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

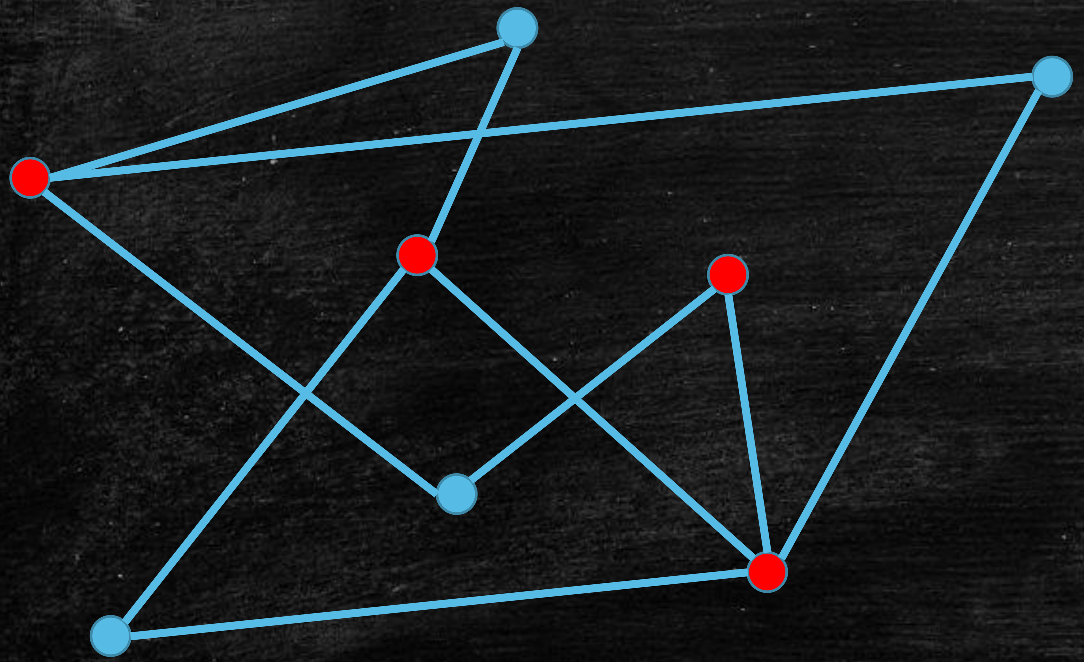
- SAT
- Vertex Cover
- Independent Set
- Subset Sum
- Hamiltonian Path

SAT (Boolean Satisfiability Problem)

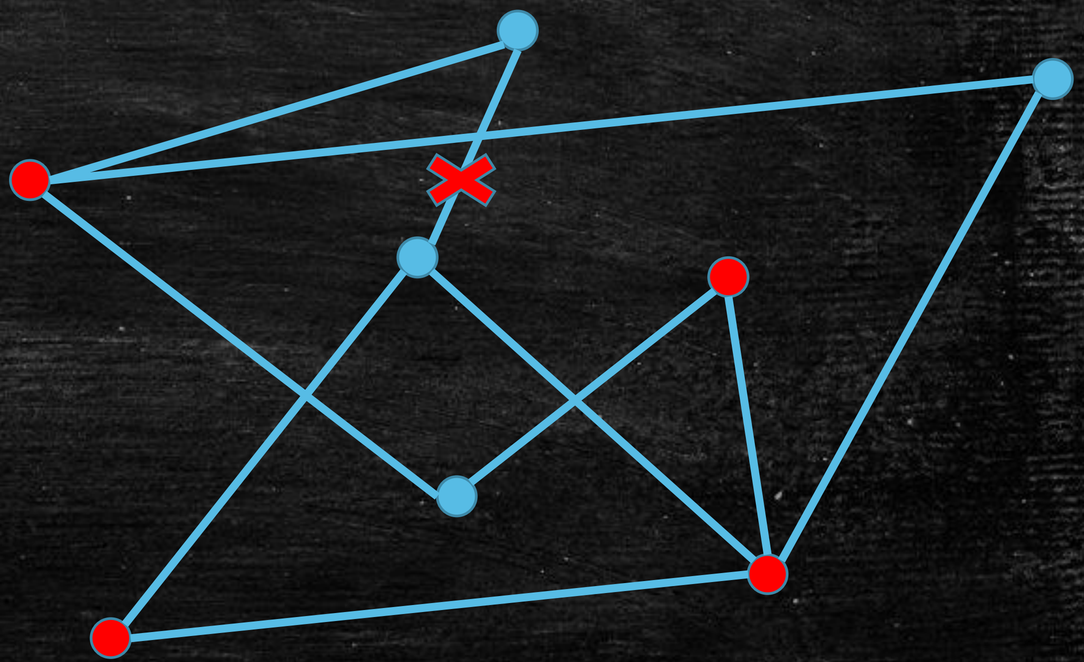
- A Boolean formula is built from variables, operators AND (∧), OR (∨), NOT (¬), and parentheses.
  - Example: $(x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$
- A Boolean formula is in conjunctive normal form (CNF) if it is an "AND" of many clauses:
  - Each 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 ϕ, decide if there is a value assignment to the variables to make ϕ 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

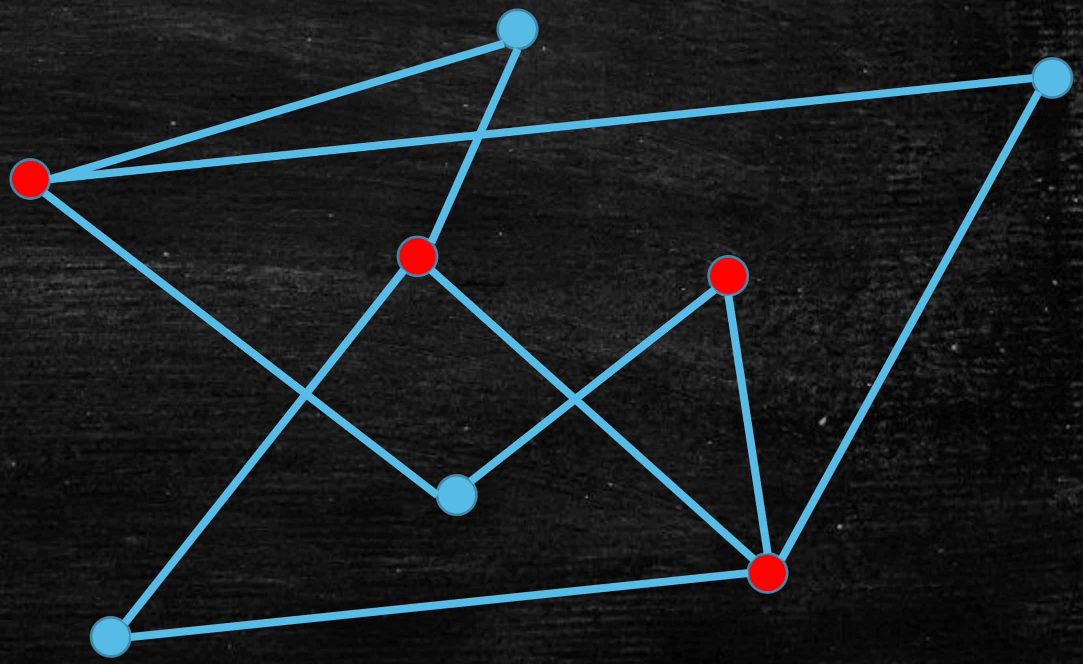


not a vertex cover

Vertex Cover Problem

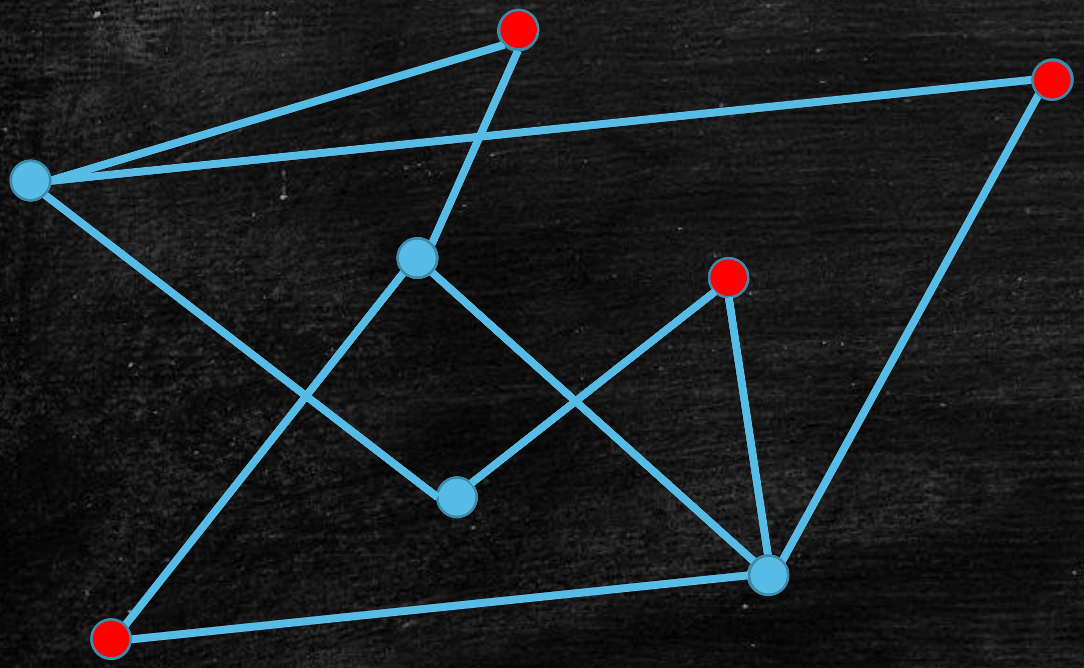
- [Vertex Cover Problem] Given an undirected graph $G = (V, E)$ and $k \in \mathbb{Z}^+$, decide if the graph has a vertex cover of size k.

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

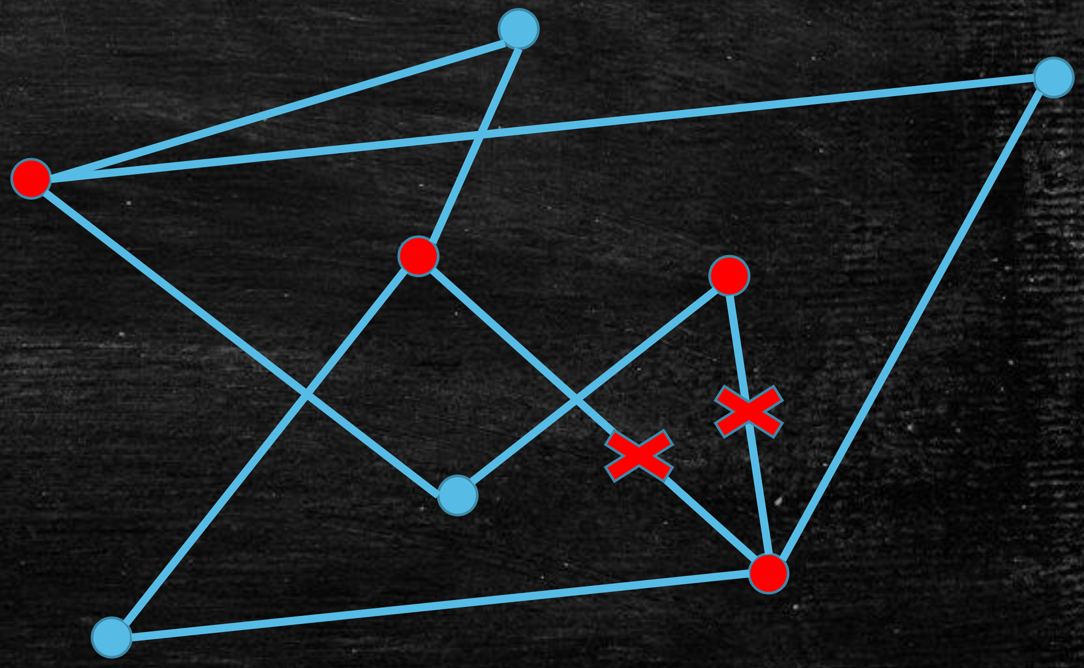


Independent Set

- 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

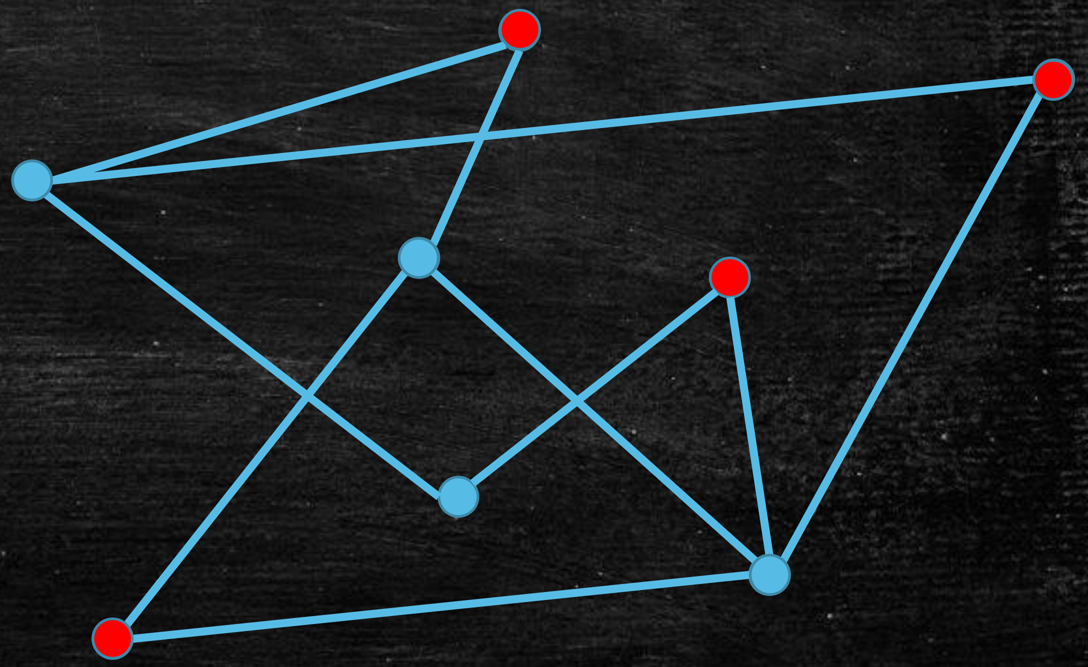


not an independent set

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.

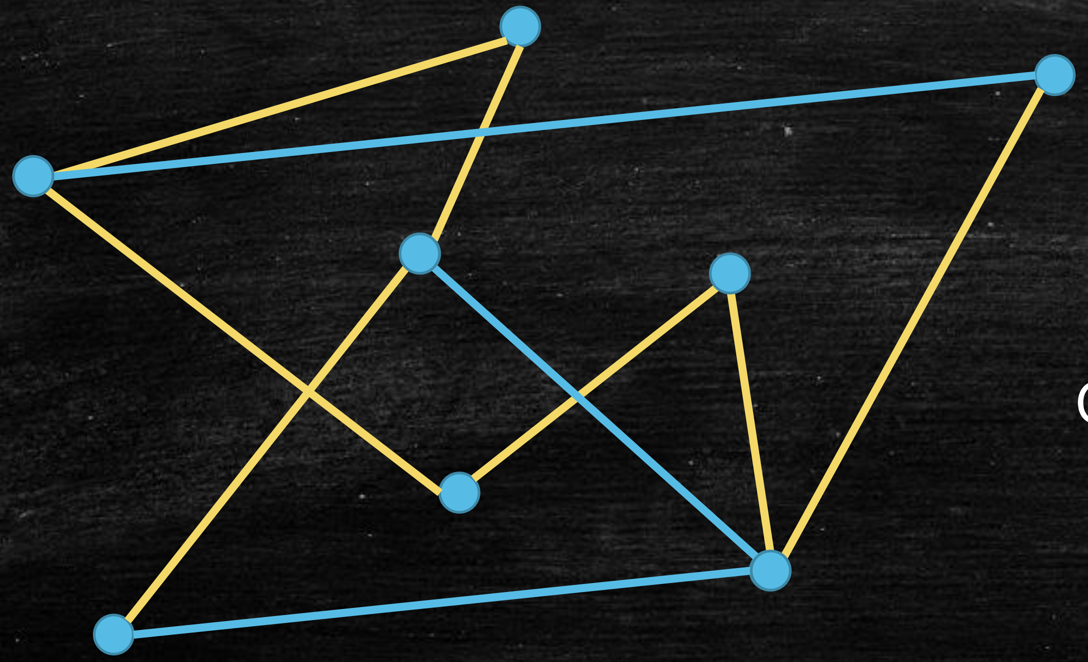


Subset Sum Problem

- [Subset Sum Problem] Given a collection of integers $S = \{a_1, \dots, a_n\}$ 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 yes for $S = \{1,1,6,13,27\}$ and $k = 21$, as $1 + 1 + 6 + 13 = 21$.
- The output should be 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

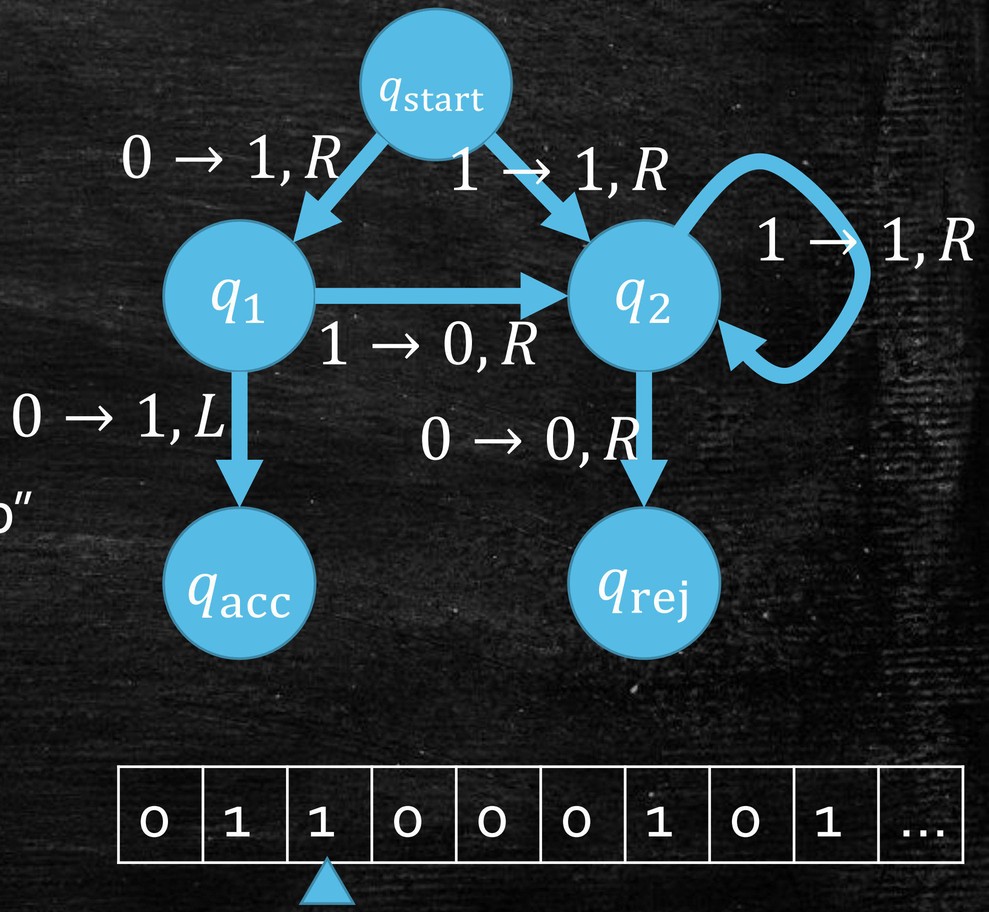
- A decision problem is a function $f: \Sigma^* \to \{0,1\}$
- Σ - set of alphabets: for example, binary alphabets $\Sigma = \{0,1\}$
- Σ^n - set of strings using alphabets in Σ with length n
- $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$ - set of all strings with any lengths
- $x \in \Sigma^*$ - an 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, Σ, δ)
  - one tape: contains infinitely many cells
    - Each cell can store an alphabet
  - A moving head pointing at a cell of the tape
  - Σ: set of alphabets
  - Q: set of states, each state specifying “the current step”
  - Transition function $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R\}$
    - 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

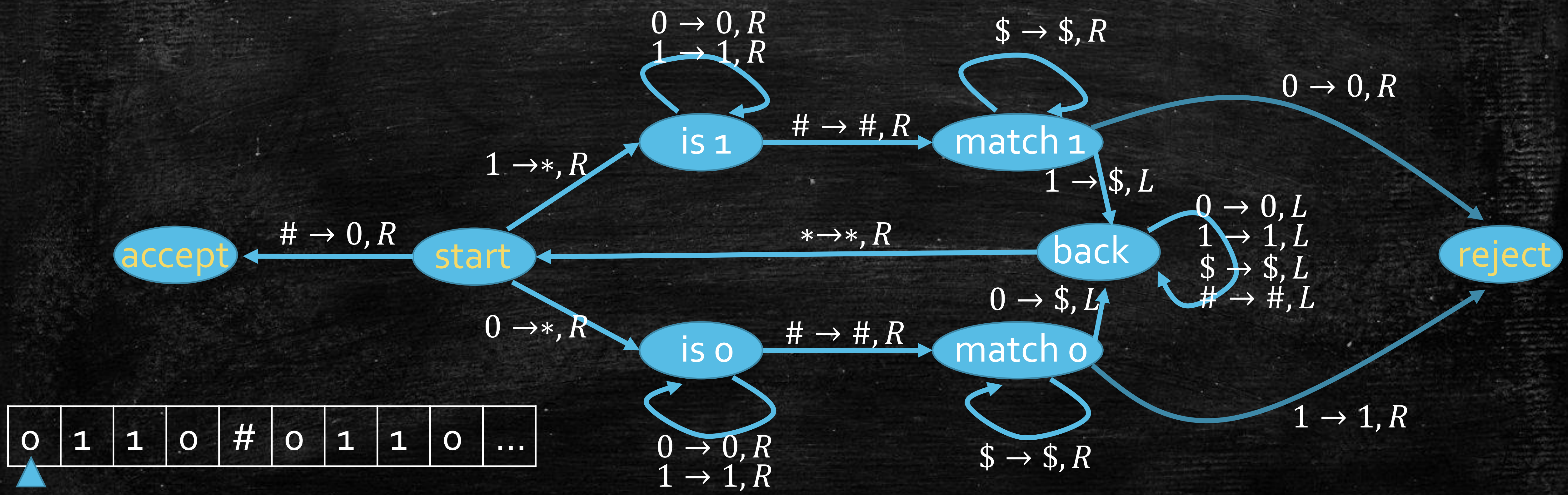


Turing Machine: Start and Terminate

- 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_{acc} and q_{rej}
  - TM terminates when reaching a halting state
  - TM accepts a string if q_{acc} is reached
  - TM rejects a string if q_{rej} is reached
  - TM's output is the content on the tape when TM terminates

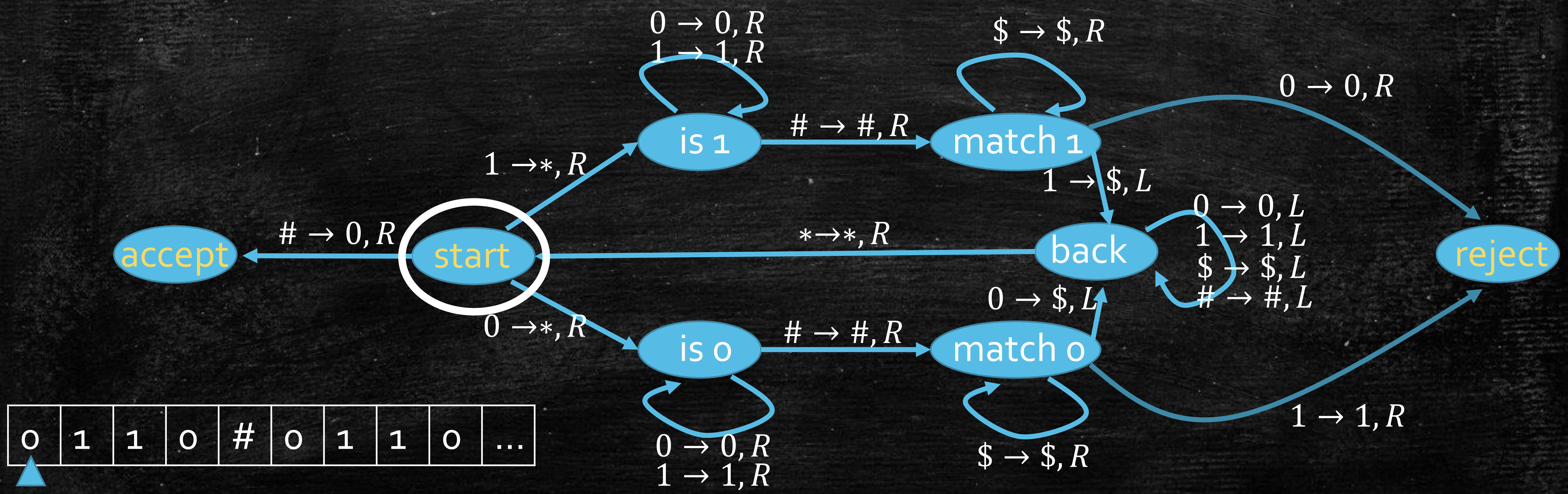
TM Example: Check if two strings are identical

- 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



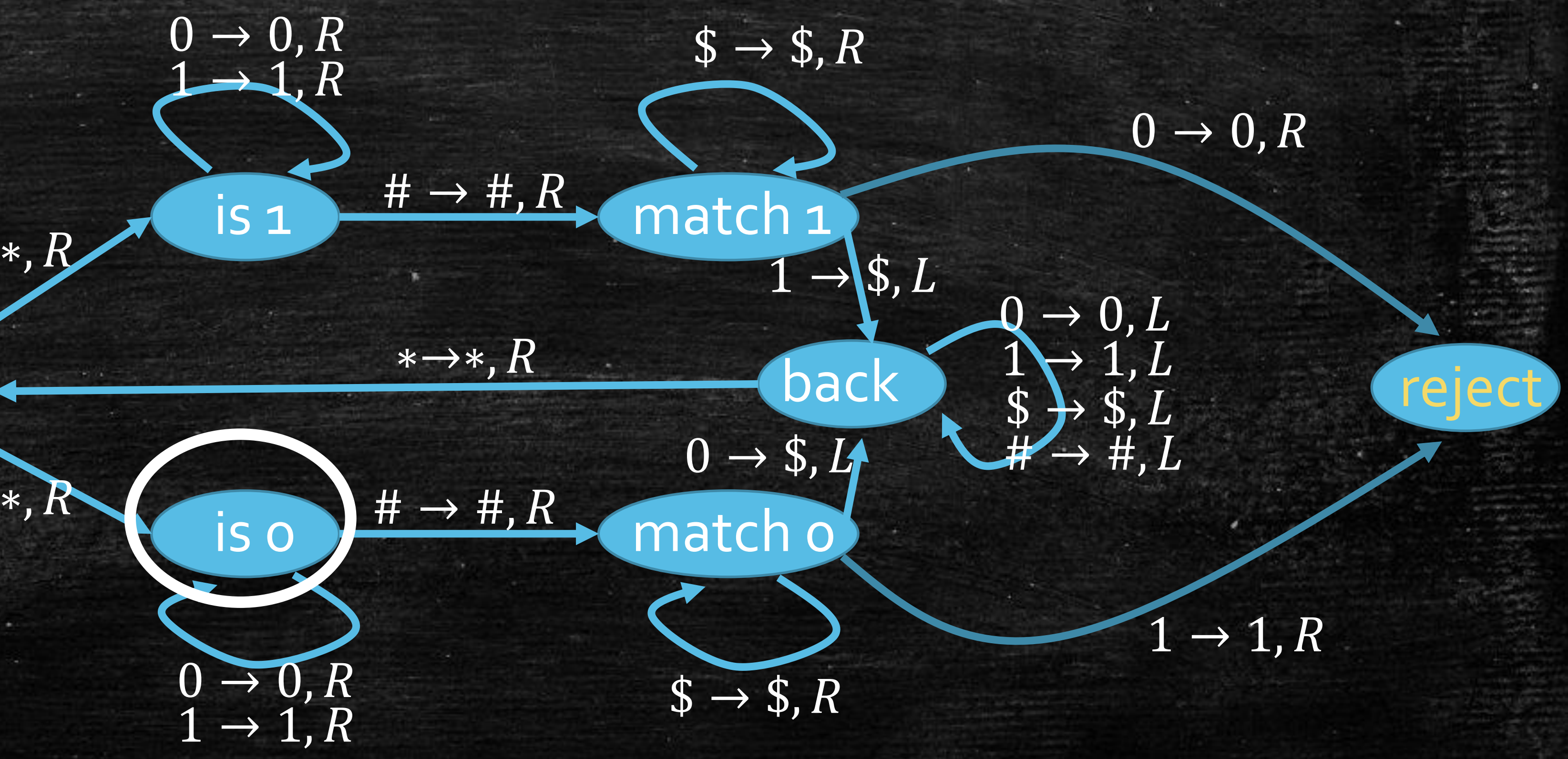
TM Example: Check if two strings are identical

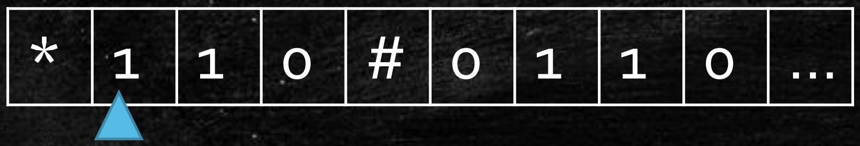
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



TM Example: Check if two strings are identical

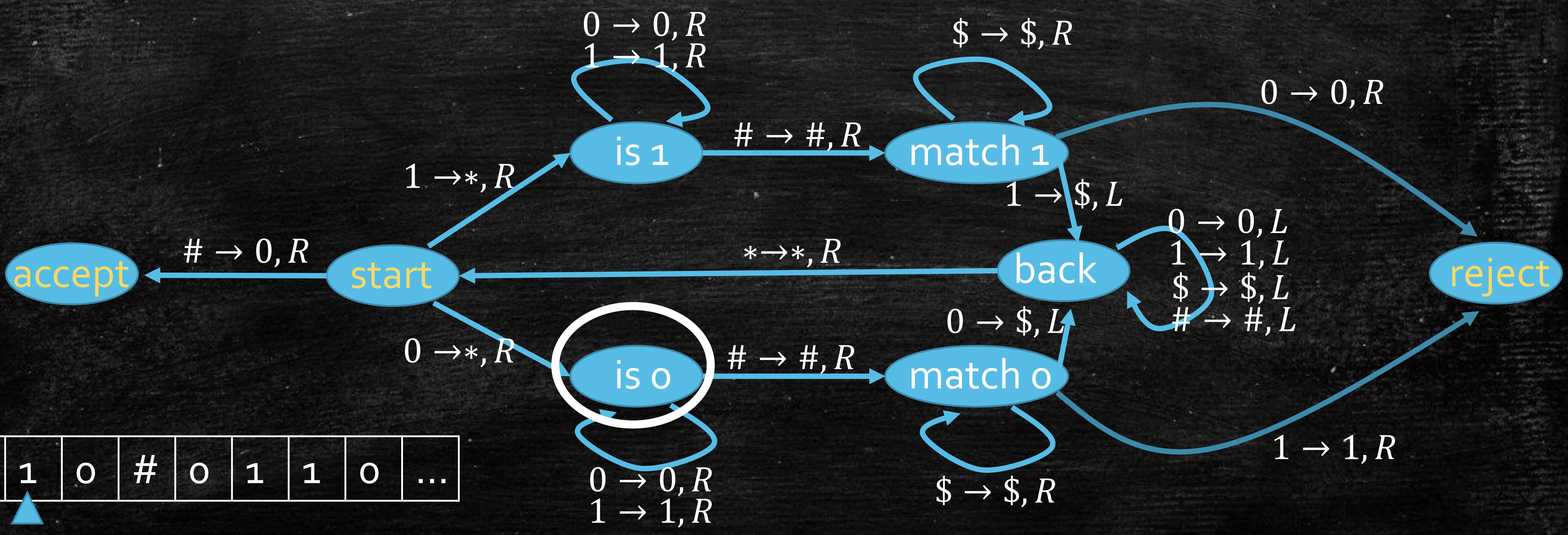
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical





TM Example: Check if two strings are identical

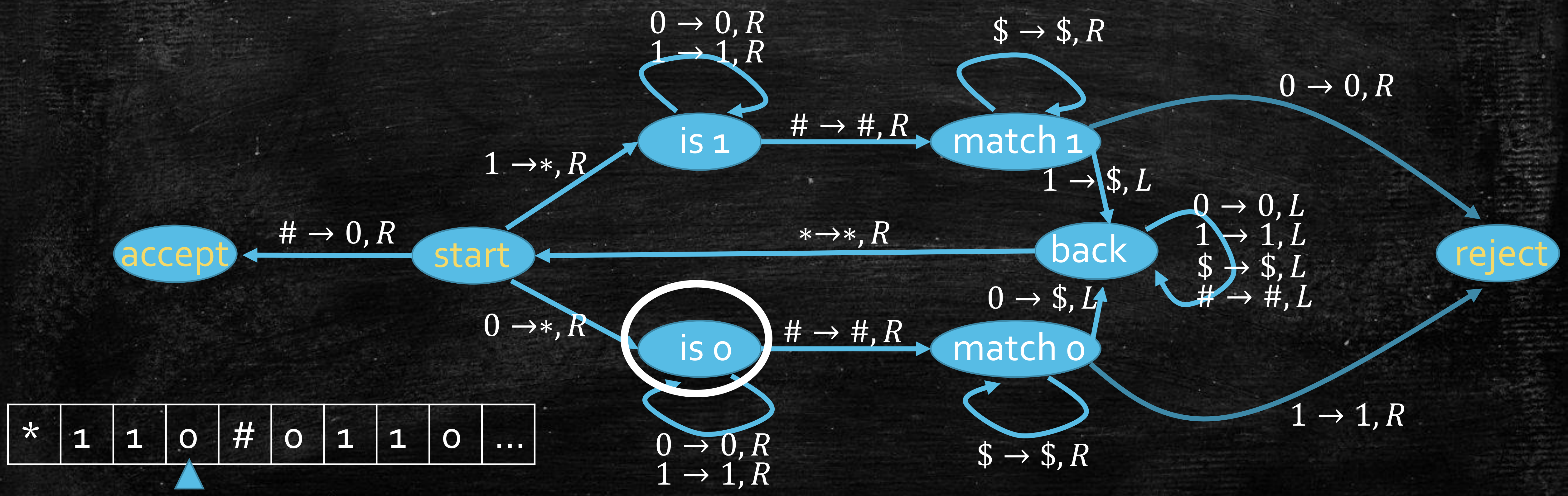
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	1	1	0	#	0	1	1	0	…

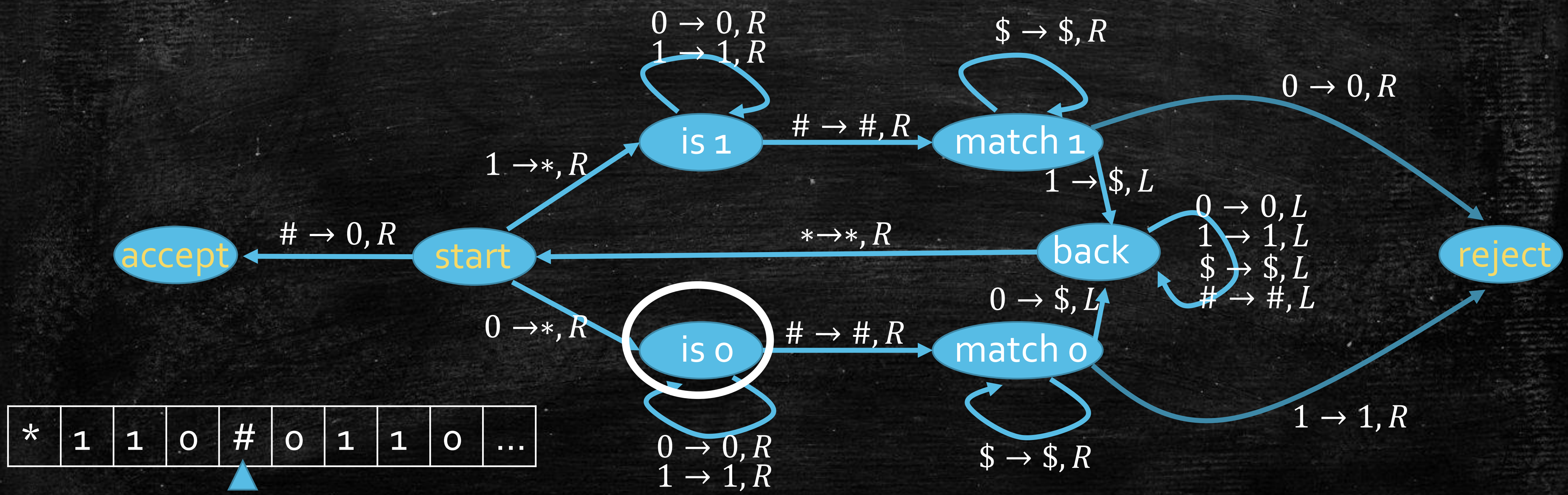
TM Example: Check if two strings are identical

- 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



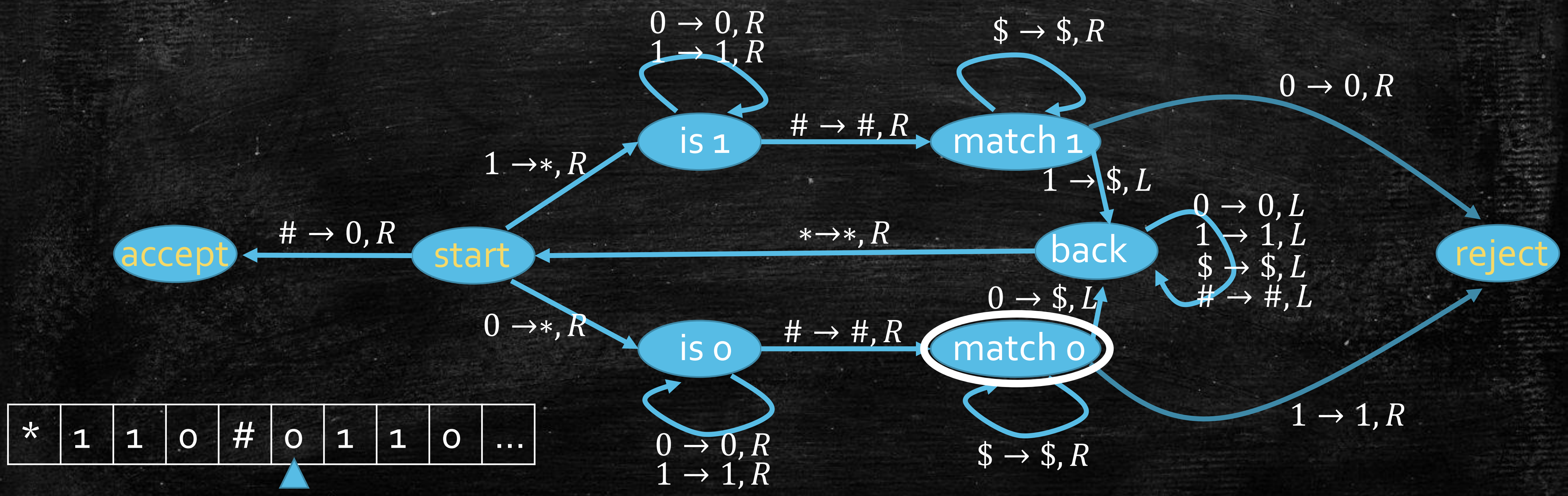
TM Example: Check if two strings are identical

- 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



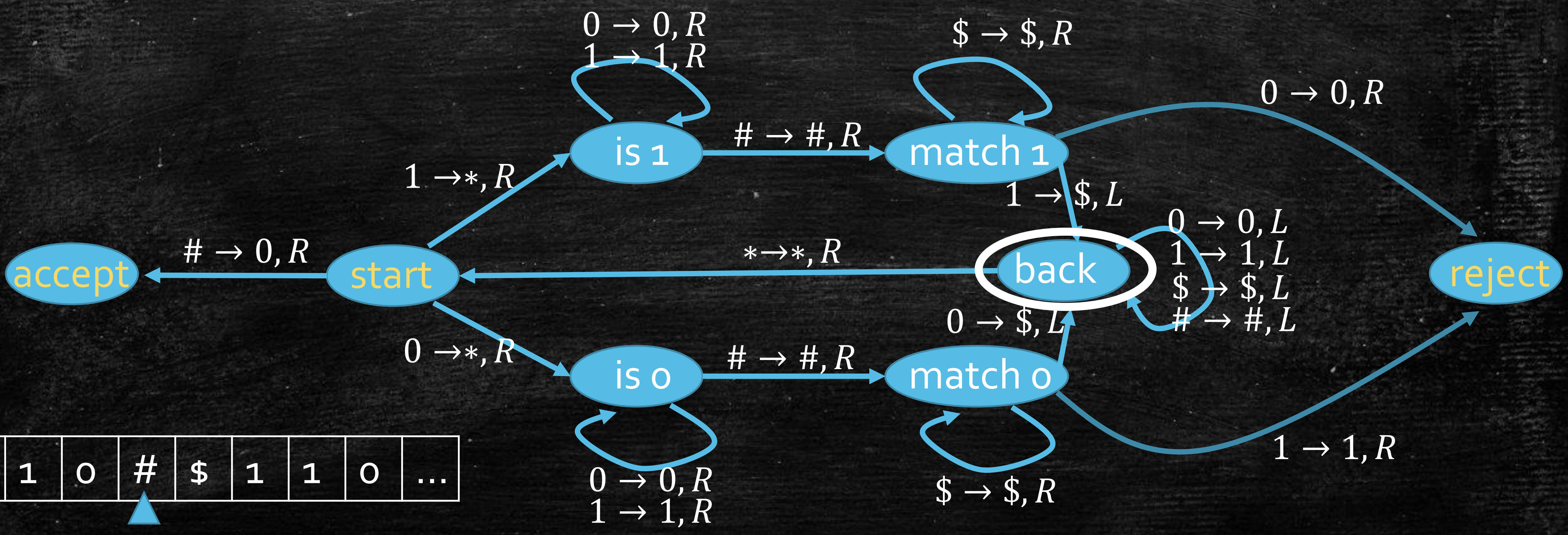
TM Example: Check if two strings are identical

- 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

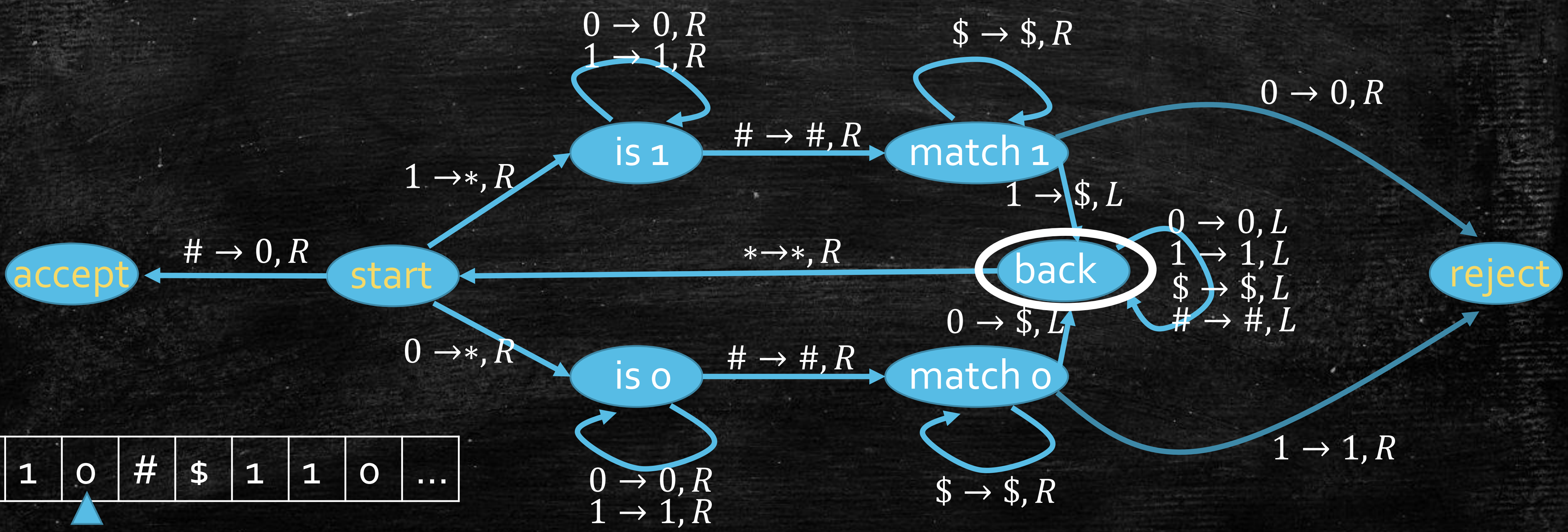
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	1	1	0	#	$	1	1	0	…

TM Example: Check if two strings are identical

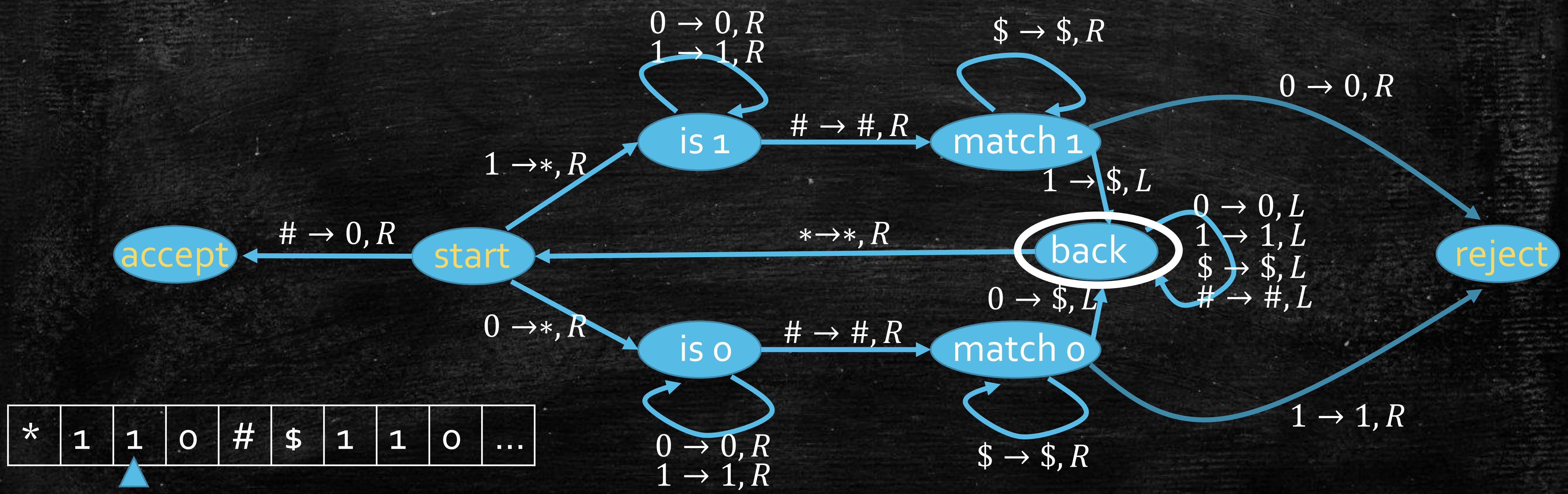
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	1	1	0	#	$	1	1	0	…

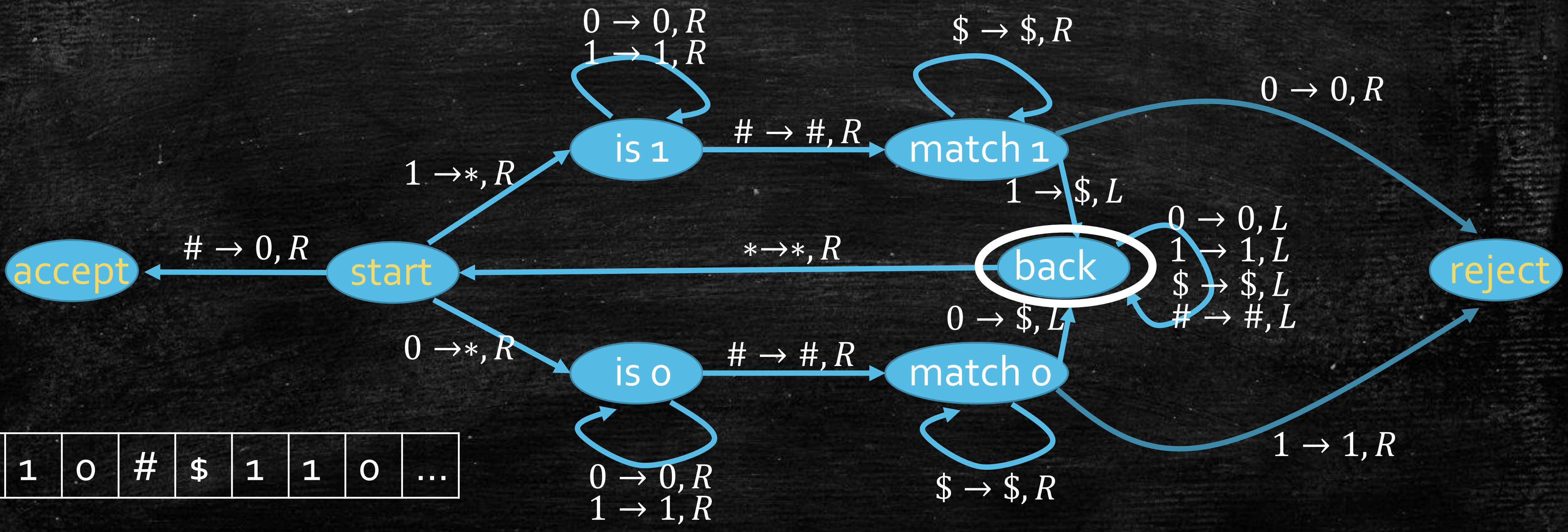
TM Example: Check if two strings are identical

- 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

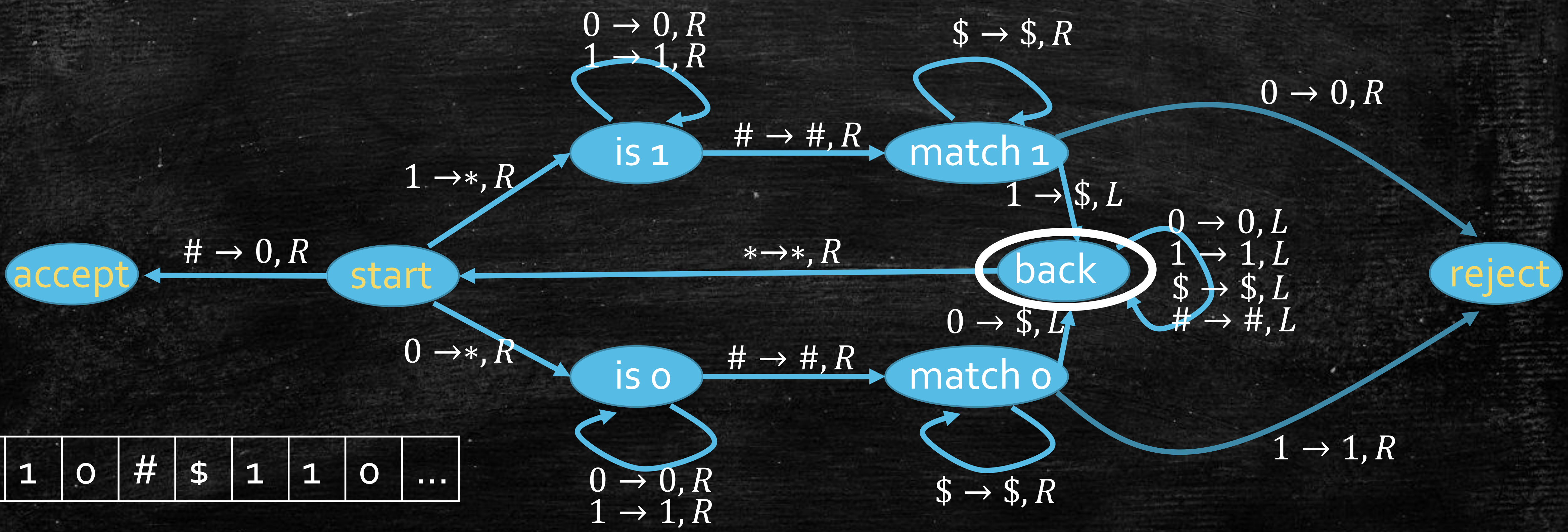
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	1	1	0	#	$	1	1	0	…

TM Example: Check if two strings are identical

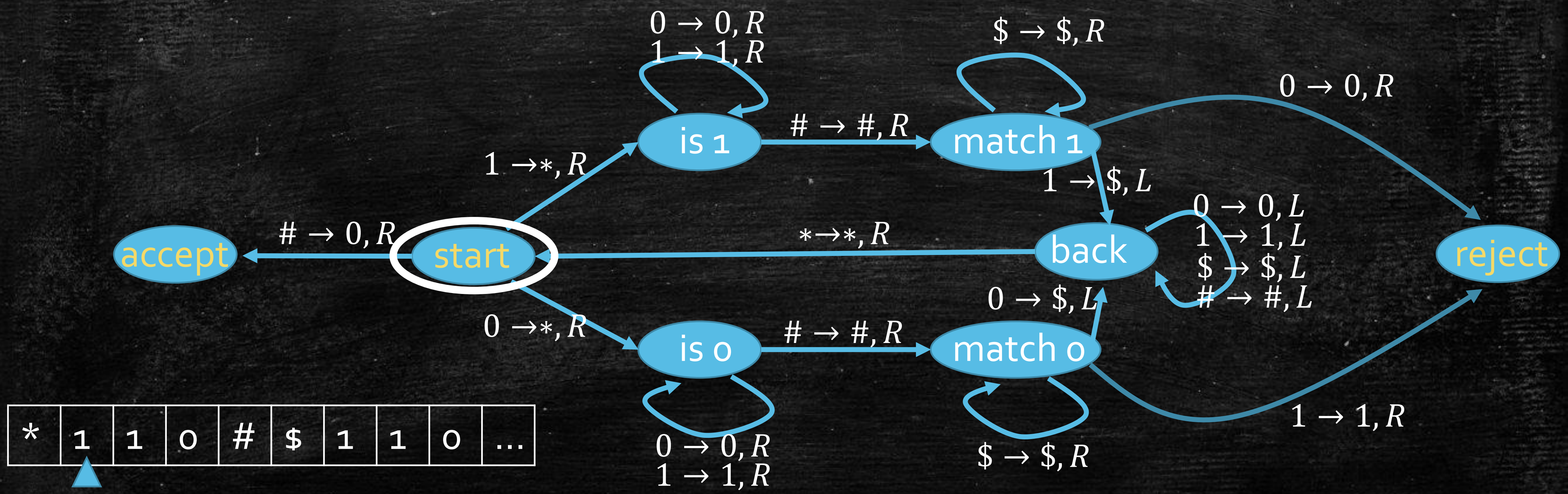
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	1	1	0	#	\$	1	1	0	...

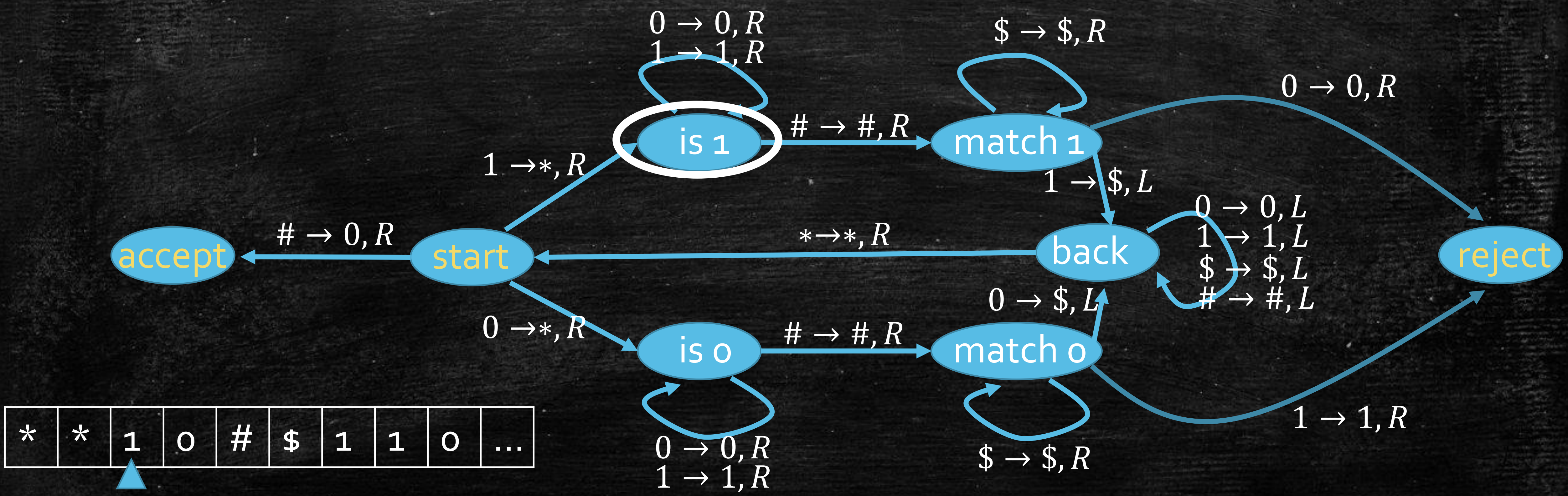
TM Example: Check if two strings are identical

- 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



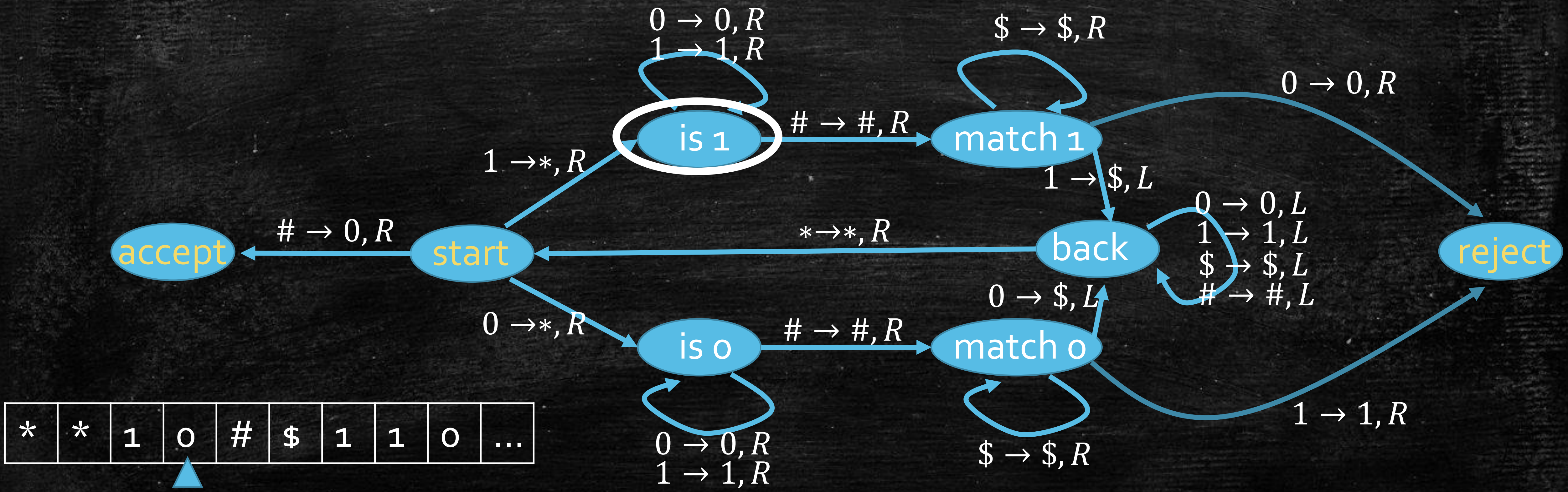
TM Example: Check if two strings are identical

- 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



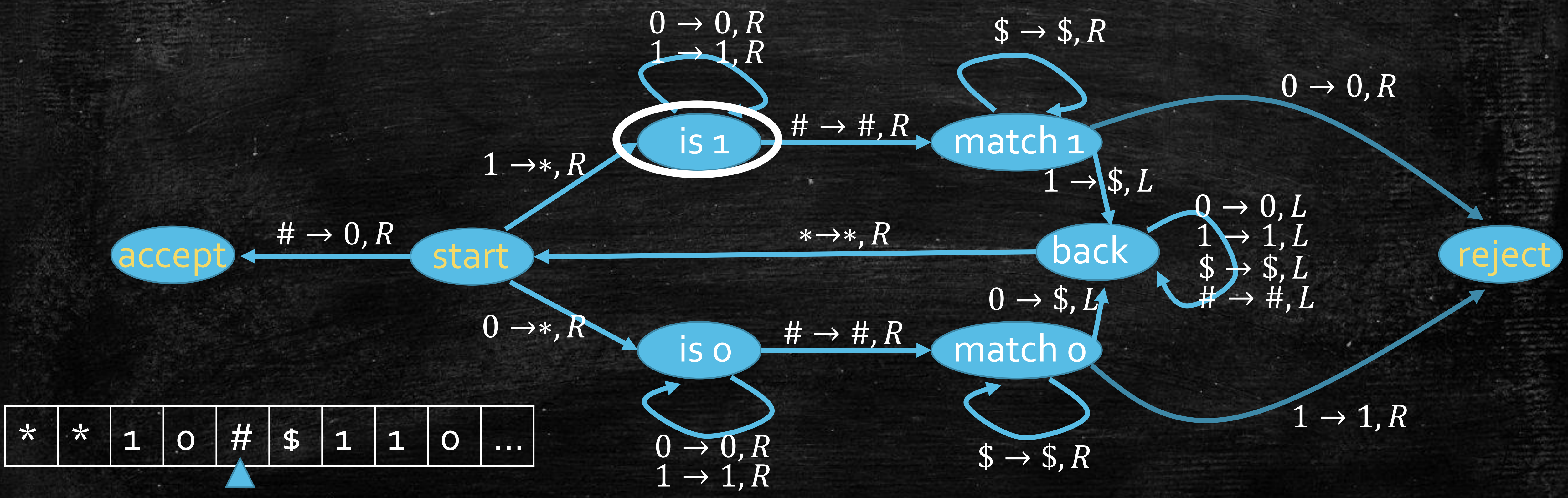
TM Example: Check if two strings are identical

- 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



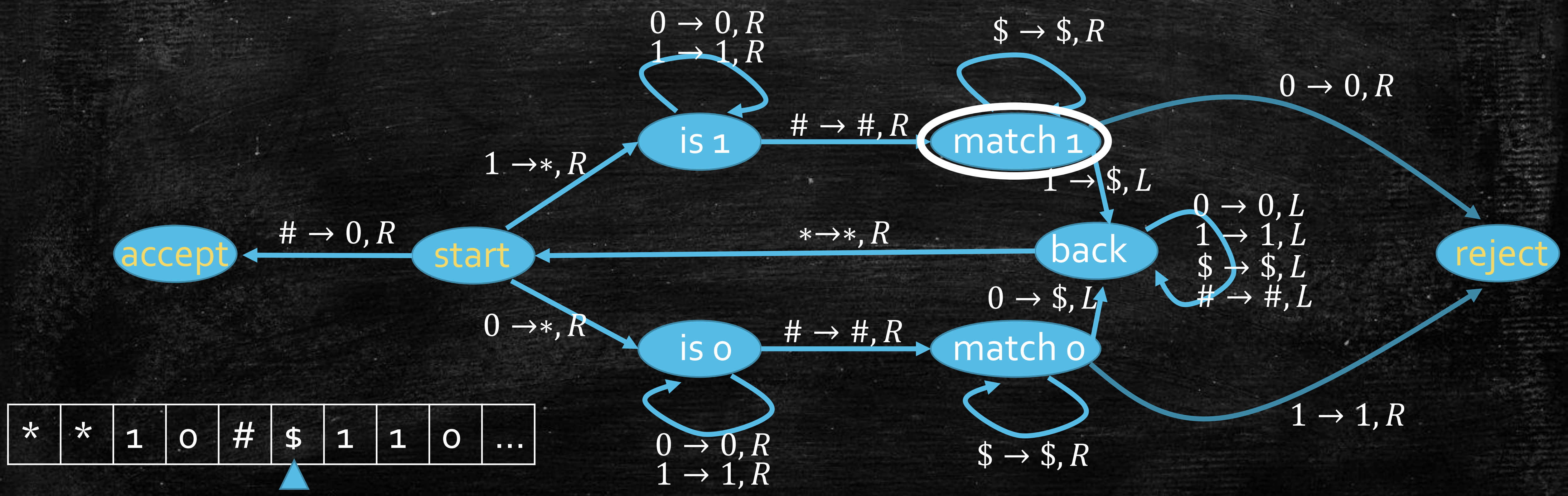
TM Example: Check if two strings are identical

- 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



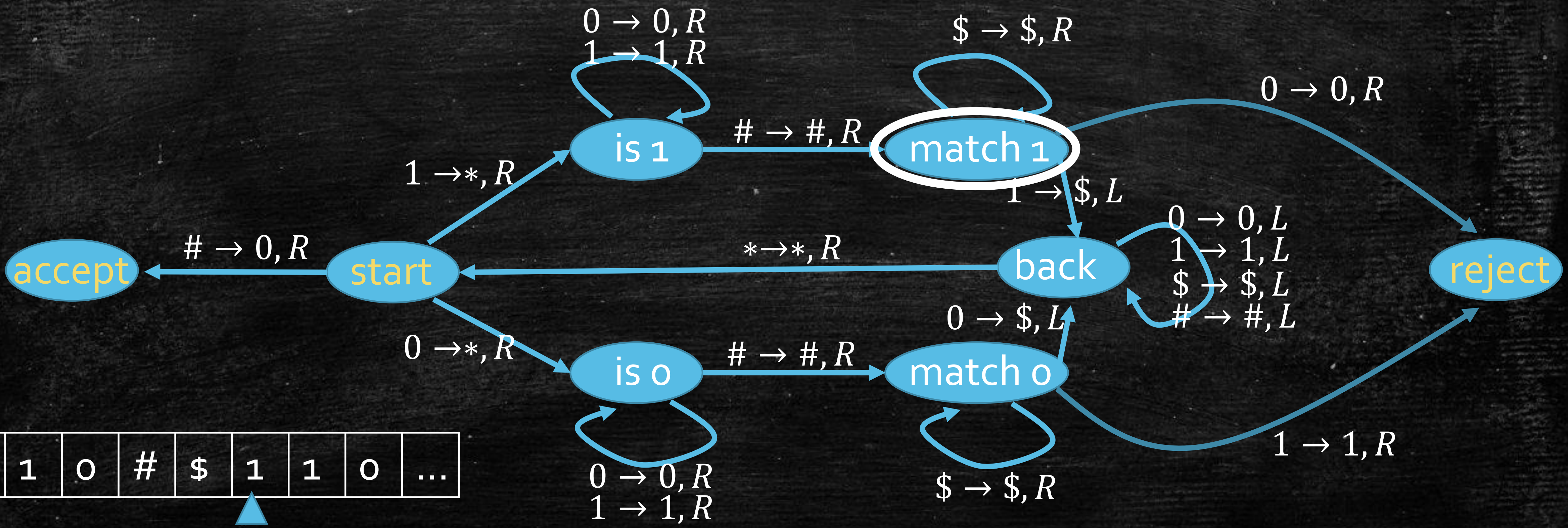
TM Example: Check if two strings are identical

- 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

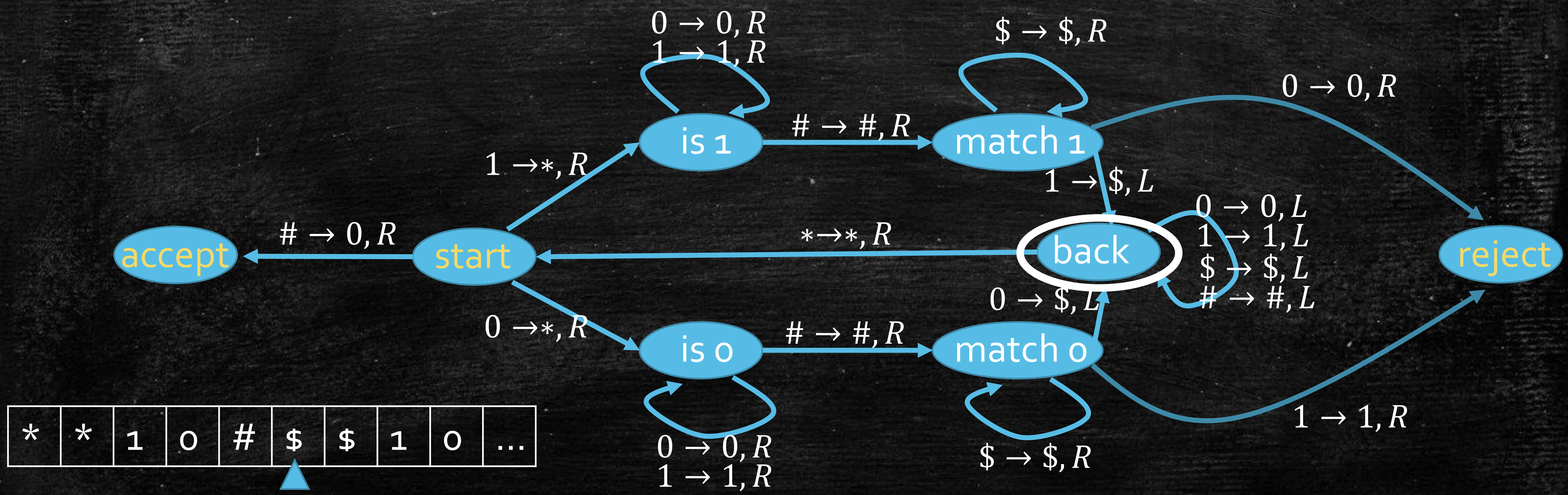
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	*	1	0	#	$	1	1	0	…

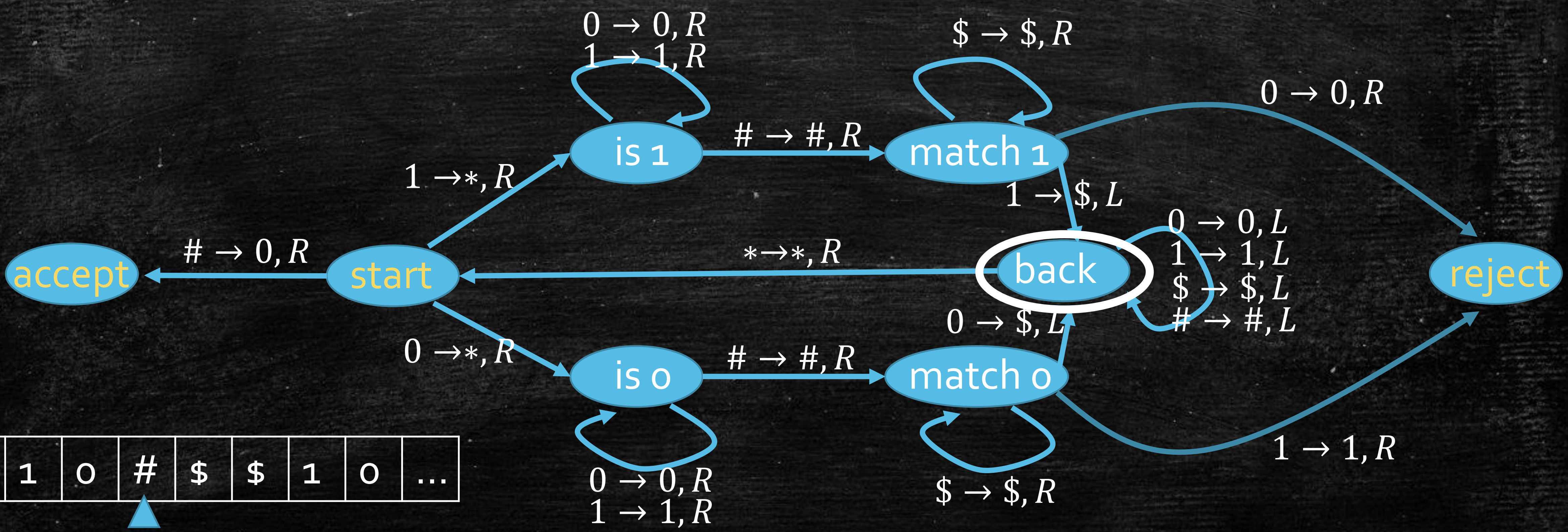
TM Example: Check if two strings are identical

- 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

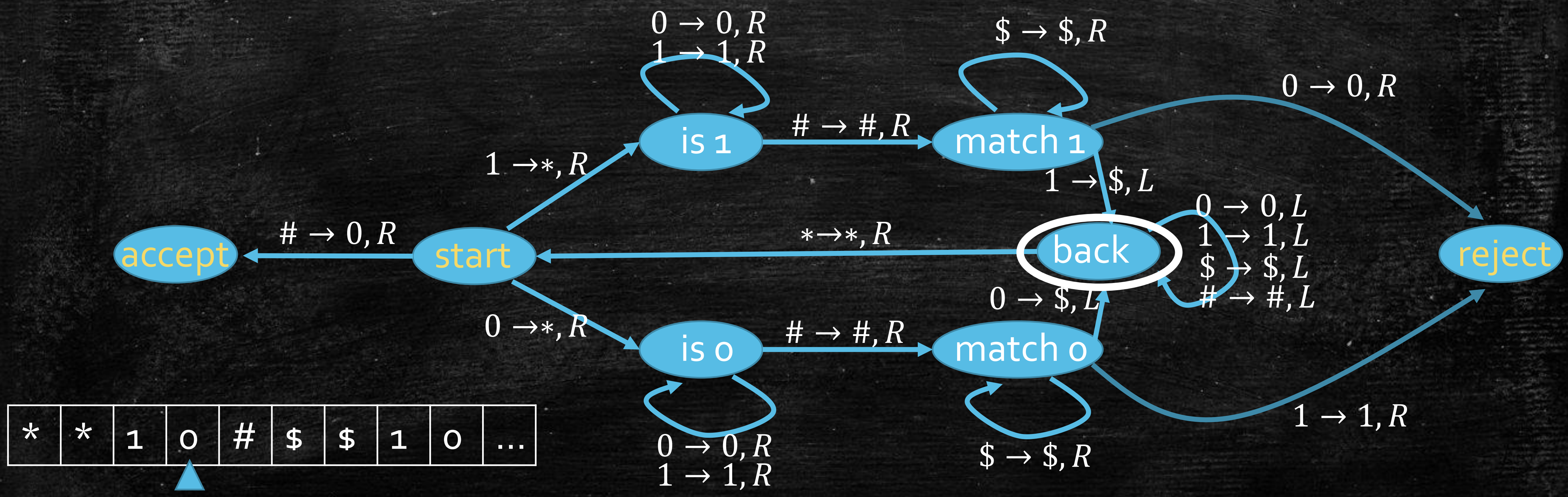
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	*	1	0	#	\$	\$	1	0	…

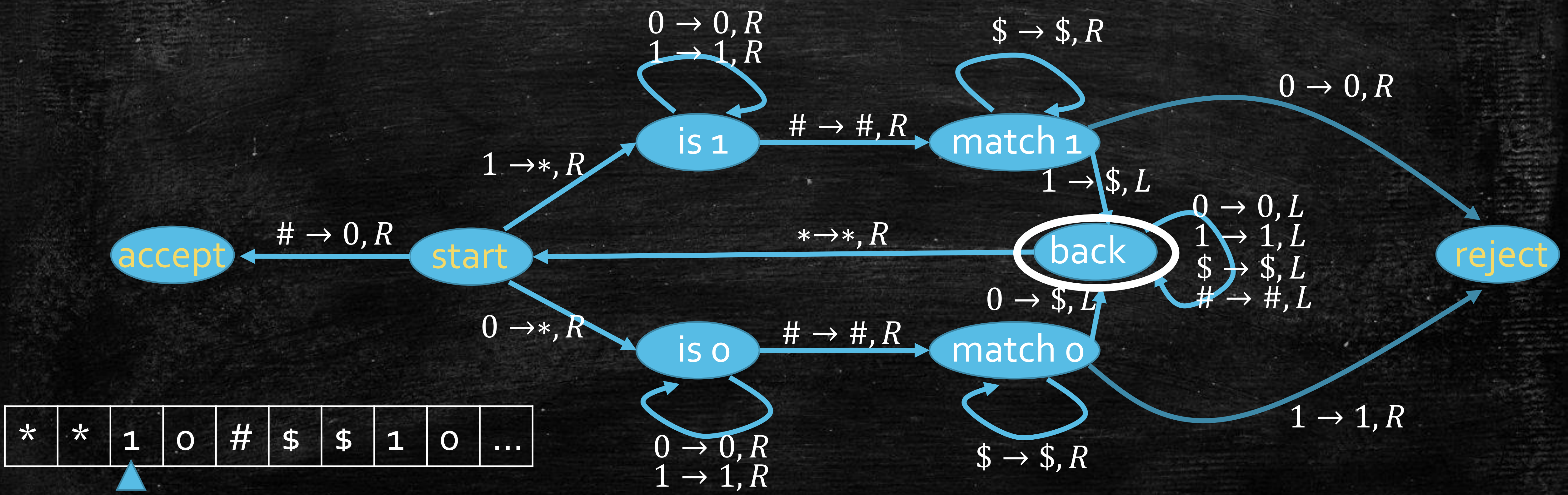
TM Example: Check if two strings are identical

- 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



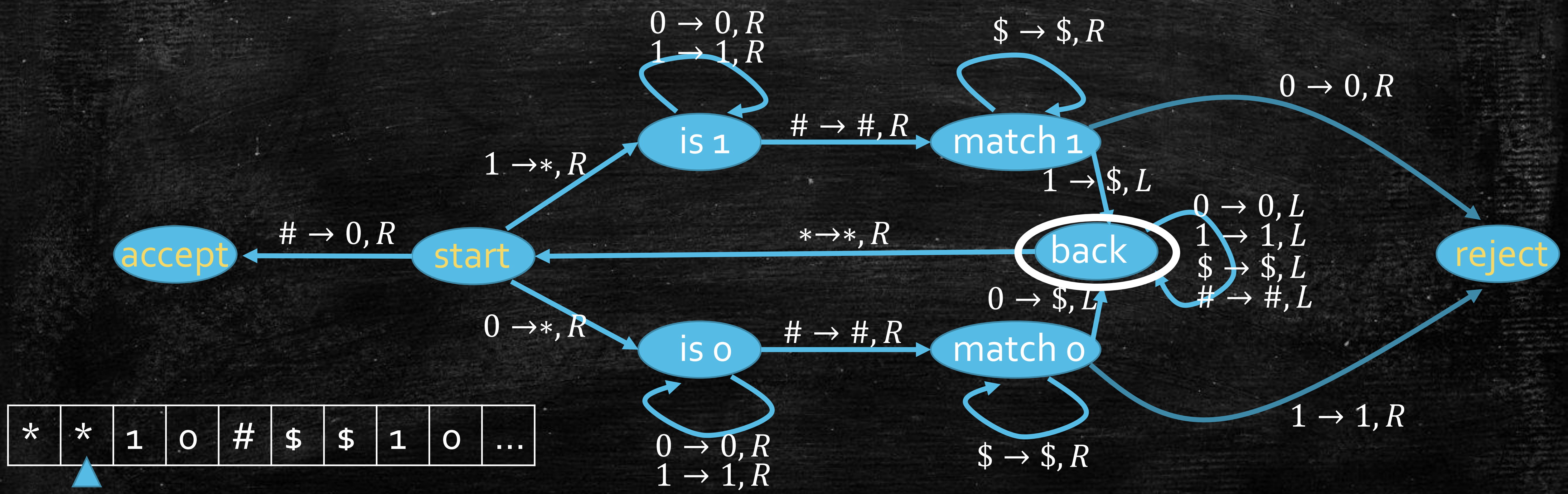
TM Example: Check if two strings are identical

- 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



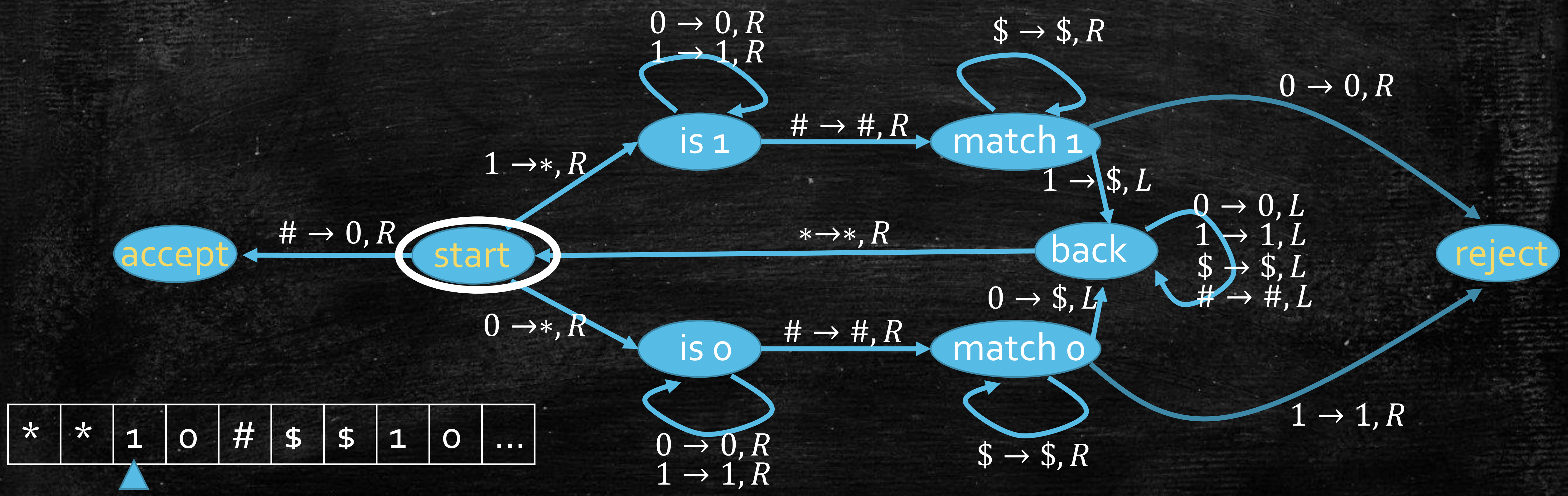
TM Example: Check if two strings are identical

- 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



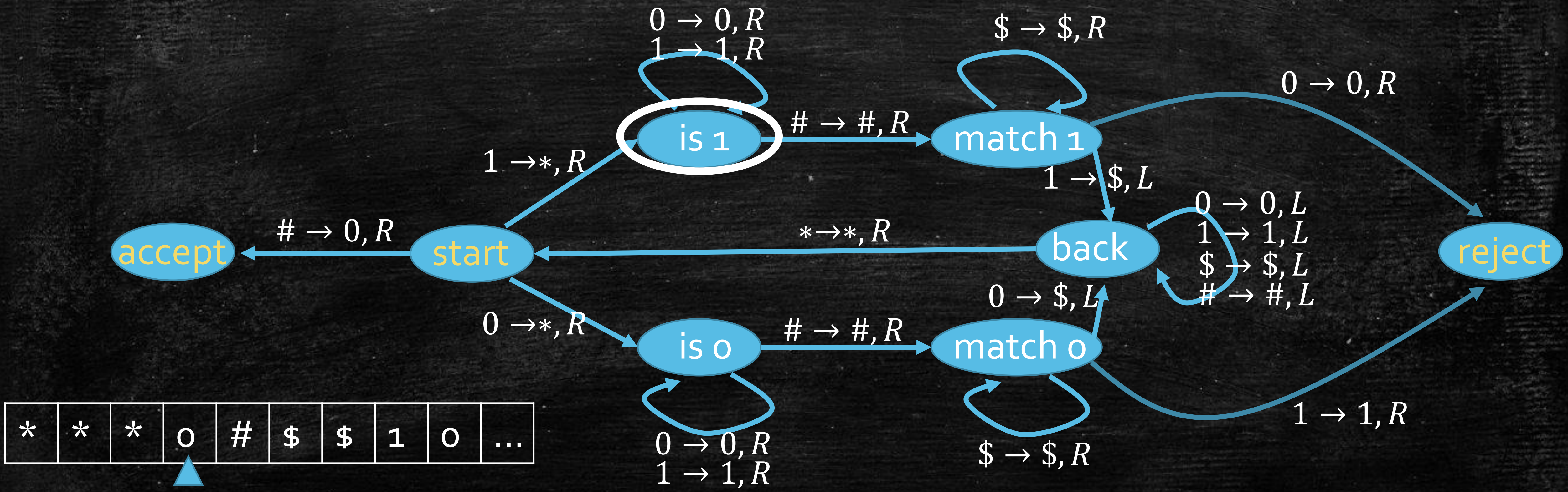
TM Example: Check if two strings are identical

- 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



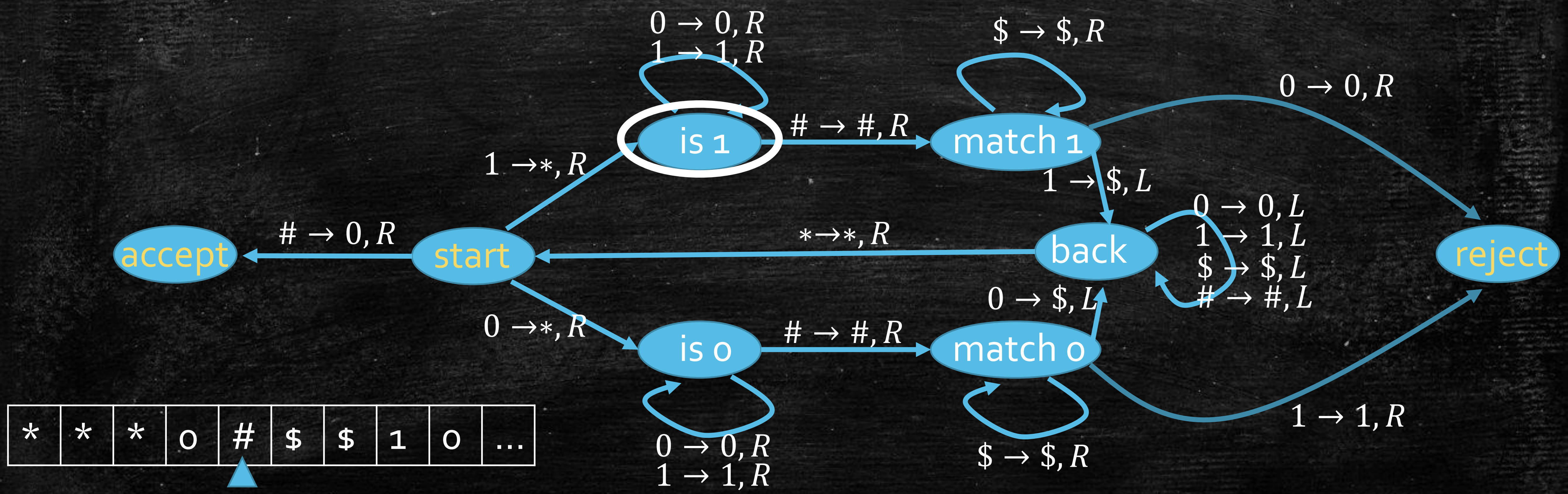
TM Example: Check if two strings are identical

- 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



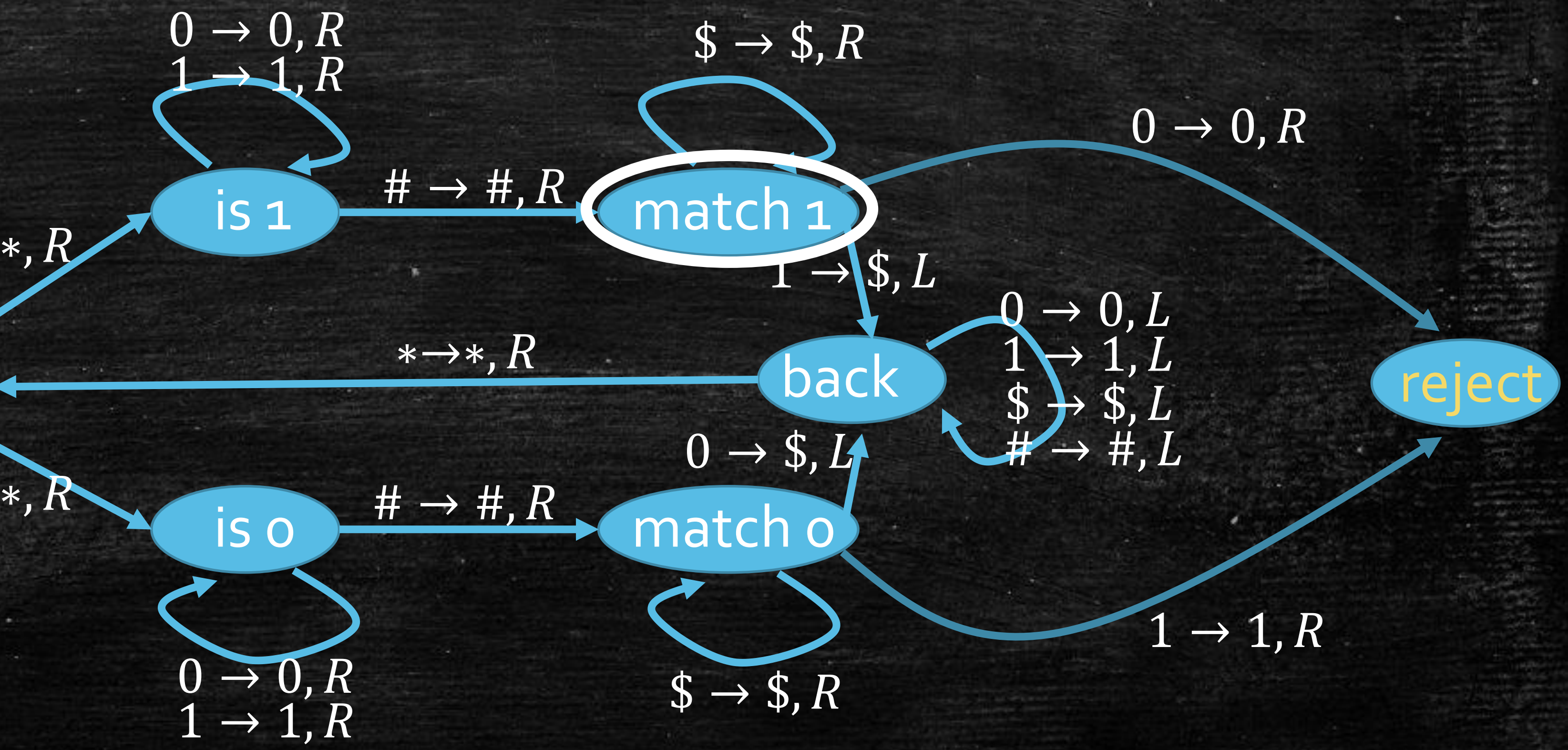
TM Example: Check if two strings are identical

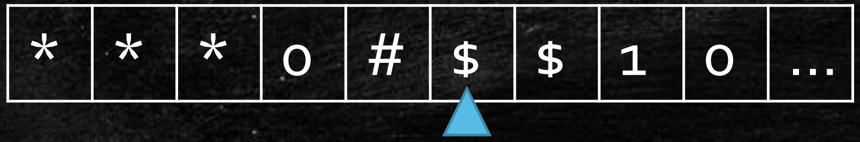
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



TM Example: Check if two strings are identical

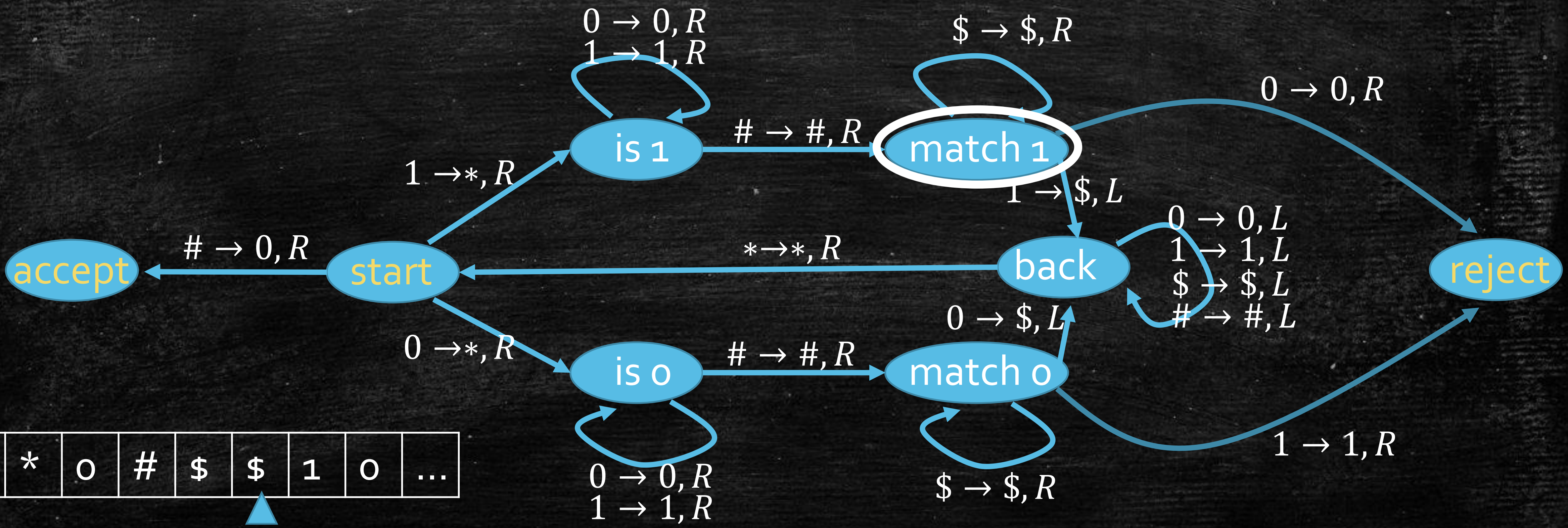
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical





TM Example: Check if two strings are identical

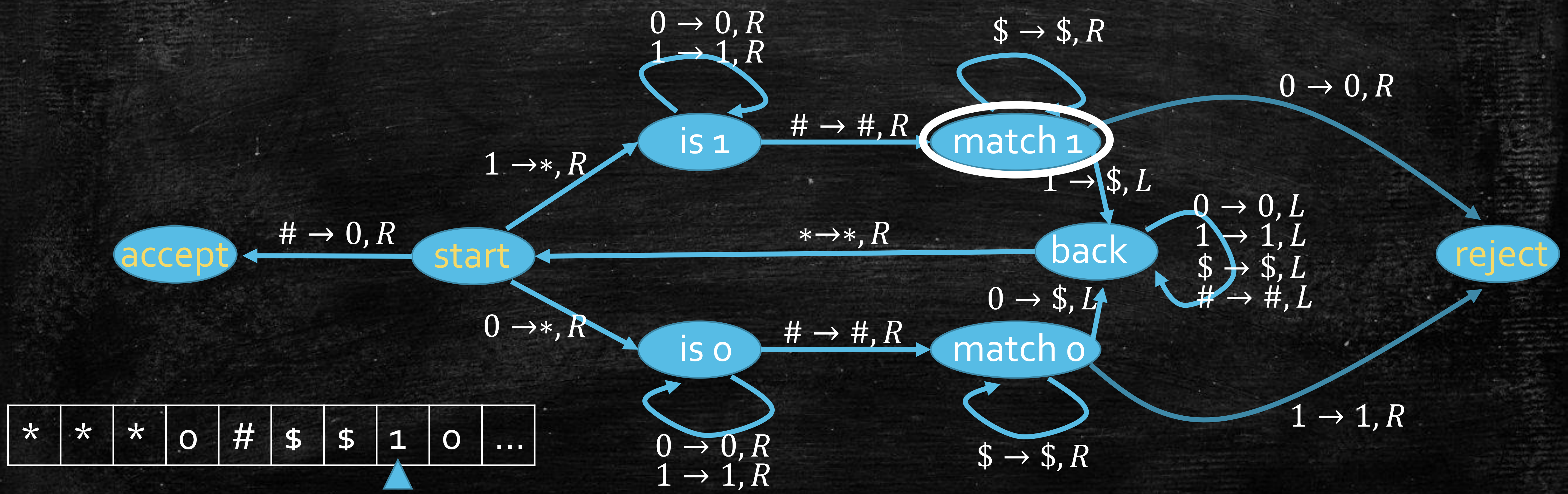
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	*	*	0	#	$	$	1	0	…

TM Example: Check if two strings are identical

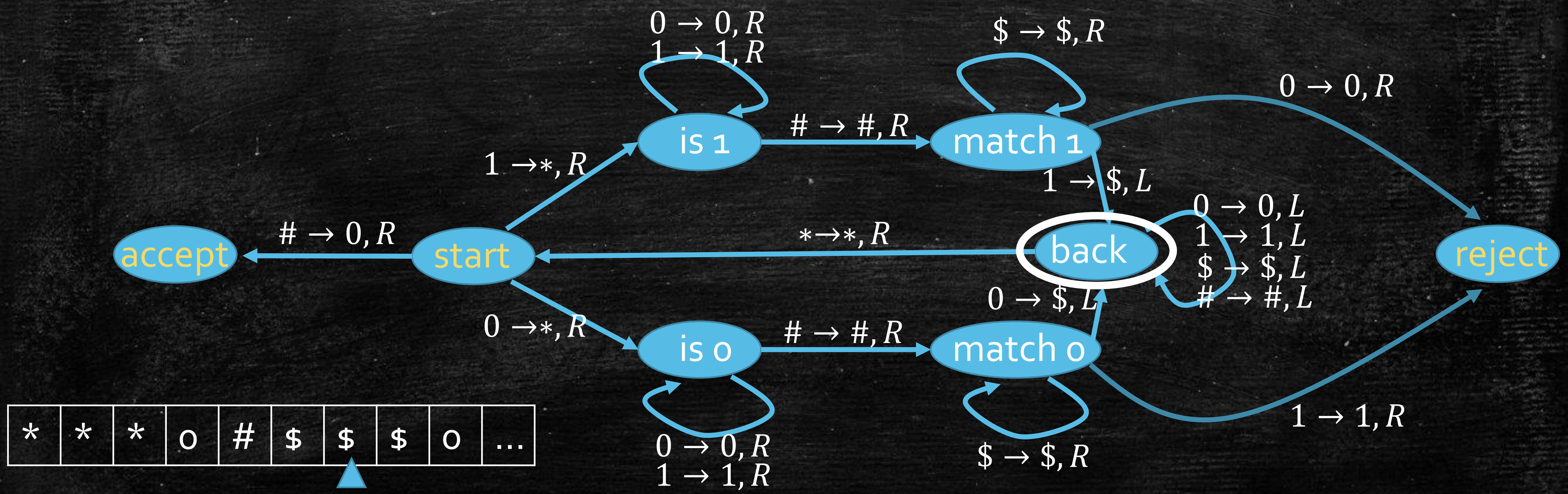
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



TM Example: Check if two strings are identical

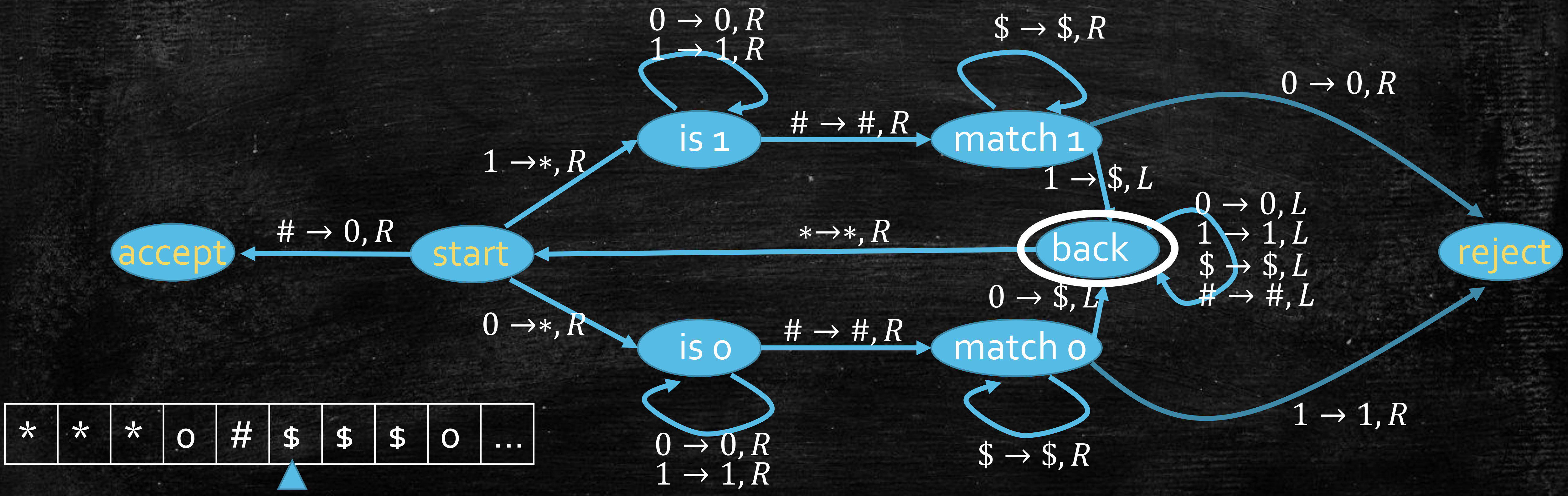
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet

- Decide if the binary strings x and y is identical



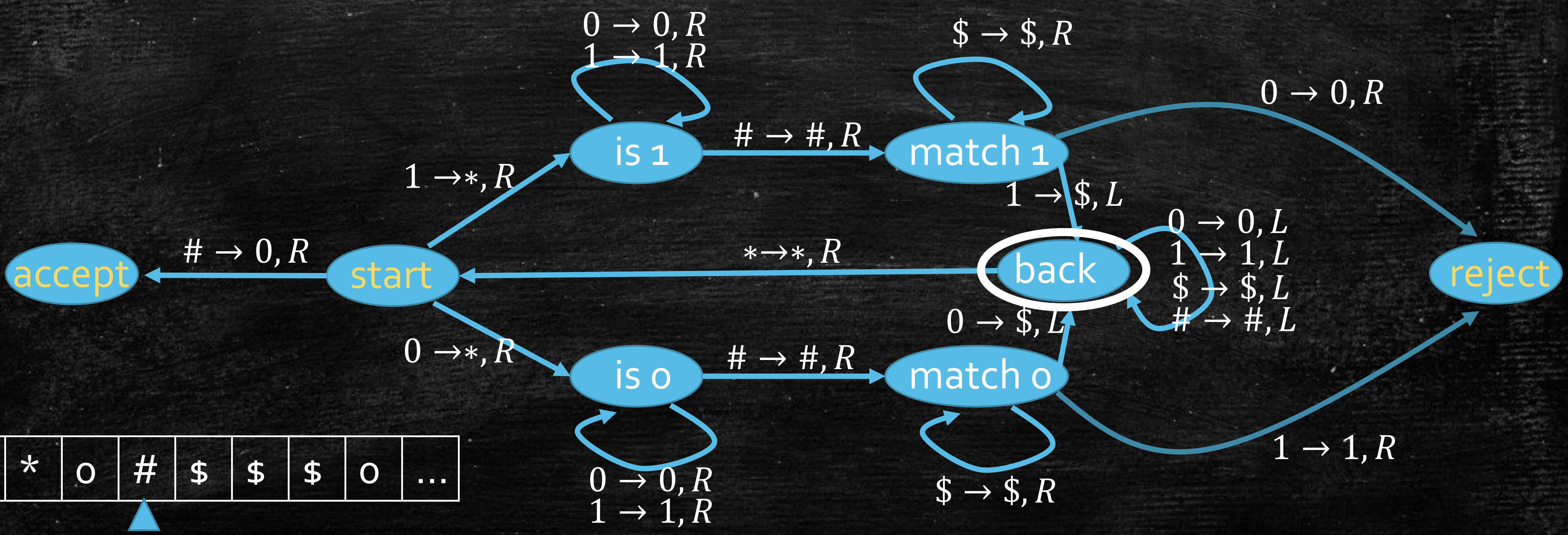
TM Example: Check if two strings are identical

- 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

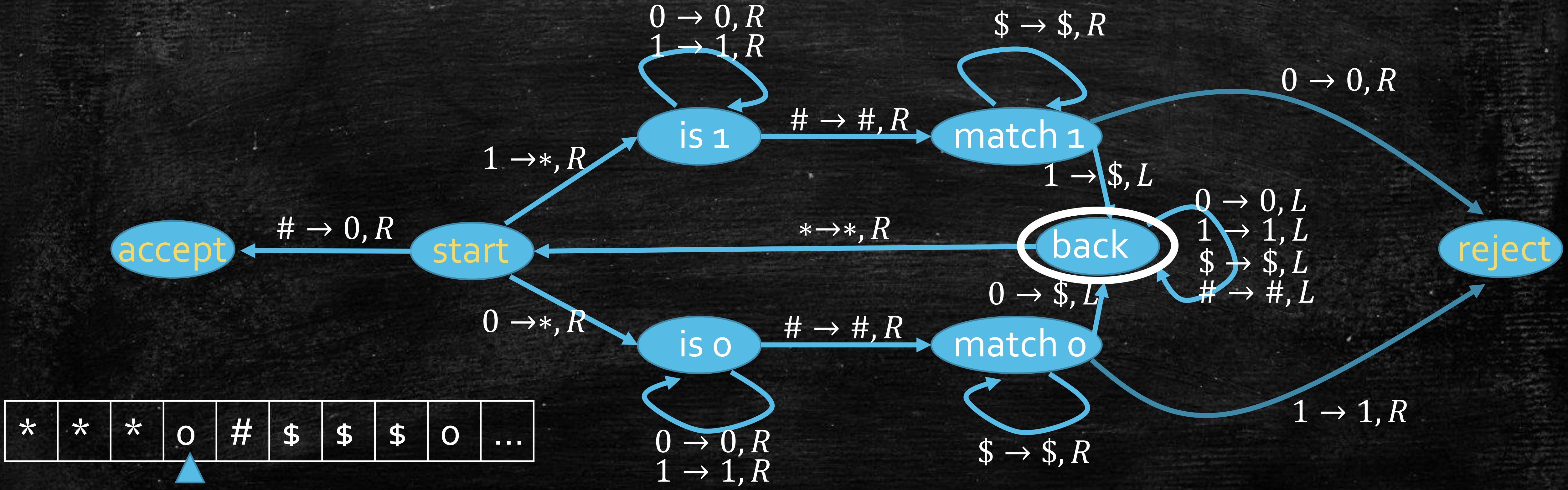
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	*	*	0	#	\$	\$	\$	0	…

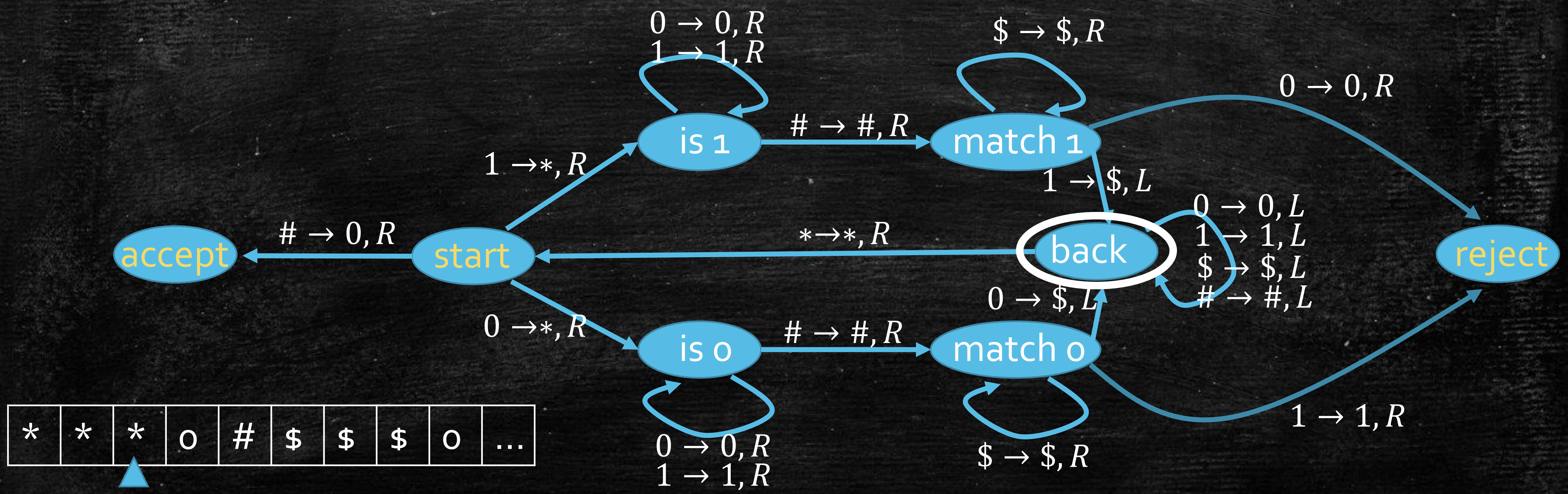
TM Example: Check if two strings are identical

- 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



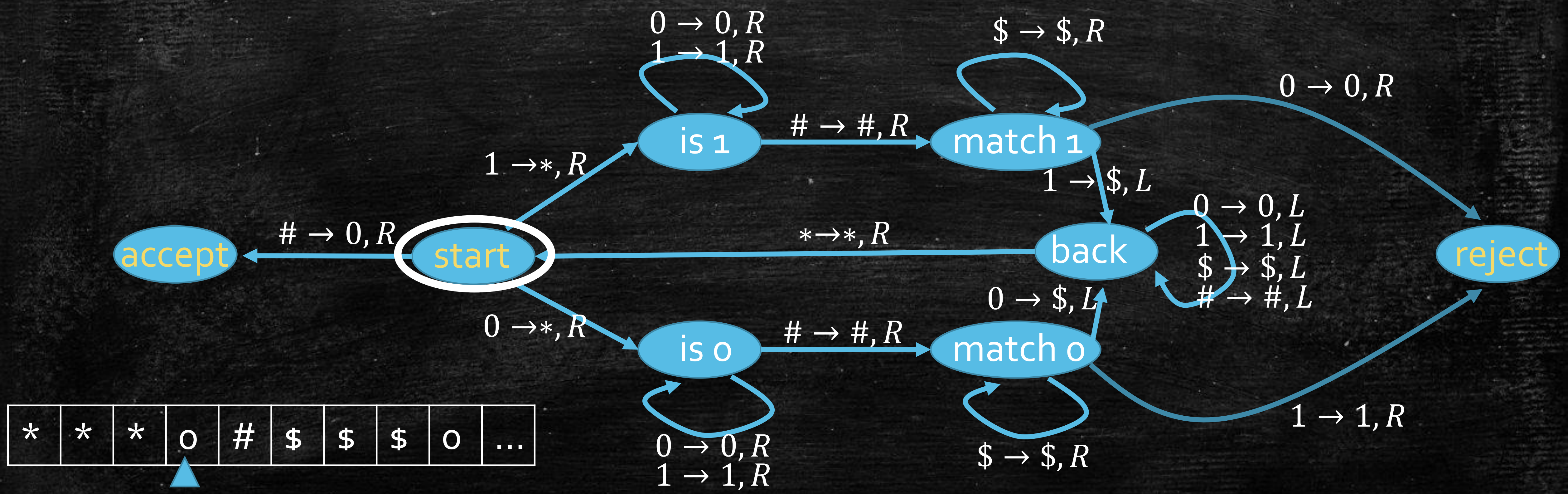
TM Example: Check if two strings are identical

- 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



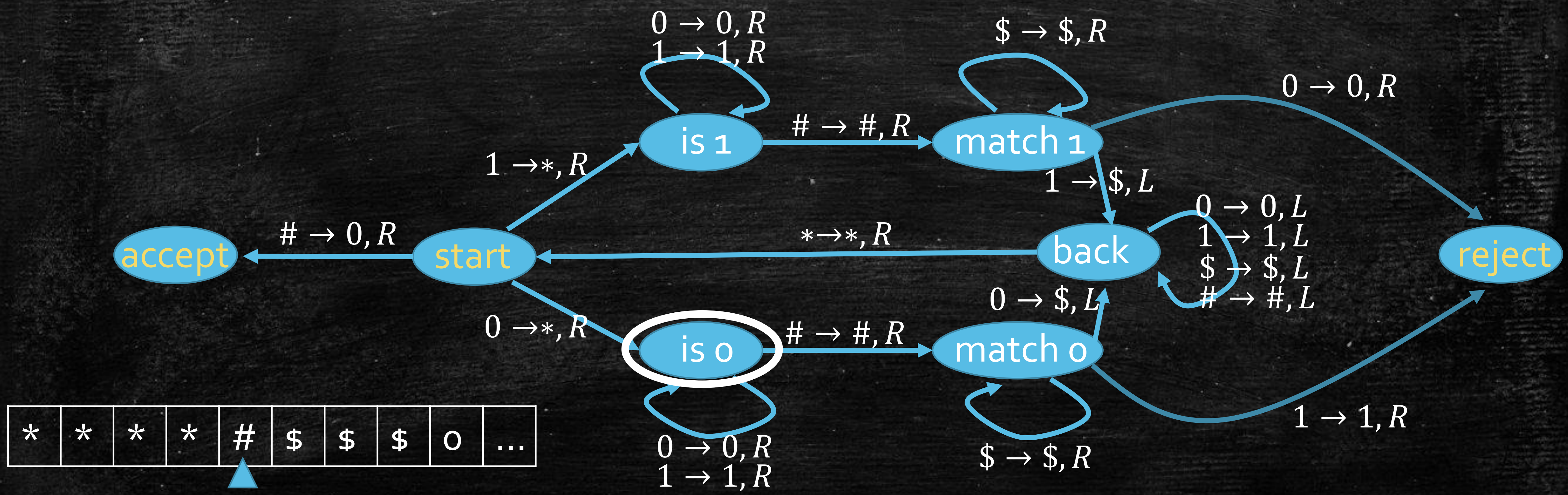
TM Example: Check if two strings are identical

- 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



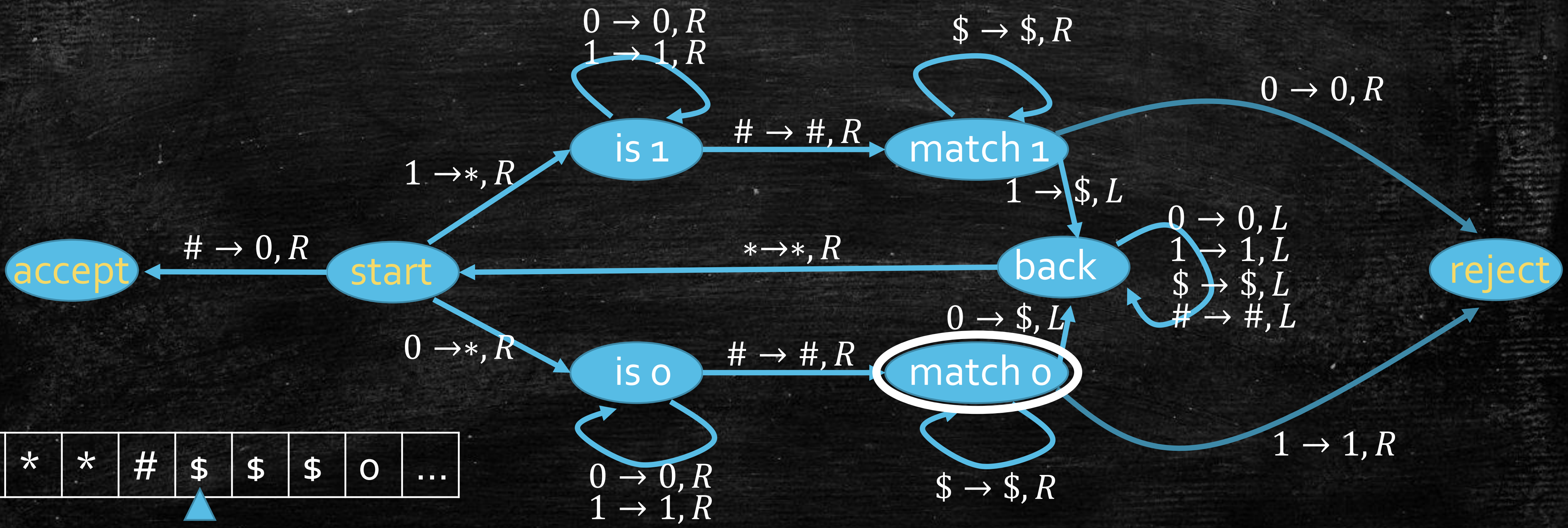
TM Example: Check if two strings are identical

- 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

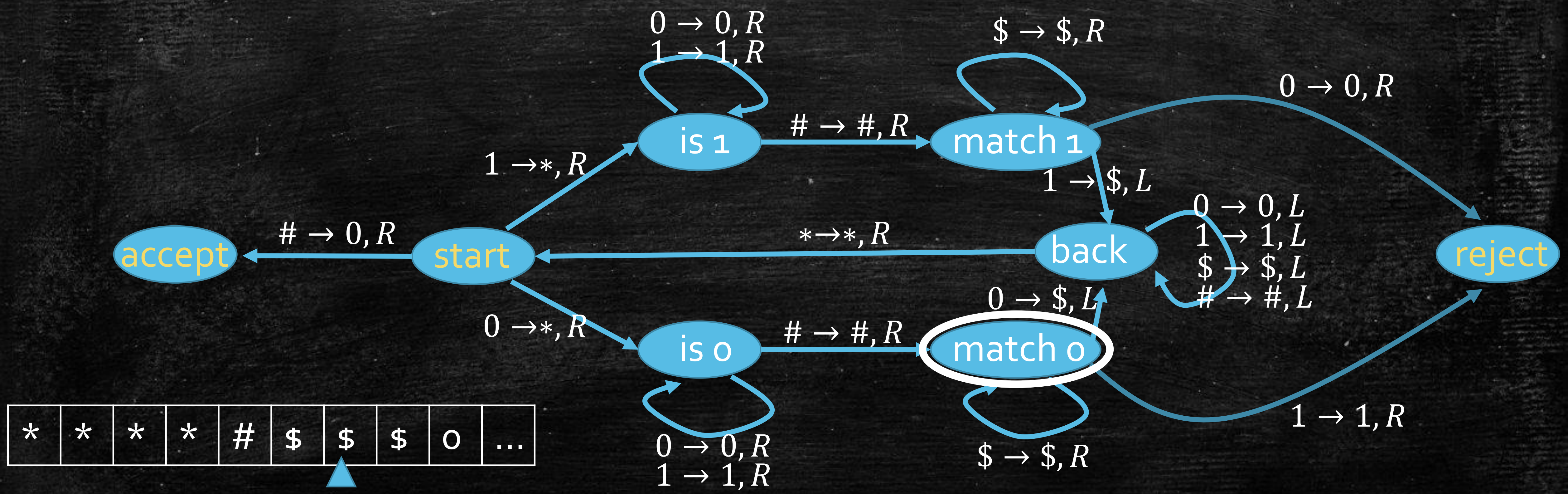
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	*	*	*	#	$	$	$	0	…

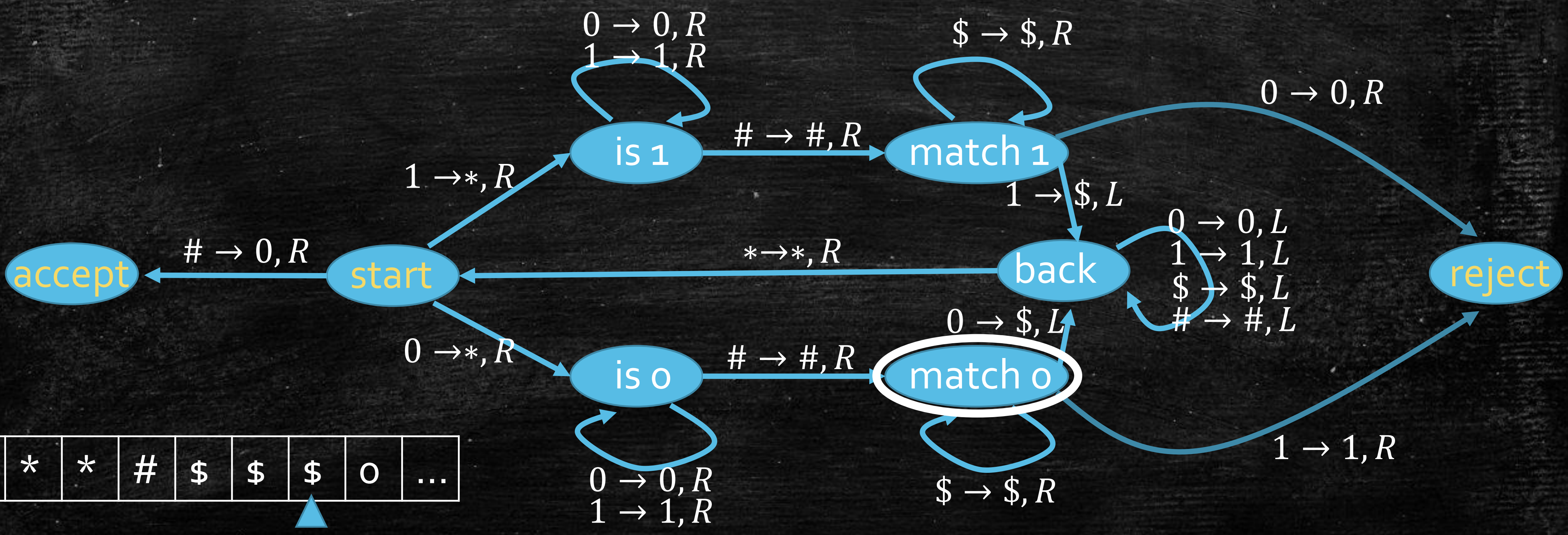
TM Example: Check if two strings are identical

- 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

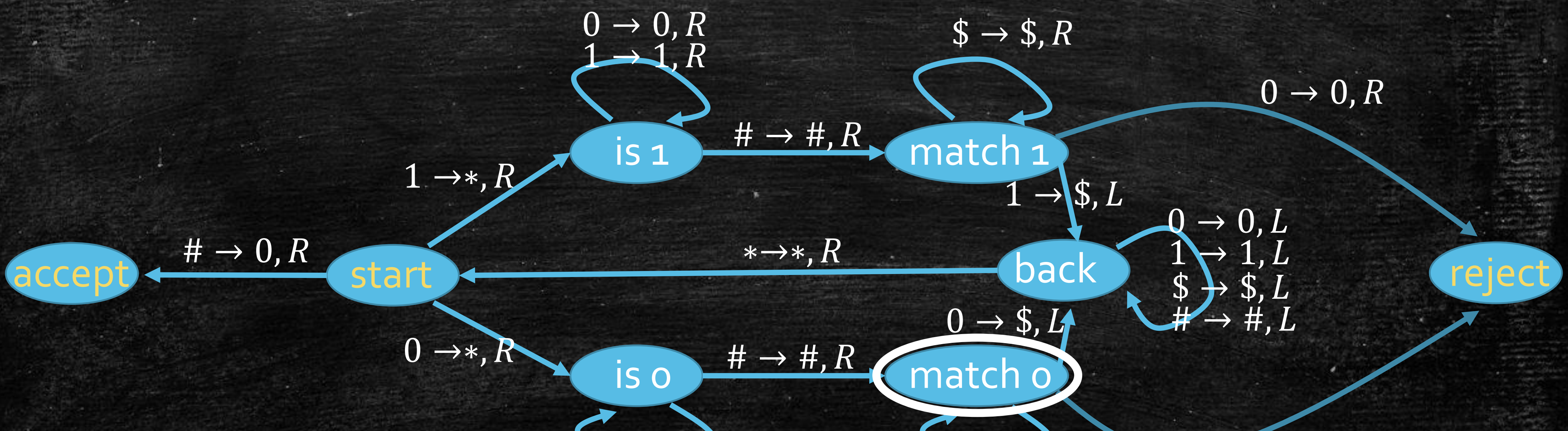
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	*	*	*	#	$	$	$	0	…

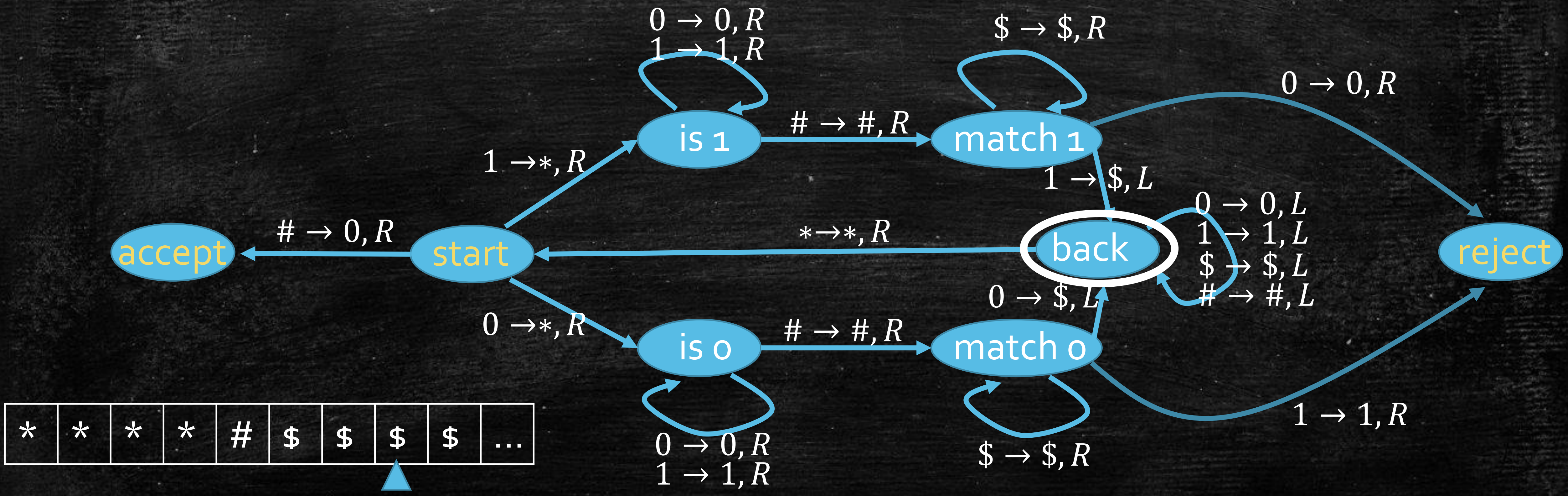
TM Example: Check if two strings are identical

- 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



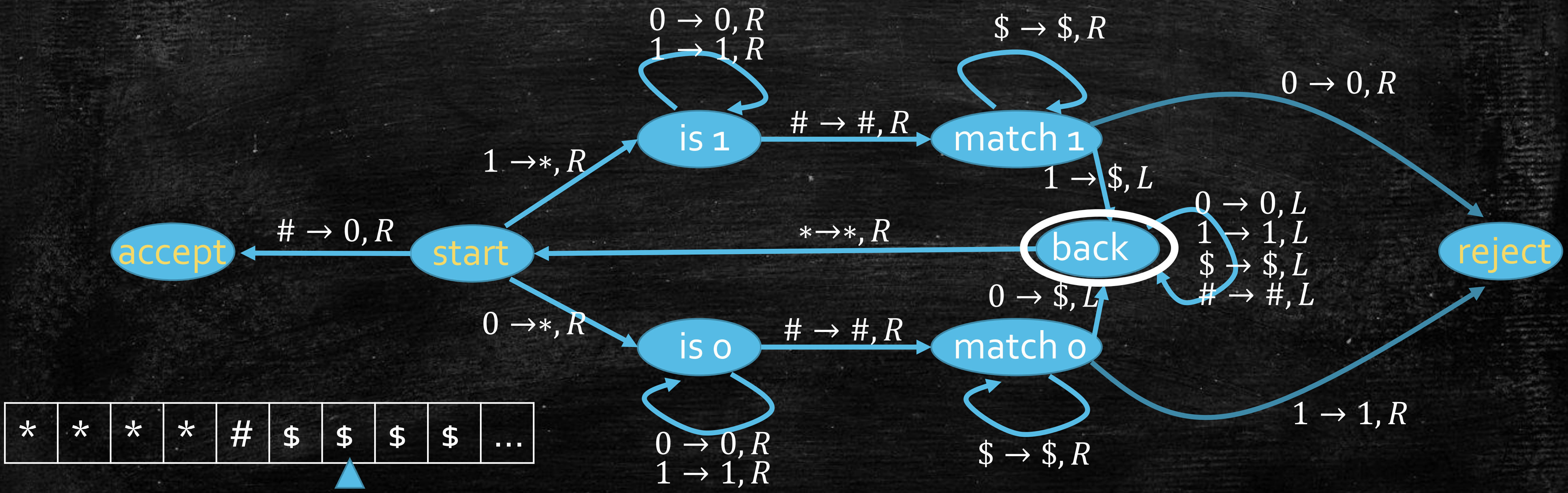
TM Example: Check if two strings are identical

- 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



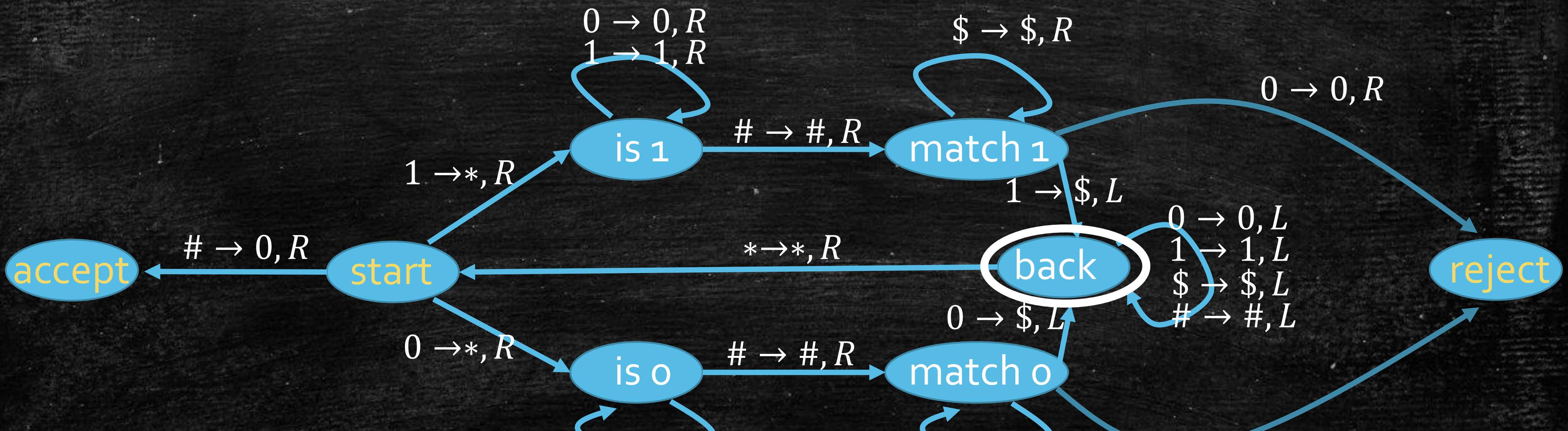
TM Example: Check if two strings are identical

- 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

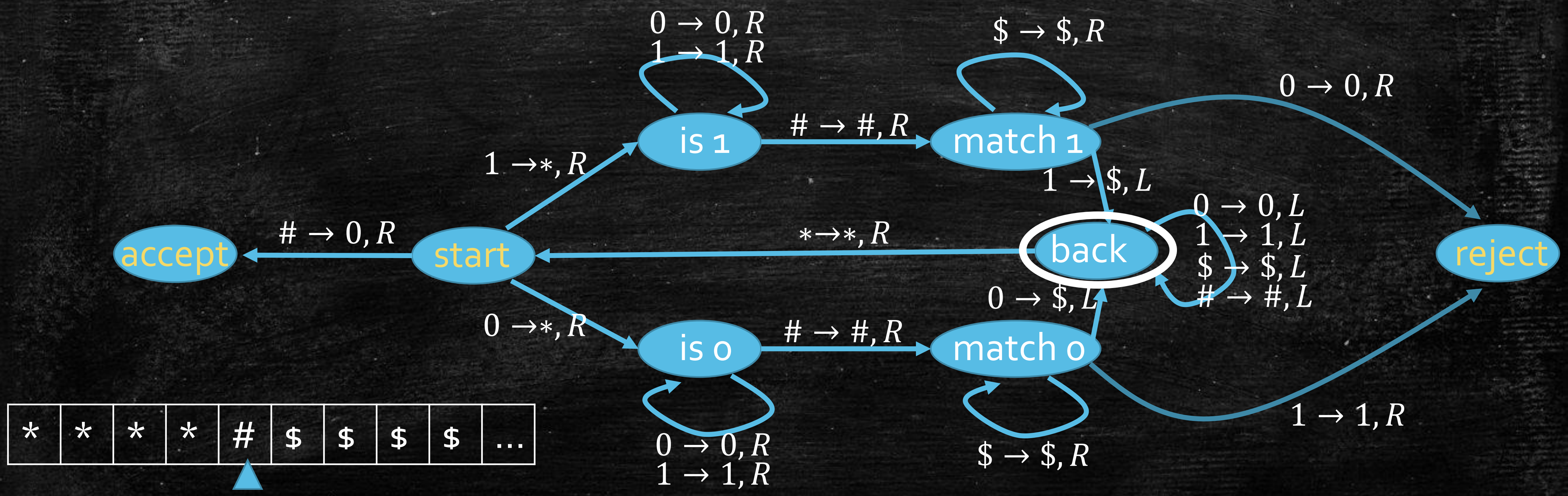
- Input: a string of format $x\#y$ where $x, y \in \{0,1\}^n$ and $\# \in \Sigma$ is a special separating alphabet
- Decide if the binary strings x and y is identical



*	*	*	*	#	\$	\$	\$	\$	…

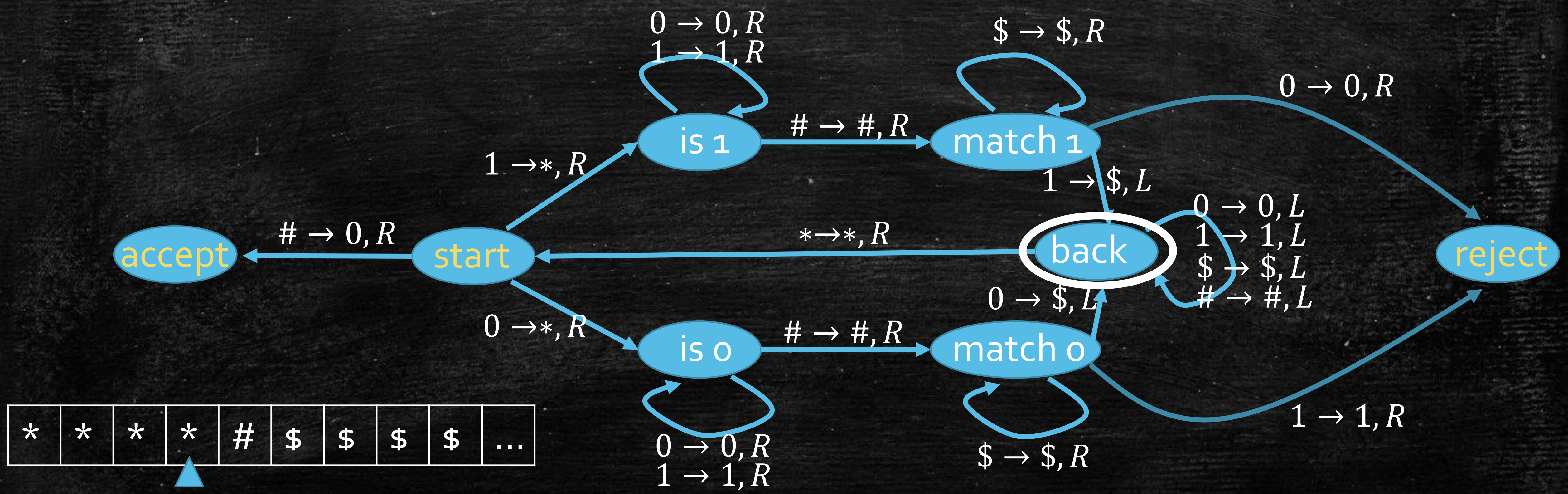
TM Example: Check if two strings are identical

- 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



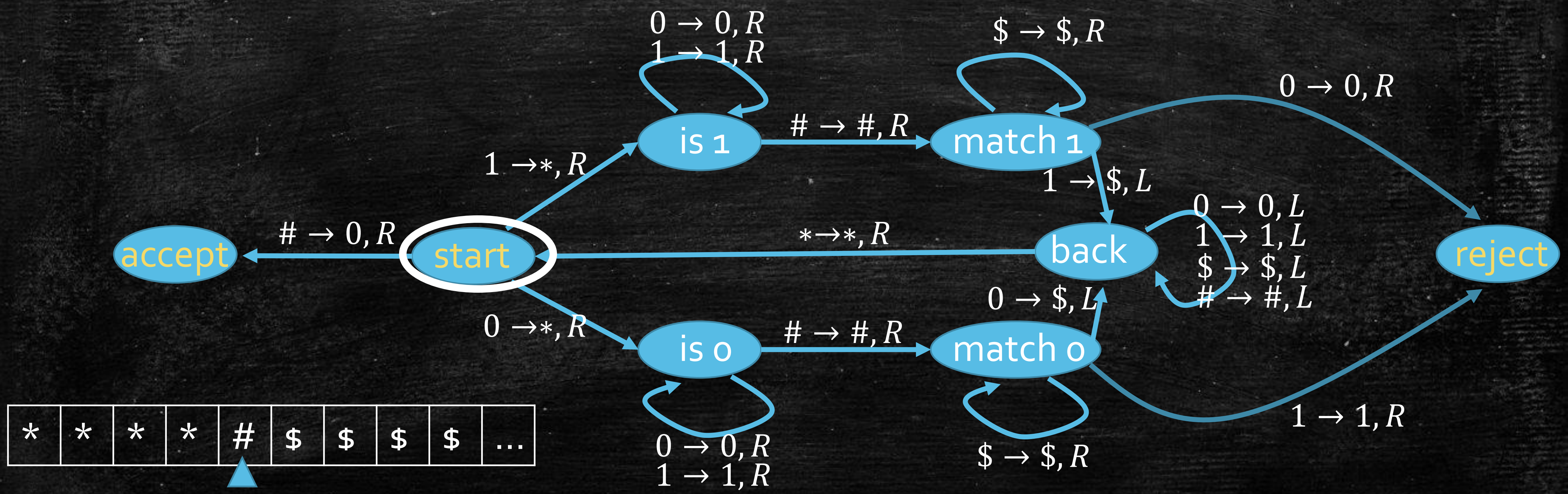
TM Example: Check if two strings are identical

- 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



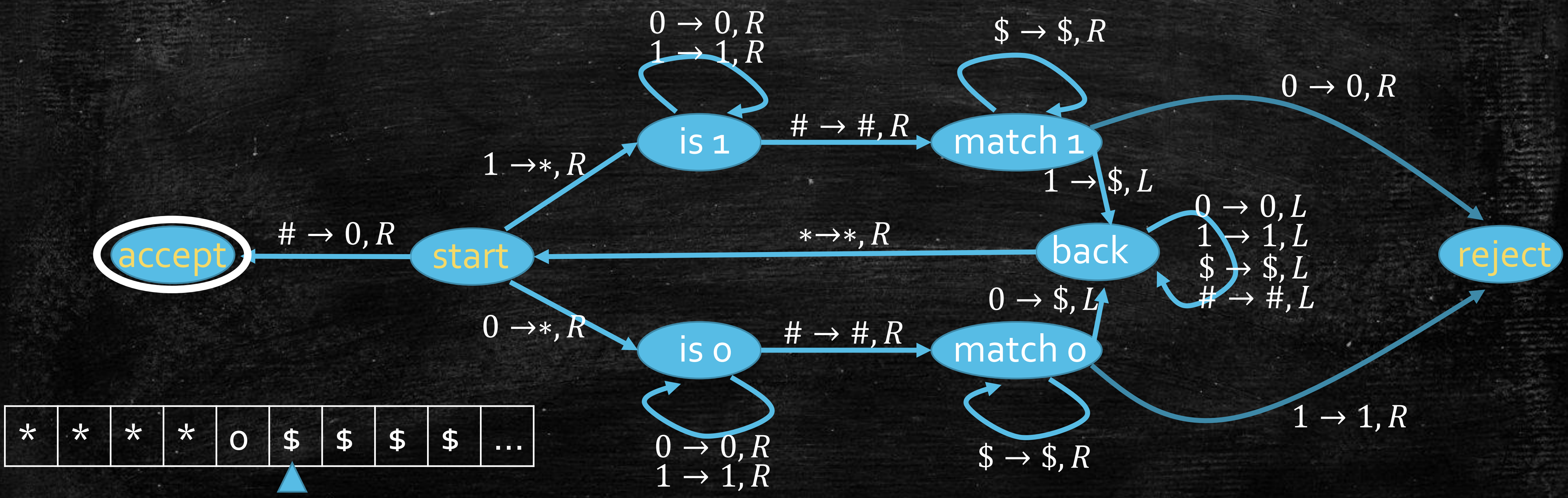
TM Example: Check if two strings are identical

- 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



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**

What does P mean?

The Problem Set we can solve efficiently.

The Problems We Care

- Recall lots of problems we learns in the lecture.
- They are all searching problems.
  - We have a problem, and an instance x.
  - We have many possible solution y of the instance x.
  - If one of the solution y is correct, then we answer "yes" for x.
  - If all solutions y is not correct, then we answer "no".
- Remark
  - Some of decision problems do not need searching.
  - Optimizing problem has a decision version.

Reasonable Searching Problems

- They are all searching problems.
  - We have a problem, and an instance x.
  - We have many possible solution y of the instance x.
  - If one of the solution y is correct, then we answer "yes" for x.
  - If all solutions y is not correct, then we answer "no".
- We have two challenge for searching problems
  - Search: go through every possible solution.
  - Verify: check the correctness of the solution.
- Reasonable searching problem
  - Search: **hard**
  - Verify: **easy**

Searching Problems

- Some of them are easy (in P)
  - Matching
  - Shortest Path
- Some of them are not so “easy”
  - Hamiltonian Path
  - Vertex Cover

Complexity Class **NP**

- Let us make a complexity class for reasonable searching problems.
- **NP**: verifier definition
  - We have a polynomial time verifier for the problem.

The Complexity Class **NP**

- Nondeterministic Polynomial Time
- Computational Model
  - Non-deterministic Turing Machine
- NP definition: Polynomial Solvable by Non-deterministic Turing Machine
- We only talk about some ideas

Formal definition

- Formal Definition. A decision problem $f: \Sigma^* \to \{0,1\}$ is in **NP** if there exist a polynomial time TM $\mathcal{A}(verifier)$ such that
  - If x is a yes instance $(f(x) = 1)$, there exists poly size $y \in \Sigma^*$ such that $\mathcal{A}$ accepts the input (x, y)
  - If x is a no instance $(f(x) = 0)$, for all poly size $y \in \Sigma^*, \mathcal{A}$ rejects the input (x, y)
- The string y is called a certificate.
- SAT, VertexCover, IndependentSet, SubsetSum, HamiltonianPath are all in **NP**.

Verifier of Vertex Cover

- x is (G, k).
- y is a vertex subset.
- $(x, y) \rightarrow$ Verifier $\mathcal{A}$
  - Accept: y is size k and y cover all edges in G.
  - Reject: Otherwise.
  - It can be done in poly time.

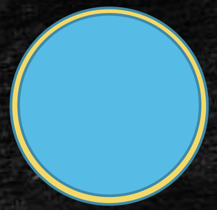
$\mathbf{P} \subseteq \mathbf{NP}$

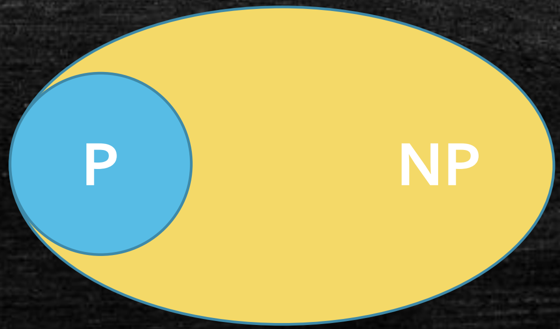
- Proof. If a decision problem $f: \Sigma^* \to \{0,1\}$ is in **P**, we will show it is in **NP**.
- By definition of **P**, there exists a polynomial time TM $\mathcal{A}$ such that $\mathcal{A}$ accepts x if and only if $f(x) = 1$.
- Let $\mathcal{A}'$ be a TM such that it outputs $\mathcal{A}(x)$ 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**?
  - Math Question: If a search problem is easy to verify, is it easy to solve?
- 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?
- Which one is the hardest?

Hard or Easy? The Reduction!

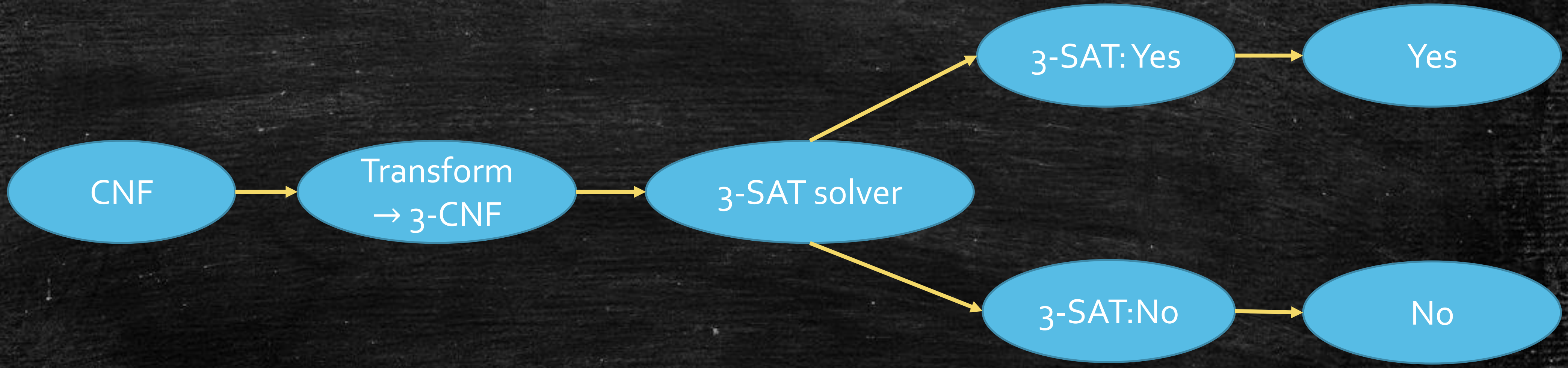
3SAT

- 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 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

- What we have:
  - A 3-SAT solver.
  - A CNF instance.
- Idea: given a CNF formula ϕ, construct a 3-CNF formula ϕ' such that ϕ is a yes SAT instance if and only if ϕ' is a yes 3SAT instance.
- If converting ϕ to ϕ' 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.

The Big Picture



- Remark: This is called Karp Reduction
  - Yes → Yes, No → No.
  - Yes→ No, No → Yes, not allowed!
  - Understanding "weakly harder".
  - 3-SAT is in NP (has "yes" verifier)→ SAT is in NP (has "yes" verifier).

Understand “hard”

- Understanding “weakly harder”.
  - 3-SAT is in NP (has “yes” verifier)→ SAT is in NP (has “yes” verifier).
  - SAT is in NP (has “yes” verifier)→ 3-SAT is in NP (has “yes” verifier).
  - 3-SAT is in P (has poly solver) → SAT is in P (has poly solver).
  - SAT is in P (has poly solver) → 3-SAT is in P (has poly solver).
- Understanding “equally hard”
  - 3-SAT is in P (has poly solver) = SAT is in P (has poly solver).
  - 3-SAT is in NP (has “yes” verifier)= SAT is in NP (has “yes” verifier).

Proving 3SAT is “weakly harder than” SAT

- We can “break” a long clause in ϕ 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” an 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 ℓ_i is true, we can make all RHS clauses true by properly setting y_i's

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

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

Proving 3SAT is “weakly harder than” SAT

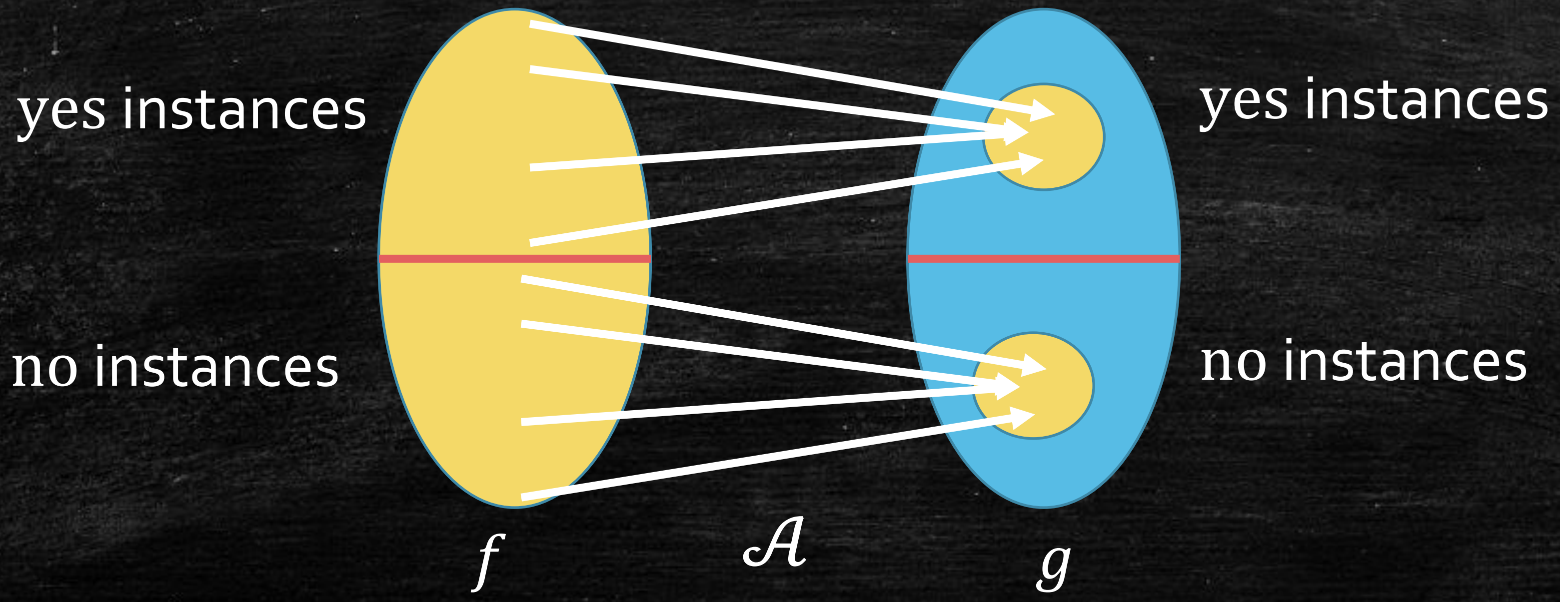
- We have described how to convert a CNF formula ϕ to a 3-CNF formula ϕ'.
- The conversion can clearly be done in polynomial time.
- We have shown that ϕ is a yes SAT instance if and only if ϕ' is a yes 3SAT instance.
- If we have a polynomial time algorithm for 3SAT, we have a polynomial time algorithm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

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 yes instance of g if a yes instance of f is input
  - $\mathcal{A}$ outputs a no instance of g if a 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:
  - $\text{SAT} \leq_k \text{3SAT}$

Reduction

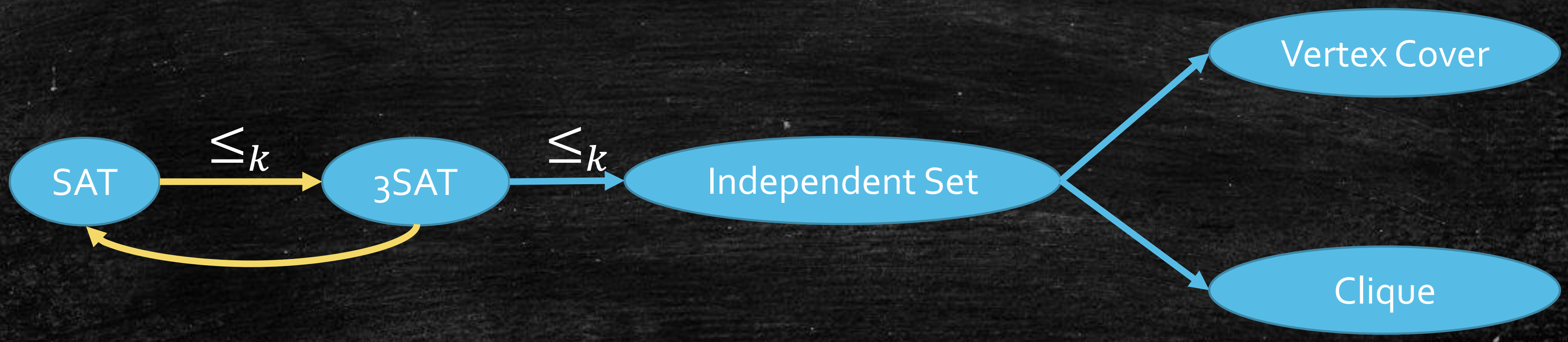
- In the reduction, $f \leq_k g$, the TM $\mathcal{A}$ defines a mapping.
- The mapping needs not to be one-to-one.
- The mapping needs not to be onto.



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



IndependentSet is “weakly harder” than 3SAT

- Same Idea before:
  - Given a 3SAT instance ϕ,
  - Construct a IndependentSet instance $(G = (V, E), k)$,
  - Such that ϕ is a yes instance if and only if $(G = (V, E), k)$ is a yes instance.
  - Polynomial time construction.

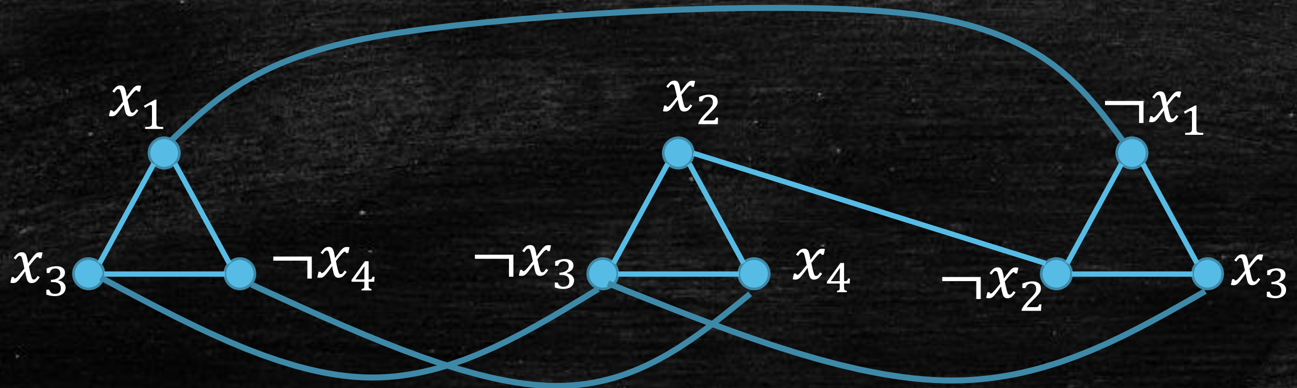
IndependentSet is “weakly harder” than 3SAT

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

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

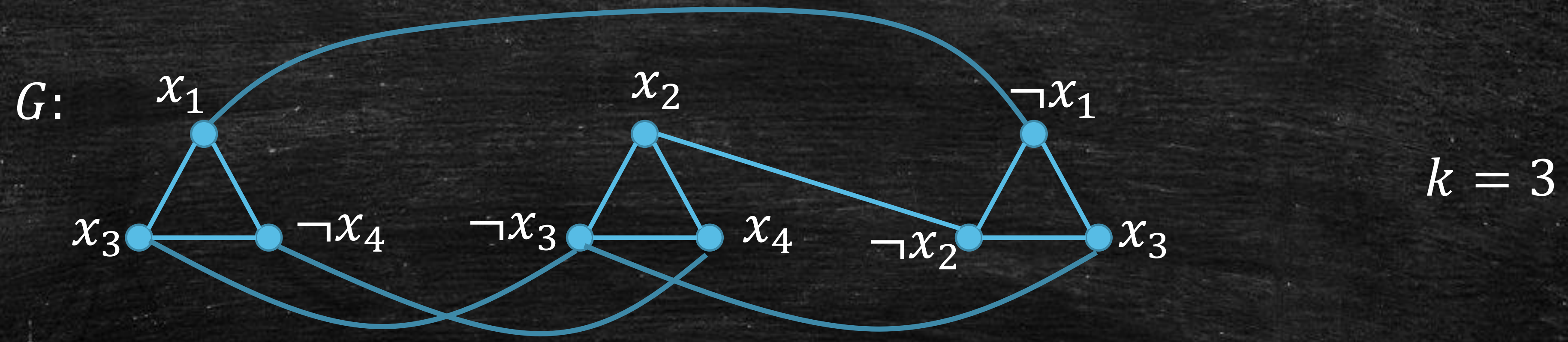
G:



$k = 3$

3SAT Yes → Independent Set Yes

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

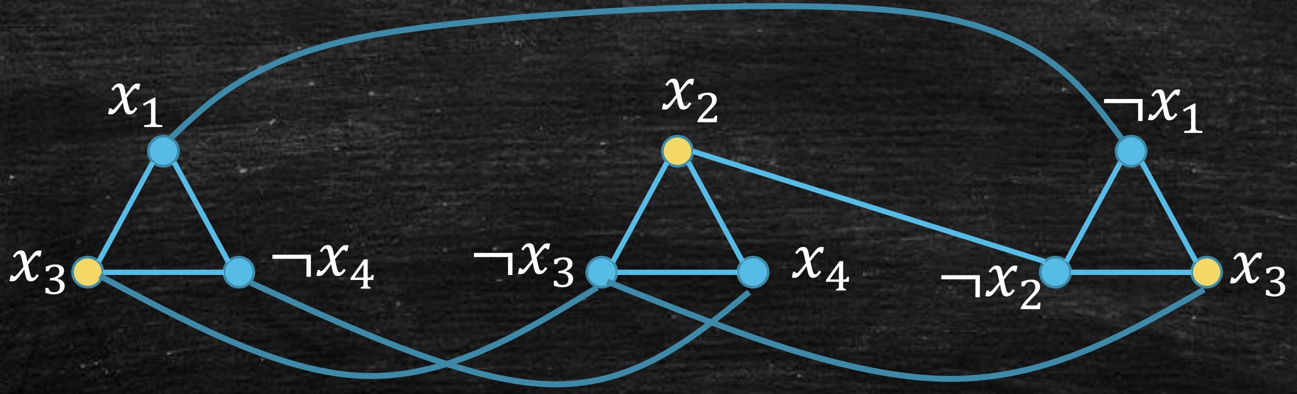


- If ϕ is a yes instance, each clause must have a literal with value true.
- For each triangle in G, pick exactly one vertex representing a true literal in S.
- S is an independent set and $|S| = k$. So (G, k) is a yes instance.

3SAT Yes → Independent Set Yes

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

G:



$k = 3$

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

3SAT No → Independent Set No

- If (G, k) is a yes instance → ϕ is yes.
- 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 ϕ true.

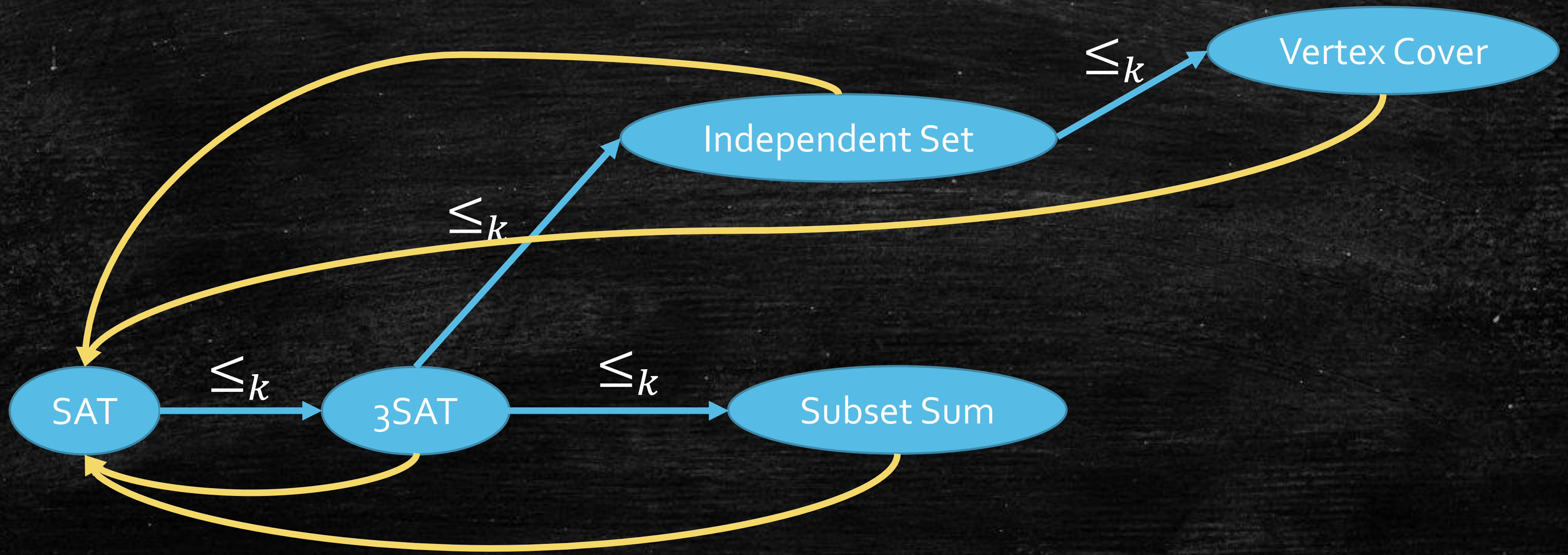
Now we know what is “hard” means.

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 $g \in$ **NP**.
- **Definition.** A decision problem f is NP-complete if $f \in$ **NP** and $g \leq_k f$ for any problem $g \in$ **NP**.
- **[Cook-Levin Theorem] SAT is NP-complete.**

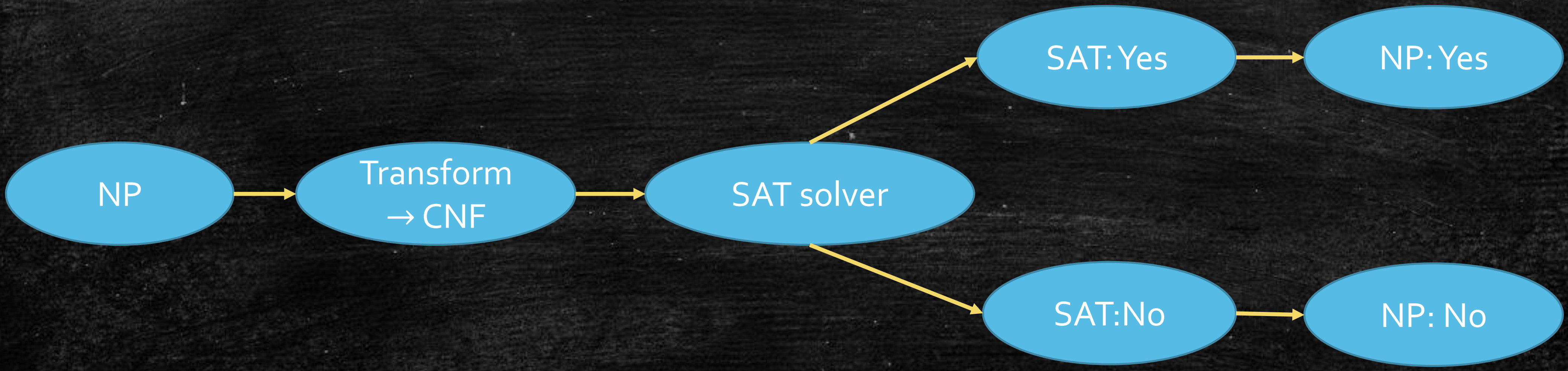
More NP-Complete Problems

- 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**.



Proof Sketch of Cook-Levin Theorem

- We have seen SAT is in **NP**.
- Consider an arbitrary **NP** problem f. We will show $f \leq_k \mathrm{SAT}$.



Proof Sketch of Cook-Levin Theorem

- 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	s	p
Step 1	1	1	0	0	0	1	1	1	1	0	1	0
Step 2	0	1	0	0	0	1	1	1	1	0	2	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	3	2
Final Step	0	1	1	0	0	0	1	1	0	0	9(a/r)	100

What does it mean?

- If x is a yes instance.
- Put x in step 0, we can find a y in step 0, so that the final step go into an accept status.
- Put x in step 0, we can not find a y in step 0, so that the final step go into an accept status.

	x					y						
Step 0	x_0	x_1	x_2	…	x_n	y_0	y_1	y_2	…	y_m	s	p
Step 1	1	1	0	0	0	1	1	1	1	0	1	0
Step 2	0	1	0	0	0	1	1	1	1	0	2	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	3	2
Final Step	0	1	1	0	0	0	1	1	0	0	9(a/r)	100

Proof Sketch of Cook-Levin Theorem

Transform → CNF

	x_0	x_1	x_2	…	x_n	y_0	y_1	y_2	…	y_m	s	p
Step 0	x_0	x_1	x_2	…	x_n	y_0	y_1	y_2	…	y_m	s	p
Step 1	1	1	0	0	0	1	1	1	1	0	1	0
Step 2	0	1	0	0	0	1	1	1	1	0	2	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	3	2
Final Step	0	1	1	0	0	0	1	1	0	0	9(a/r)	100

- View each cell as a variable in CNF.
  - Restrict $x_0 \sim x_n$ is the input x in step 0. (We do not need to restrict y)
  - Restrict s to be a accept status at the final step.
  - Restrict step i to step $i+1$ to be a feasible transform of $\mathcal{A}$.
- Make a CNF formula to complete these tasks!

How to make the CNF? Easy Examples

- Restrict $x_1^0 \sim x_0^0$ to be exactly x.
  - If $x_1 = 0$, we add … $\wedge \neg x_1^0 \wedge \cdots$
  - If $x_1 = 1$, we add … $\wedge\, x_1^0 \wedge \cdots$
- Restrict the final status is accepted.
  - $s^{last} = accept$
- Restrict step i to step $i+1$ to be a feasible transform of $\mathcal{A}$.
  - For example, maybe some x_j^i should not change.
  - We add … $\wedge\, (x_j^i \vee z) \wedge (\neg x_j^{i+1} \vee \neg z) \wedge \cdots$.

What do we have?

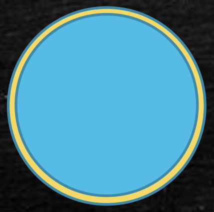
- We have a CNF formula, such that
  - If SAT answer “yes”, we have a “y” such that $\mathcal{A}$ accepts (x, y).
  - If SAT answer “yes”, we do not have “y” such that $\mathcal{A}$ accepts (x, y).
- So
  - If SAT answer “yes”, x is “yes”.
  - If SAT answer “no”, x is “no”.
- High-level Intuition: a CNF formula is sufficient to simulate the execution of a Turing Machine!

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

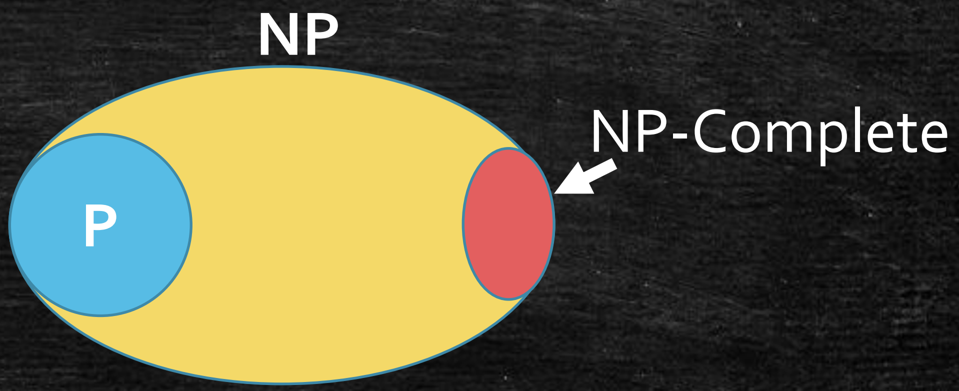
- **Theorem.** If f is NP-complete and $f \in$ **P**, then **P** = **NP**.
- Proof. Suppose there is a polynomial time TM $\mathcal{A}$ that decides f. We will show $g \in$ **P** for any $g \in$ **NP**.
- Since f is NP-hard, $g \leq_k f$, and let $\mathcal{A}'$ be the polynomial time TM that does the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



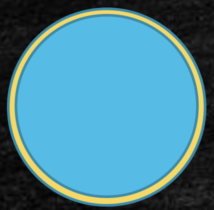
P = NP



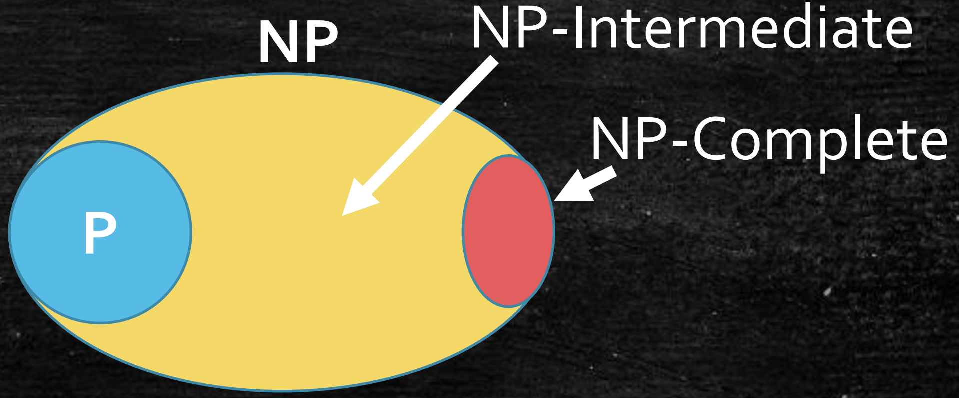
$P \subsetneq NP$

NP-Intermediate

- **[Ladner's Theorem]** If P ≠ NP, then there exist decision problems that are neither in P nor NP-complete.
- Such problems are called NP-intermediate.
- Candidates:
  - graph isomorphism problem
  - factoring



P = NP



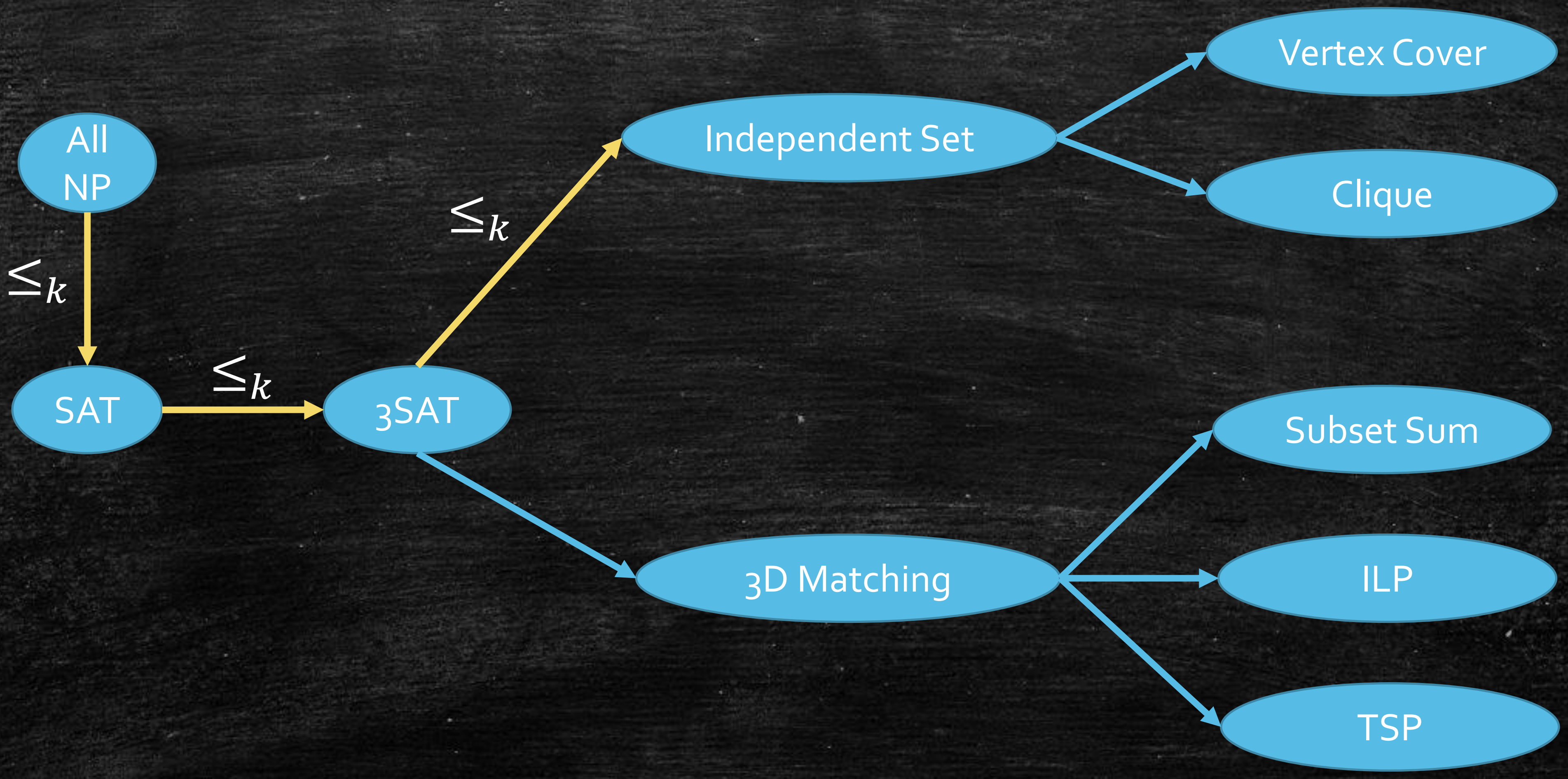
P ⊊ NP

NP-Hard vs NP-Complete

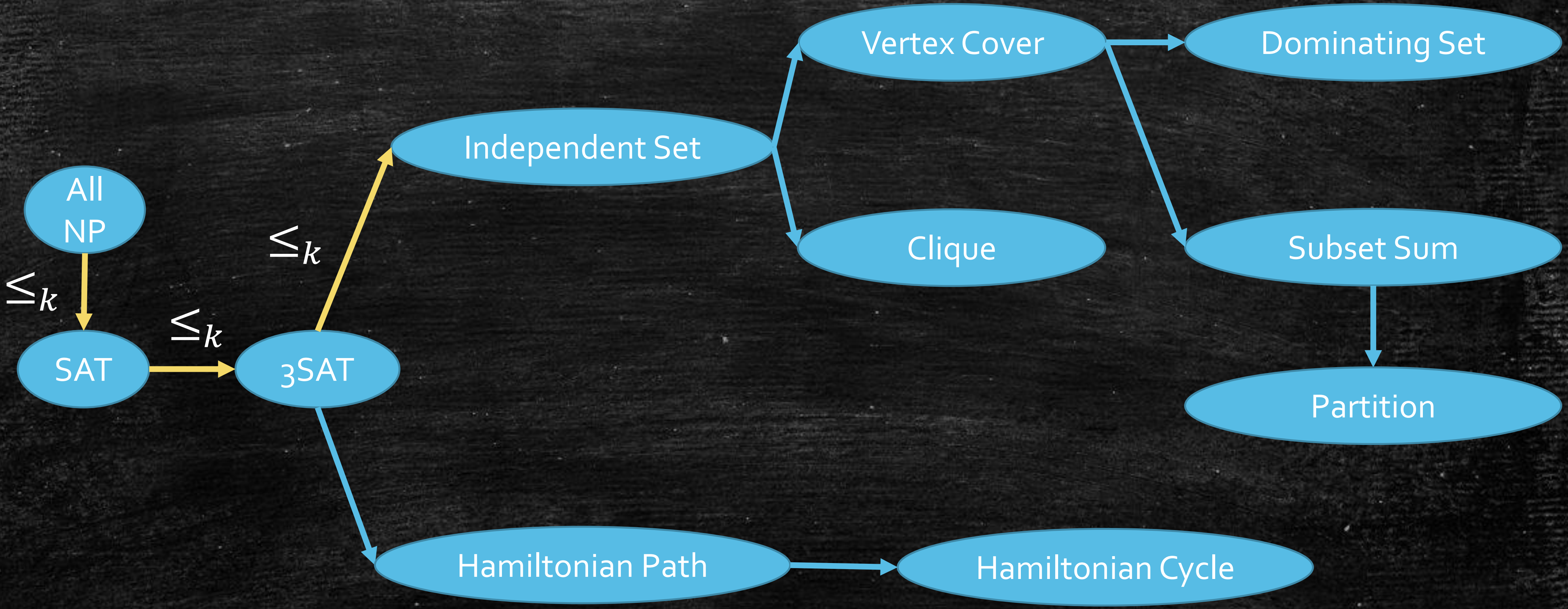
Difference between NP-hardness and NP-completeness:

- 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.

The Reduction Graph (on Book)

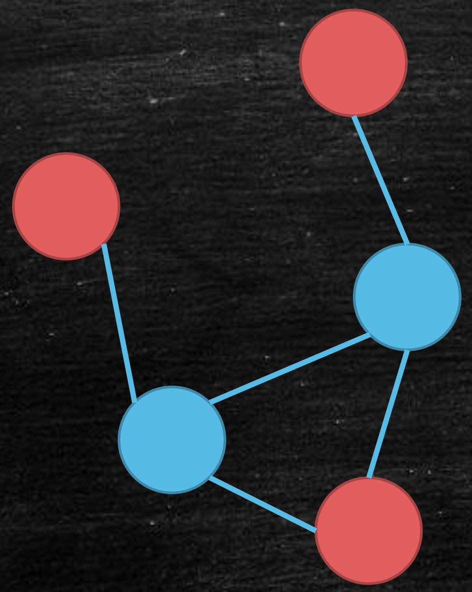


Our Reduction Graph

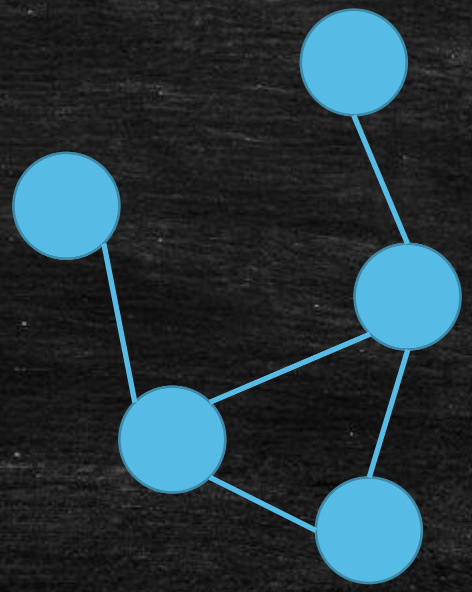


Independent Set $\leq_k$ Vertex Cover

Independent Set



Vertex Cover?



Independent Set $\leq_k$ Vertex Cover

- Key Observation
- S is an Independent Set of G, if and only if $V - S$ is a Vertex Cover.
- So, if we have a Vertex Cover Solver, how to solve Independent Set?
- Vertex Cover Solver, input (G, k)
  - Answer "yes" if there is a vertex cover with size k.
  - Answer "no" if there does not exist a vertex cover with size k.

Independent Set $\leq_k$ Vertex Cover

- When we receive an Independent Set Input (G, k).
- How to use the Vertex Cover solver?

Independent Set $\leq_k$ Vertex Cover

- When we receive an Independent Set Input (G, k).
- How to use the Vertex Cover solver?
- Just input $(G, |V| - k)$ to the Vertex Cover solver!
  - If G has a $|V| - k$ size vertex cover, we have $|S| = k$, where S is an independent set.
  - If G has a size k independent set, we should have a $|V| - k$ size vertex cover.

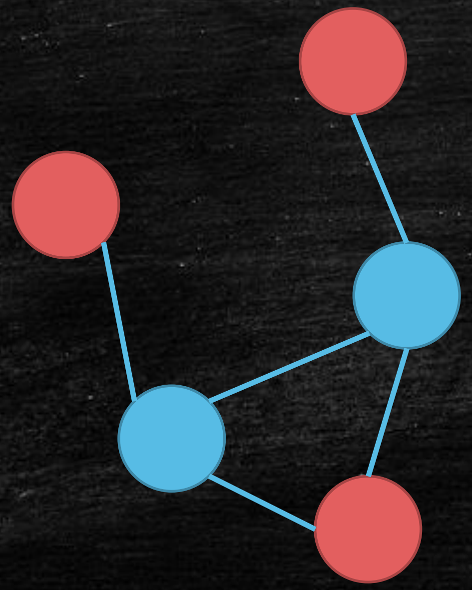
NP-completeness

- Does it mean Vertex Cover is NP-Complete?
- One more step:
  - Show Vertex Cover is in NP.

Independent Set $\leq_k$ Clique

- k –Clique Problem
  - Is there a size k clique inside G?

Independent Set



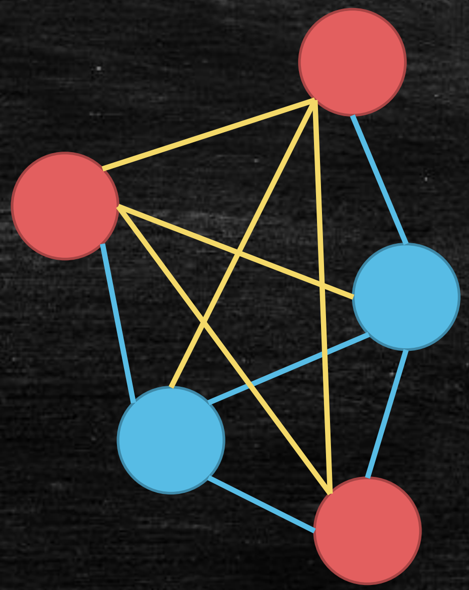
Clique

Key Observation

- S is an Indecent Set in G if and only if S is a clique in $\bar{G}$.

Independent Set (G)

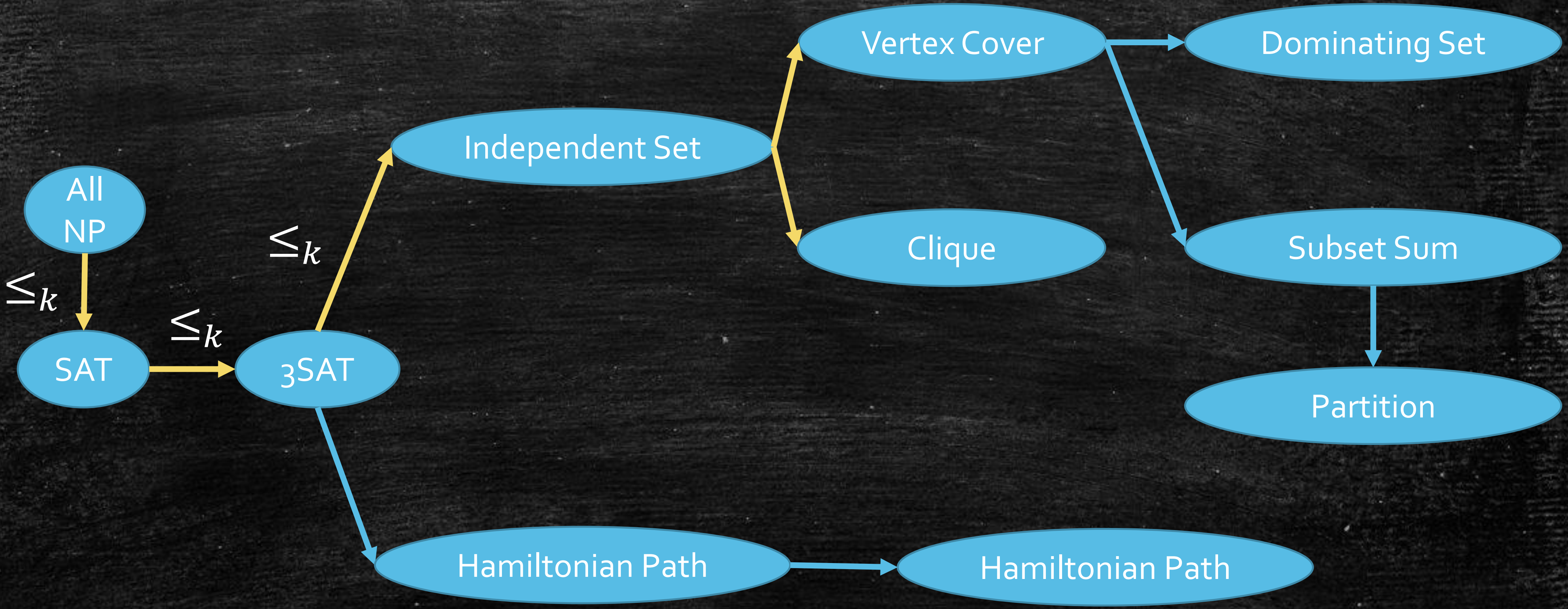
Clique $(\bar{G})$



Independent Set $\leq_k$ Clique

- When we receive an Independent Set Input (G, k).
- How to use the Clique solver?
- Just input $(\bar{G}, k)$ to the Clique solver!
  - If G has a k size independent set, we have $|S| = k$, where S is a clique of $\bar{G}$.
  - If $\bar{G}$ has a size k clique, we should have a k size independent set in G.

Our Reduction Graph



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 an effective tool to show one problem is "weakly harder" than another.