdots \vee \ell_k) = (\ell_1 \vee \ell_2 \vee y_1) \wedge (\neg y_1 \vee \ell_3 \vee y_2) \wedge \cdots \wedge (\neg y_{k-2} \vee \ell_{k-1} \vee \ell_k)$
- If a literal $\ell_i$ is true, we can make all RHS clauses true by properly setting $y_i$'s

$$(\ell_1 \vee \ell_2 \vee \underset{\substack{\uparrow\\ \text{true}}}{y_1}) \wedge \cdots \wedge (\underset{\substack{\uparrow\\ \text{false}}}{\neg y_{i-3}} \vee \ell_{i-1} \vee \underset{\substack{\uparrow\\ \text{true}}}{y_{i-2}}) \wedge (\underset{\substack{\uparrow\\ \text{false}}}{\neg y_{i-2}} \vee \overset{\substack{\text{true}\\ \downarrow}}{\ell_i} \vee \underset{\substack{\uparrow\\ \text{false}}}{y_{i-1}}) \wedge (\underset{\substack{\uparrow\\ \text{true}}}{\neg y_{i-1}} \vee \ell_{i+1} \vee \underset{\substack{\uparrow\\ \text{false}}}{y_i}) \wedge \cdots \wedge (\underset{\substack{\uparrow\\ \text{true}}}{\neg y_{k-2}} \vee \ell_{k-1} \vee \ell_k)$$

- If all of $\ell_i$'s are false, we cannot make all RHS clauses true:
  - We have to set $y_1 =$ true to make the first clause true
  - After that, we have to make $y_2 =$ true to make the second clause true
  - ……
  - We have to make $y_{k-2} =$ true; however, this will make the last clause false

# Proving 3SAT is “weakly harder than” SAT

- We have described how to convert a CNF formula $\phi$ to a 3-CNF formula $\phi'$.
- The conversion can clearly done in polynomial time.
- We have shown that $\phi$ is a $\text{yes}$ SAT instance if and only if $\phi'$ is a $\text{yes}$ 3SAT instance.
- If we have a polynomial time algorithm for 3SAT, we have a polynomial time algorithm for SAT:
  - Given input $\phi$, compute $\phi'$
  - Solve 3SAT instance $\phi'$ and obtain answer $\text{yes}$ or $\text{no}$
  - Output the same answer for $\phi$

# IndependentSet is "weakly harder" than 3SAT

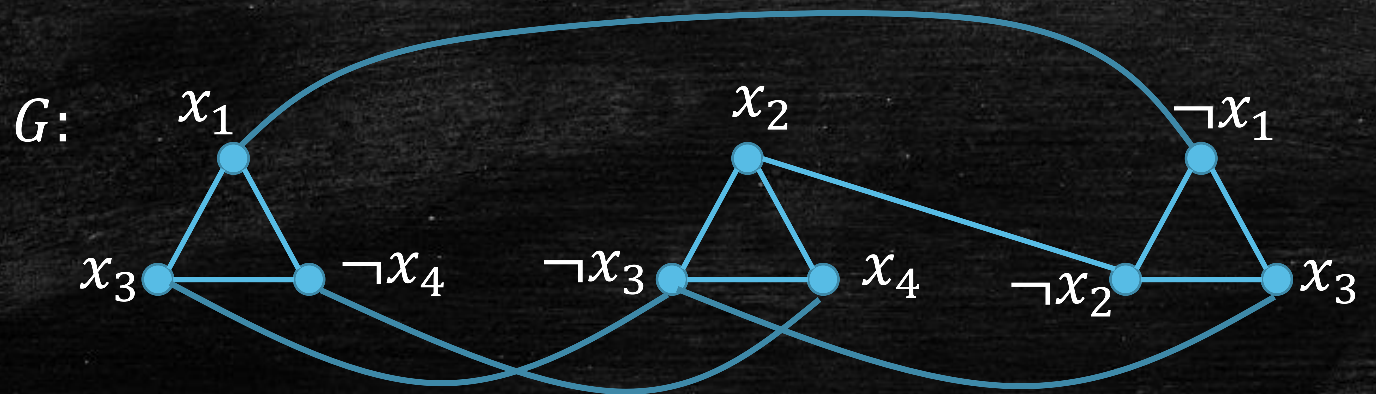
- Same Idea before: Given a 3SAT instance $\phi$, construct a IndependentSet instance $(G = (V, E), k)$ such that $\phi$ is a $\text{yes}$ instance if and only if $(G = (V, E), k)$ is a $\text{yes}$ instance.
- If construction can be done in polynomial time, this implies IndependentSet is weakly harder than 3SAT.

# IndependentSet is "weakly harder" than 3SAT

Here is how we do it:

- For each clause, construct a triangle where three vertices represent three literals.
- Connect two vertices if one represents the negation of the other.
- Set $k$ in IndependentSet instance to the number of clauses

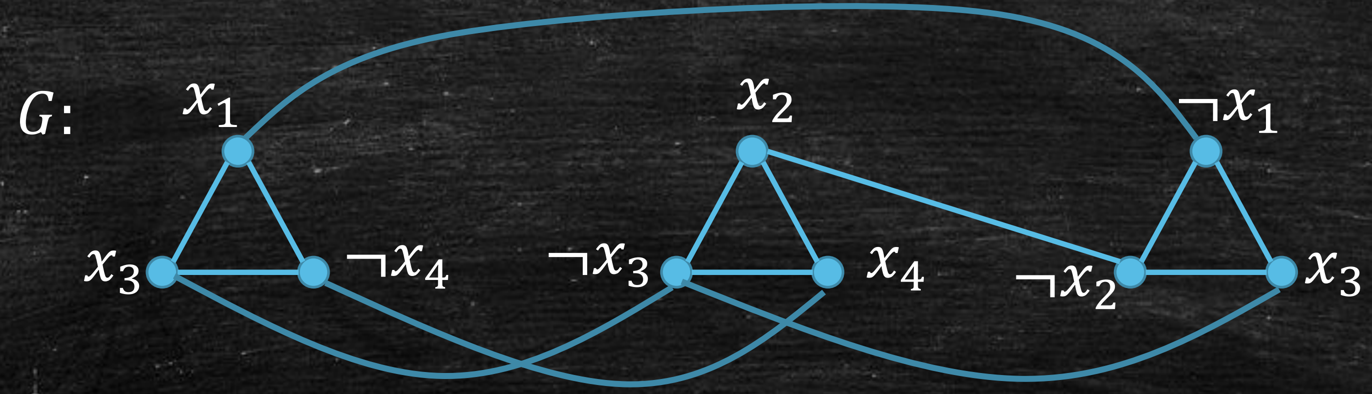
$$\phi = (x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$




$k = 3$

# IndependentSet is "weakly harder" than 3SAT

$$\phi = (x_1 \lor x_3 \lor \neg x_4) \land (x_2 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor x_3)$$

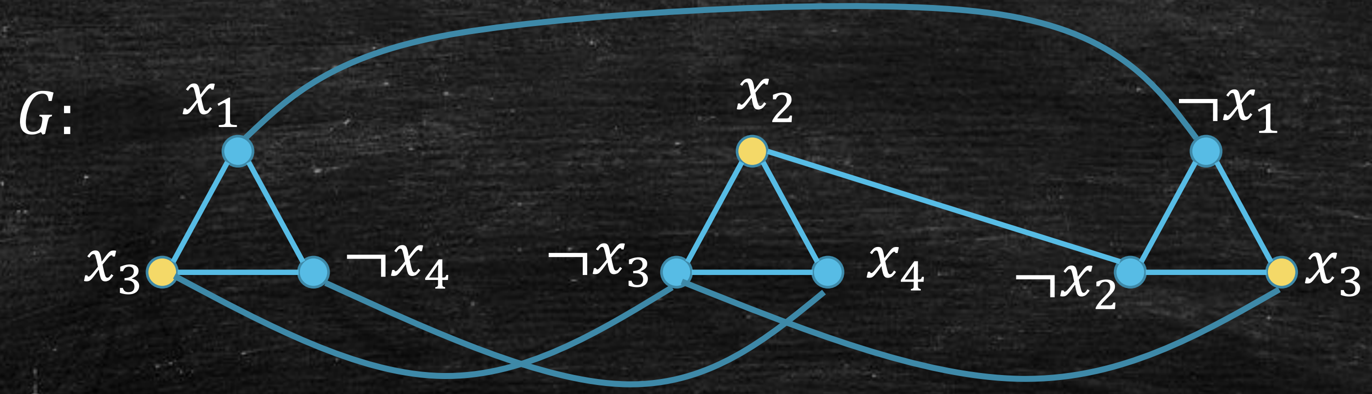



$k = 3$

- If $\phi$ is a yes instance, each clause must have a literal with value true.
- For each triangle in $G$, pick exactly one vertex representing a true literal in $S$.
- $S$ is an independent set and $|S| = k$. So $(G, k)$ is a yes instance.

# IndependentSet is "weakly harder" than 3SAT

$$\phi = (x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$




$k = 3$

- Example: $x_1 = x_2 = x_3 = x_4 =$ true makes $\phi =$ true
- We choose exactly one true literal in each clause, for example,
  - $(x_1 \vee x_3 \vee \neg x_4)$
  - $(x_2 \vee \neg x_3 \vee x_4)$
  - $(\neg x_1 \vee \neg x_2 \vee x_3)$

# IndependentSet is "weakly harder" than 3SAT

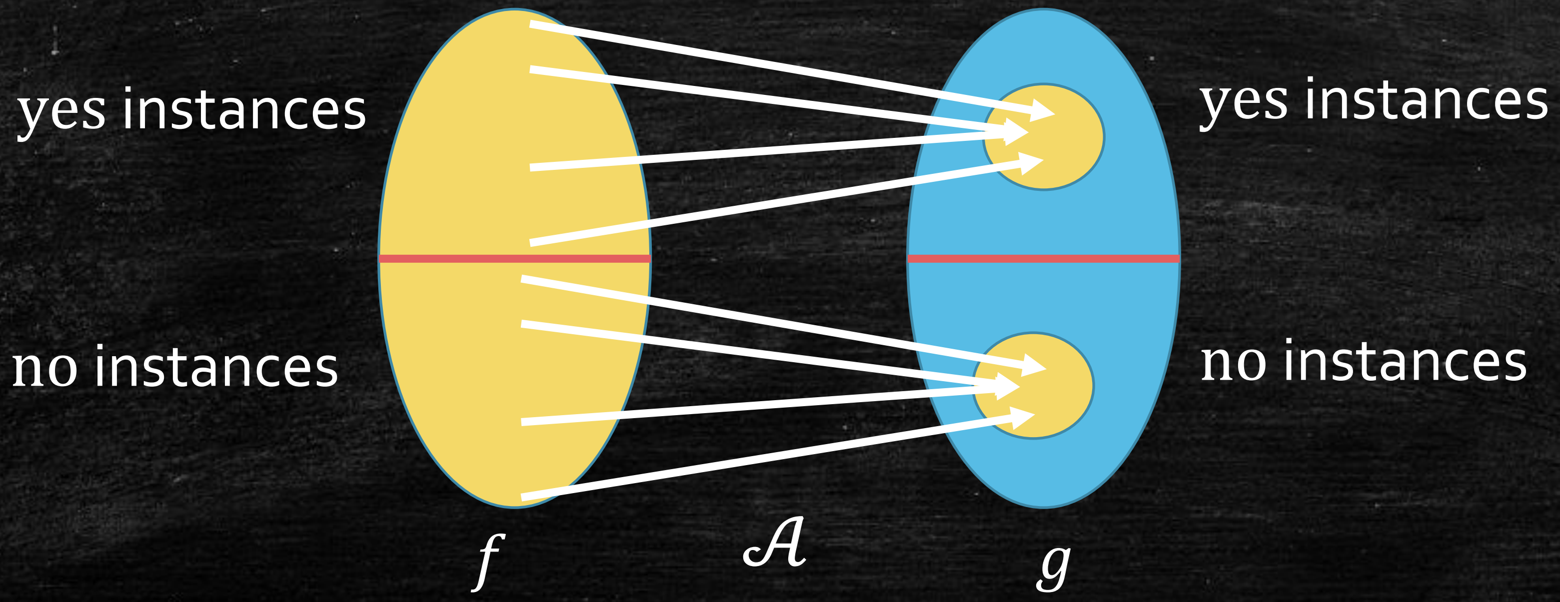
- If $\phi$ is a no instance, for contradiction, assume $(G, k)$ is a yes instance. Let $S$ with $|S| = k$ be the independent set.
- $S$ must contain exactly one vertex in each triangle.
  - because any two vertices in a triangle is connected
- Assign true to the literals representing the chosen vertices.
  - We will not assign both true and false to a same literal, as $x_i$ and $\neg x_i$ is connected.
- For variables not yet assigned a value, assign values to them arbitrarily.
- The resultant assignment makes $\phi$ true (as each clause has a true literal), contradicting to that $\phi$ is a no instance!

# Reduction

- A decision problem $f$ Karp reduce to (or simply, reduce to) a decision problem $g$ if there is a polynomial time TM $\mathcal{A}$ such that
  - $\mathcal{A}$ outputs a $\text{yes}$ instance of $g$ if a $\text{yes}$ instance of $f$ is input
  - $\mathcal{A}$ outputs a $\text{no}$ instance of $g$ if a $\text{no}$ instance of $f$ is input
- Denoted as $f \leq_k g$
  - Very intuitive: the difficulty level of $f$ is weakly less than that of $g$
- We have just proved:
  - $\text{SAT} \leq_k \text{3SAT}$
  - $\text{3SAT} \leq_k \text{IndependentSet}$

# Reduction

- In the reduction, $f \leq_k g$, the TM $\mathcal{A}$ defines a mapping.
- The mapping needs not to be one-to-one.
- The mapping needs not to be onto.



# Transitivity of Reduction

- **Theorem.** If $f \leq_k g$ and $g \leq_k h$, then $f \leq_k h$.
- If $g$ is (weakly) harder than $f$ and $h$ is (weakly) harder than $g$, then $h$ is (weakly) harder than $f$.
- Proof. Let $\mathcal{A}_1$ be the polynomial time TM doing $f \leq_k g$ and $\mathcal{A}_2$ be the polynomial time TM doing $g \leq_k h$.
- Let $\mathcal{A} = \mathcal{A}_1 \circ \mathcal{A}_1$ be the TM that first executes $\mathcal{A}_1$ and then executes $\mathcal{A}_2$ (using the output of $\mathcal{A}_1$ as input of $\mathcal{A}_2$).
- Then $\mathcal{A}$ does the job of $f \leq_k h$.
- $\mathcal{A}$ runs in polynomial time: the time complexity of $\mathcal{A}$ is the sum of the time complexities of $\mathcal{A}_1$ and $\mathcal{A}_2$, and $\mathcal{A}_1$ and $\mathcal{A}_2$ are polynomial time TMs.

# More Results in Reduction

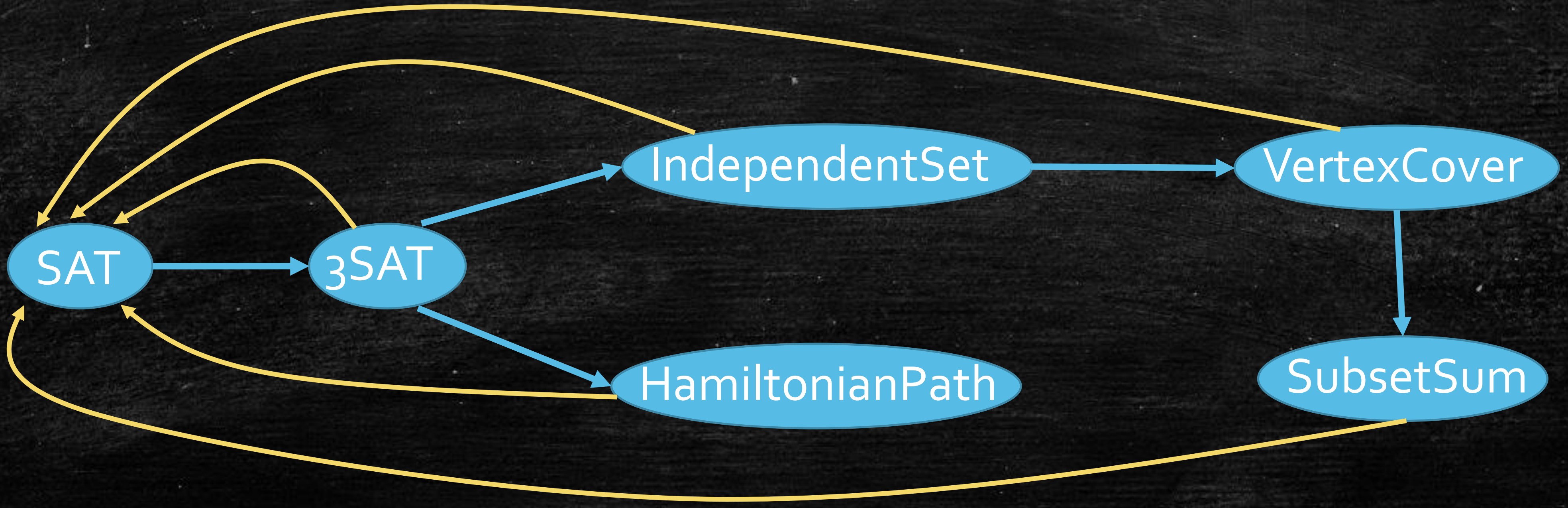
- In Problem 3 of Assignment 5, you will prove $S$ is an independent set of $G = (V, E)$ if and only if $V \setminus S$ is a vertex cover.
- Thus, $\text{IndependentSet} \leq_k \text{VertexCover}$
  - The reduction $\mathcal{A}$ simply maps $(G = (V, E), k)$ to $(G = (V, E), |V| - k)$
- It is also true that:
  - $\text{VertexCover} \leq_k \text{SubsetSum}$
  - $\text{3SAT} \leq_k \text{HamitonianPath}$

# The Hardest Problem in **NP**

- We have built difficulty relations between many problems in **NP**.
- Does there exist a problem in **NP** that is the hardest?
- **Definition.** A decision problem $f$ is NP-hard if $g \leq_k f$ for any problem $g \in \mathbf{NP}$.
- **Definition.** A decision problem $f$ is NP-complete if $f \in \mathbf{NP}$ and $g \leq_k f$ for any problem $g \in \mathbf{NP}$.
- **[Cook-Levin Theorem] SAT is NP-complete.**

# More NP-Complete Problems

- Cook-Levin Theorem implies the yellow arrows, since all the problems below are in **NP**.
- Each problem is NP-complete
  - By transitivity: any **NP** problem reduce to SAT, and SAT reduce to each of these problems.
- These problems are "equally hard", and are the hardest problems in **NP**.



# Intuition behind Cook-Levin Theorem

- We have seen SAT is in **NP**.
- Consider an arbitrary **NP** problem $f$. We will show $f \leq_k \mathrm{SAT}$.
- For a yes instance $x$, there exist a polynomial time TM $\mathcal{A}$ and a polynomial length certificate $y$ such that $\mathcal{A}$ accepts $(x, y)$.
- Consider a computation tableau that records the tape at every step of $\mathcal{A}$'s execution.

| | | | $x$ | | | | | $y$ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Step 0 | $x_0$ | $x_1$ | $x_2$ | … | $x_n$ | $y_0$ | $y_1$ | $y_2$ | … | $y_m$ |
| Step 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| Step 2 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| Final Step | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

# Intuition behind Cook-Levin Theorem

| | $\overbrace{\hphantom{x_0 \quad x_1 \quad x_2 \quad \dots \quad x_n}}^{x}$ | | | | | $\overbrace{\hphantom{y_0 \quad y_1 \quad y_2 \quad \dots \quad y_m}}^{y}$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Step 0 | $x_0$ | $x_1$ | $x_2$ | … | $x_n$ | $y_0$ | $y_1$ | $y_2$ | … | $y_m$ |
| Step 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| Step 2 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| Final Step | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

- For each $y_i$ and each cell in the tape from Step 1 to the final step, construct a Boolean variable for the SAT instance.
- We can use clauses to ensure the tableau gives a valid TM computation.
- E.g., we can use two clauses $(x \vee \neg y) \wedge (\neg x \vee y)$ to enforce $x = y$.

# Intuition behind Cook-Levin Theorem

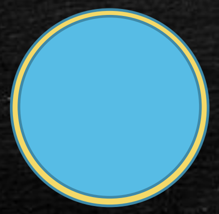
- High-level Intuition: a CNF formula is sufficient to simulate the execution of a Turing Machine!
- If $x$ for the **NP** problem $f$ is a $\mathrm{yes}$ instance, the CNF formula constructed can be satisfied:
  - Assign $y_i = \mathrm{true}$ if and only if the $i$-th bit of $y$ is $1$.
  - Assign each other variable the value corresponding to the value of the cell in the computation tableau.
- If $x$ for the **NP** problem $f$ is a $\mathrm{no}$ instance, the CNF formula constructed <u>cannot</u> be satisfied:
  - Otherwise, we can find a certificate $y = y_1 y_2 \cdots y_m$ that fools the TM to accept $(x, y)$.

# Solving a NP-complete problem implies **P** = **NP**

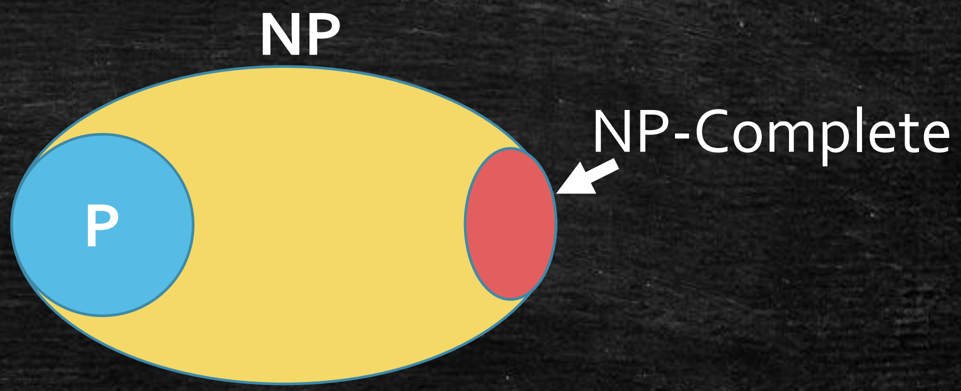
- **Theorem.** If $f$ is NP-complete and $f \in$ **P**, then **P** = **NP**.
- Proof. Suppose there is a polynomial time TM $\mathcal{A}$ that decides $f$. We will show $g \in$ **P** for any $g \in$ **NP**.
- Since $f$ is NP-hard, $g \leq_k f$, and let $\mathcal{A}'$ be the polynomial time TM that does the reduction.
- Then $\mathcal{A} \circ \mathcal{A}'$ is the polynomial time TM that decides $g$.
- Thus, $g \in$ **P**.

- If you solve any of SAT, 3SAT, IndependentSet, VertexCover, SubsetSum, HamiltionianPath, you will be the greatest person in the 21st century!

# P vs NP



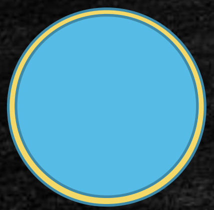
$P = NP$



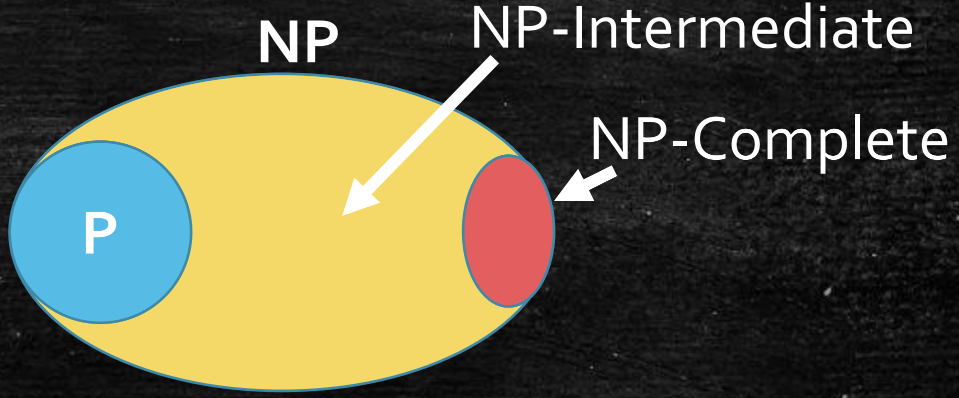

$P \subsetneq NP$

# NP-Intermediate

- **[Ladner's Theorem]** If P $\neq$ NP, then there exist decision problems that are neither in P nor NP-complete.
- Such problems are called NP-intermediate.
- However, we do not know any "natural" NP-intermediate problems.



P = NP




$P \subsetneq NP$

# NP-Hard vs NP-Complete

Difference between NP-hardness and NP-completeness:

- For decision problems: NP-complete = NP-hard + (in **NP**)
  - There are NP-hard problems that are not in **NP**; these problems are even harder than NP-complete problems.
- NP-hardness can describe optimization problems:
  - Maximum Independent Set is NP-hard
  - Minimum Vertex Cover is NP-hard
  - Max-3SAT is NP-hard
  - Finding a longest simple path is NP-hard
  - Etc.

# VertexCover $\leq_k$ SubsetSum

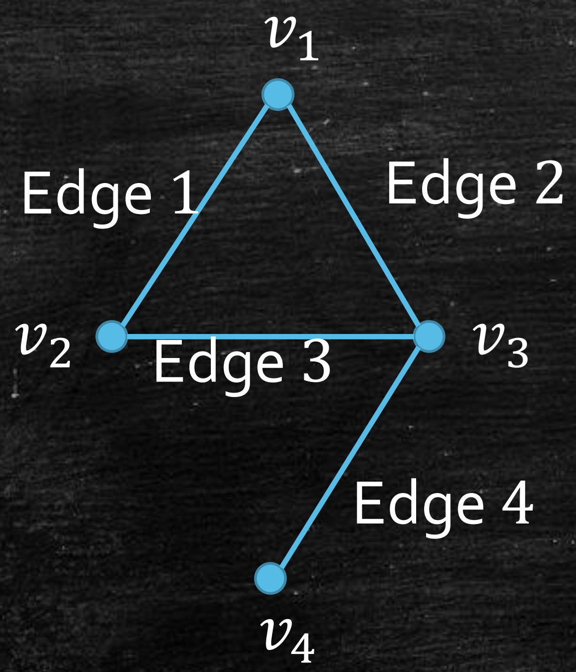
- We first consider the following "vector version" of SubsetSum.
- [VectorSubsetSum] Given a collection of integer vectors $S = \{\mathbf{a}_1, \ldots, \mathbf{a}_n : \mathbf{a}_i \in \mathbb{Z}^m\}$ and a vector $\mathbf{k} \in \mathbb{Z}^m$, decide if there exists $T \subseteq S$ with $\sum_{\mathbf{a}_i \in T} \mathbf{a}_i = \mathbf{k}$.
- We will show that
  1. VertexCover $\leq_k$ VectorSubsetSum
  2. VectorSubsetSum $\leq_k$ SubsetSum

# $\text{VertexCover} \le_k \text{VectorSubsetSum}$

- Given a VertexCover instance $(G = (V, E), k)$, we will. construct a VectorSubsetSum instance $(S, \mathbf{k})$.
- First, we label the edges with $1, 2, \dots, |E|$ (in arbitrary order).
- For each $v_i \in V$, construct a $(|E| + 1)$-dimensional vector $\mathbf{a}_i \in S$ such that $\mathbf{a}_i[0] = 1$ and for each $j = 1, \dots, |E|$:
$$\mathbf{a}_i[j] = \begin{cases} 1 & \text{if } v_i \text{ is an endpoint of edge } j \\ 0 & \text{otherwise} \end{cases}$$
- For each edge $j$, construct $\mathbf{b}_j \in S$ where $\mathbf{b}_j[j] = 1$ is the only non-zero entry.
- Let $\mathbf{k} = (k, 2,2, \dots, 2)$.

# Example

$G =$




$k = 3$

a VertexCover instance

$$
\begin{aligned}
\mathbf{a}_1 &= (1,\ 1,\ 1,\ 0,\ 0)\\
\mathbf{a}_2 &= (1,\ 1,\ 0,\ 1,\ 0)\\
\mathbf{a}_3 &= (1,\ 0,\ 1,\ 1,\ 1)\\
\mathbf{a}_4 &= (1,\ 0,\ 0,\ 0,\ 1)\\
\mathbf{b}_1 &= (0,\ 1,\ 0,\ 0,\ 0)\\
\mathbf{b}_2 &= (0,\ 0,\ 1,\ 0,\ 0)\\
\mathbf{b}_3 &= (0,\ 0,\ 0,\ 1,\ 0)\\
\mathbf{b}_4 &= (0,\ 0,\ 0,\ 0,\ 1)\\
\mathbf{k} &= (3,\ 2,\ 2,\ 2,\ 2)
\end{aligned}
$$

a VectorSubsetSum instance

# Ideas Behind the Reduction

- Picking $\mathbf{a}_i \in T$ represents picking $v_i$ in the vertex cover.
- The 0-th entry of $\mathbf{k}$ is set to $k$, enforcing exactly $k$ vertices must be picked.
- The $j$-th entry of $\mathbf{k}$ is set to $2$ enforcing edge $j$ must be covered:
  - Say, edge $j$ is $(v_{i_1}, v_{i_2})$
  - If $\mathbf{a}_{i_1}, \mathbf{a}_{i_2} \in T$, we are fine, as the $j$-th entries already add up to $2$.
  - If one of $\mathbf{a}_{i_1}, \mathbf{a}_{i_2}$ is chosen in $T$, we are also fine, as we can include $\mathbf{b}_j \in T$.
  - If $\mathbf{a}_{i_1}, \mathbf{a}_{i_2} \notin T$, we are <u>not</u> fine: the $j$-th entries add up to at most $1$ even if we include $\mathbf{b}_j \in T$.
- We are done! $\mathrm{VertexCover} \leq_k \mathrm{VectorSubsetSum}$

# VectorSubsetSum $\leq_k$ SubsetSum

- We can convert a vector $\mathbf{a} = (\mathbf{a}[0], \dots, \mathbf{a}[m])$ to a large number.
- For example, convert $\mathbf{a} = (1, 4, 5, 3)$ to number $1453$
  - $1453 = \mathbf{a}[0] \times 1000 + \mathbf{a}[1] \times 100 + \mathbf{a}[2] \times 10 + \mathbf{a}[3] \times 1$
- We are using decimal representation in the above example…
- To avoid carry, use N-ary representation instead (for sufficiently large N)?
- Additions with vectors are now equivalent to additions with numbers, since we do not have carry issue.
- $\text{VectorSubsetSum} \leq_k \text{SubsetSum}$

# SubsetSum is NP-complete

- We have seen SubsetSum is in NP.
- We have proved
  1. $\text{VertexCover} \leq_k \text{VectorSubsetSum}$
  2. $\text{VectorSubsetSum} \leq_k \text{SubsetSum}$

# This Lecture

- Learn what are P and NP
- Cook-Levin Theorem and NP-complete problems
- Reduction

# Take Home Messages

- SAT (3SAT), VertexCover, IndependentSet, SubsetSum, HamiltonianPath are the hardest problems in **NP**, and they are NP-complete.
- Reduction is a effective tool to show one problem is "weakly harder" than another.