# Compiler Design

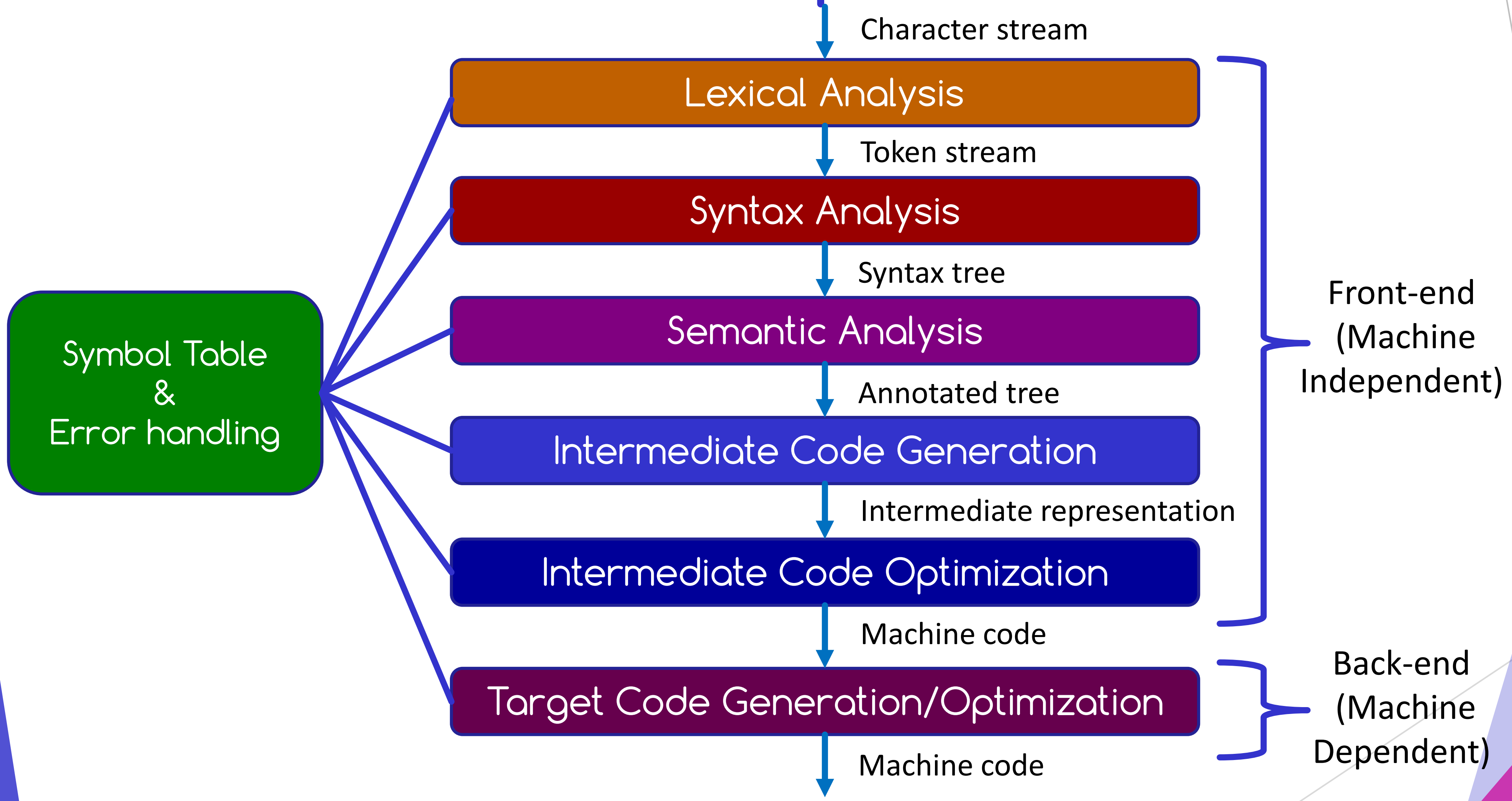
## Lecture 3: Lexical Analysis II

Sahar Selim

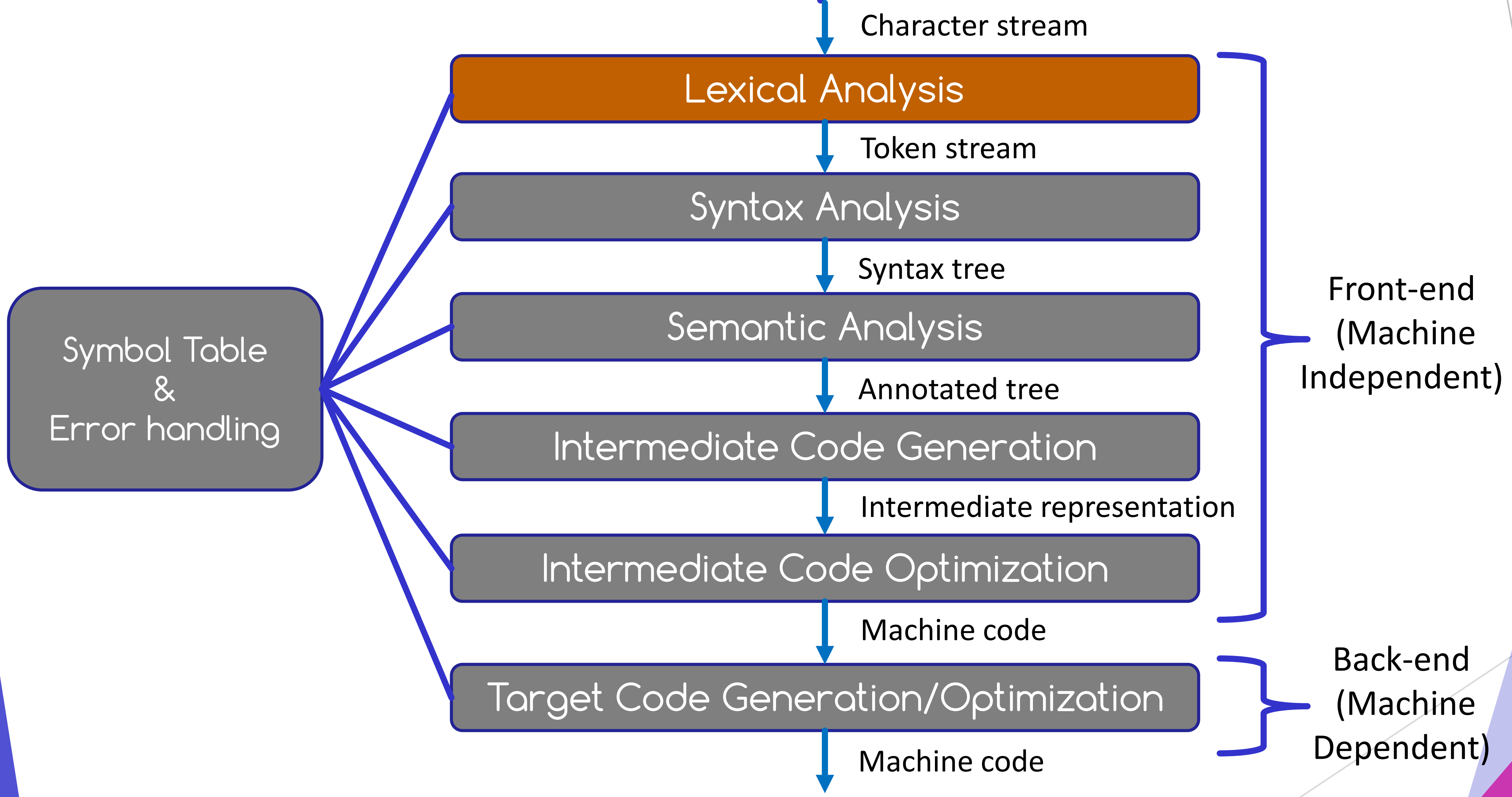
# Agenda

1. Finite Automata
2. Deterministic Finite Automaton (DFA)
3. Non-Deterministic Finite Automaton (NFA)

# Phases of a Complier



# Phases of a Complier



# Lexical Specification

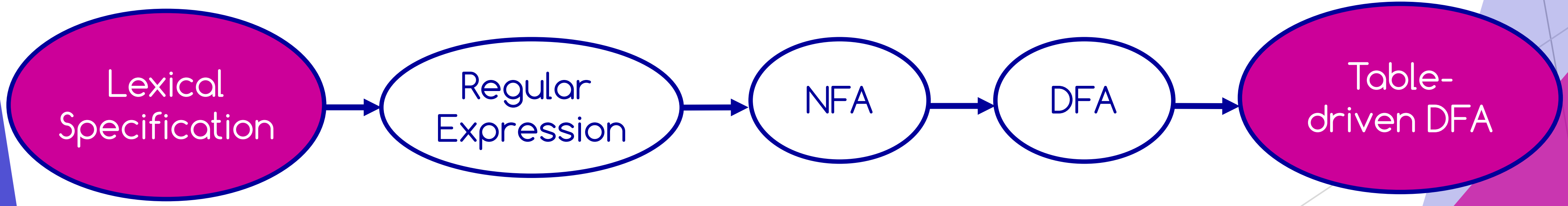
- Using regular expressions to specify tokens
  - keyword = begin | end | if | then | else
  - identifier = letter (letter | digit | underscore)*
  - integer = digit+
  - relop = < | <= | = | <> | > | >=
  - letter = a | b | … | z | A | B | … | Z
  - digit = 0 | 1 | 2 | … | 9

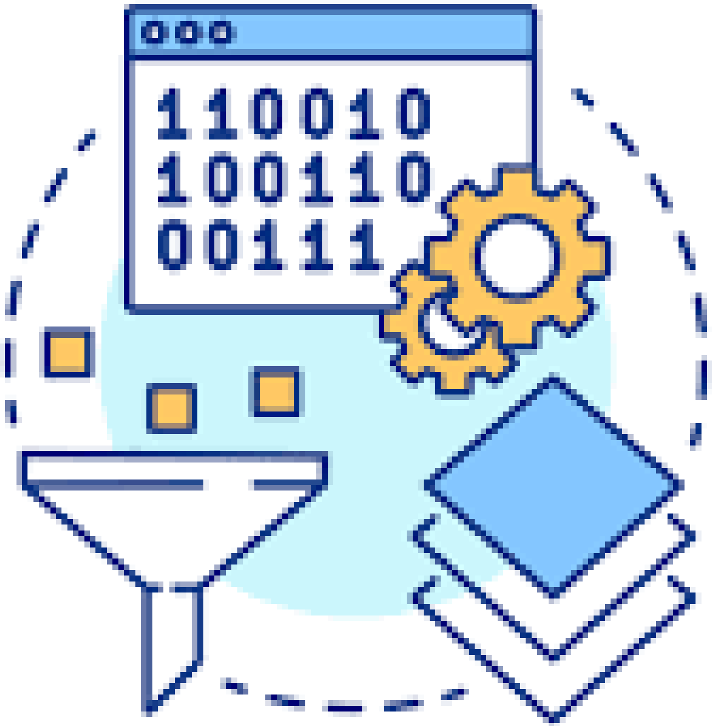
# More Examples

| Regular Expression | Meaning |
| --- | --- |
| $[abc]^+$ | |
| $[abc]^*$ | |
| $[0-9]^+$ | |
| $[1-9][0-9]^*$ | |
| $[a-zA-Z][a-zA-Z0-9\_]^*$ | |

# Implementing Regular Expressions

- Regular expressions can be implemented using **finite automata.**
- There are two kinds of finite automata:
  - **NFAs** (nondeterministic finite automata)
  - **DFAs** (deterministic finite automata)
- The steps of implementing the lexical analyzer

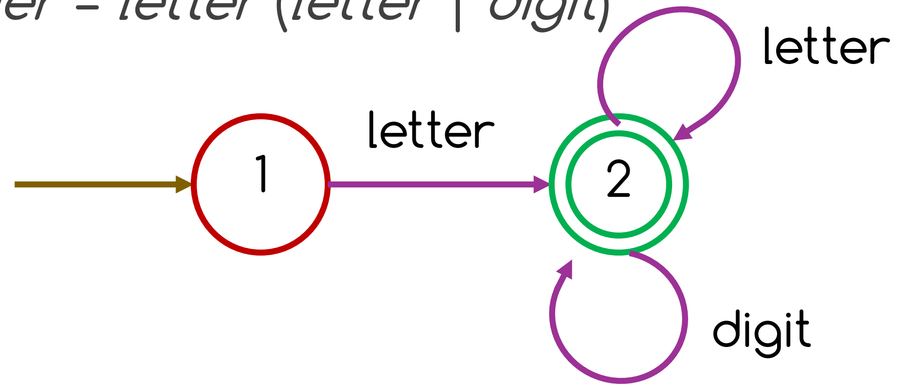




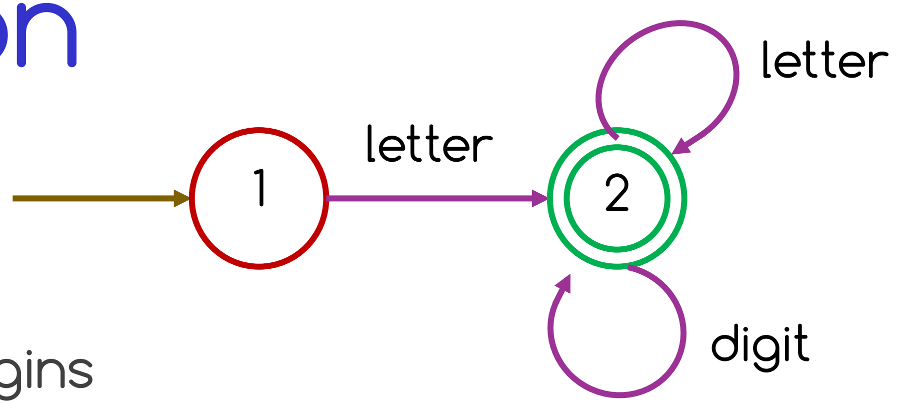

# Finite Automata

# Introduction to Finite Automata

- Finite automata (finite-state machines) are a mathematical way of describing particular kinds of algorithms.
- A strong relationship between finite automata and regular expression
  - *Identifier = letter (letter | digit)**

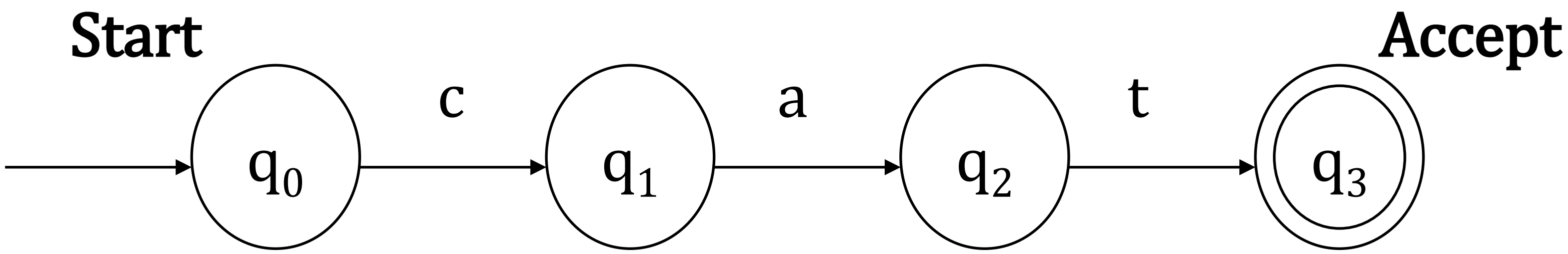


# Finite Automaton



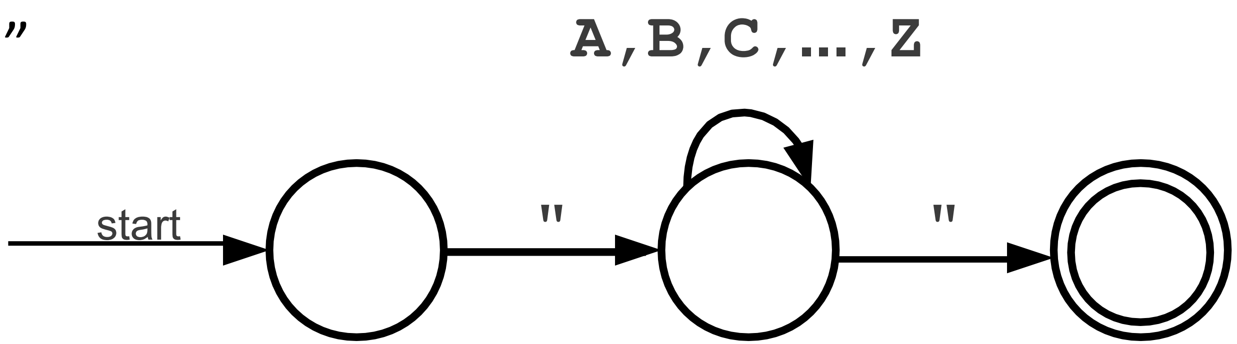

1. START STATE
   - The recognition process begins
   - Drawing an **unlabeled arrowed** line to it coming “from nowhere”
2. TRANSITION
   - Record a change from one state to another upon a match of the character or characters by which they are labeled.
3. ACCEPTING STATES
   - Represent the end of the recognition process.
   - Drawing a double-line border around the state in the diagram

# Example: FSA for “cat”

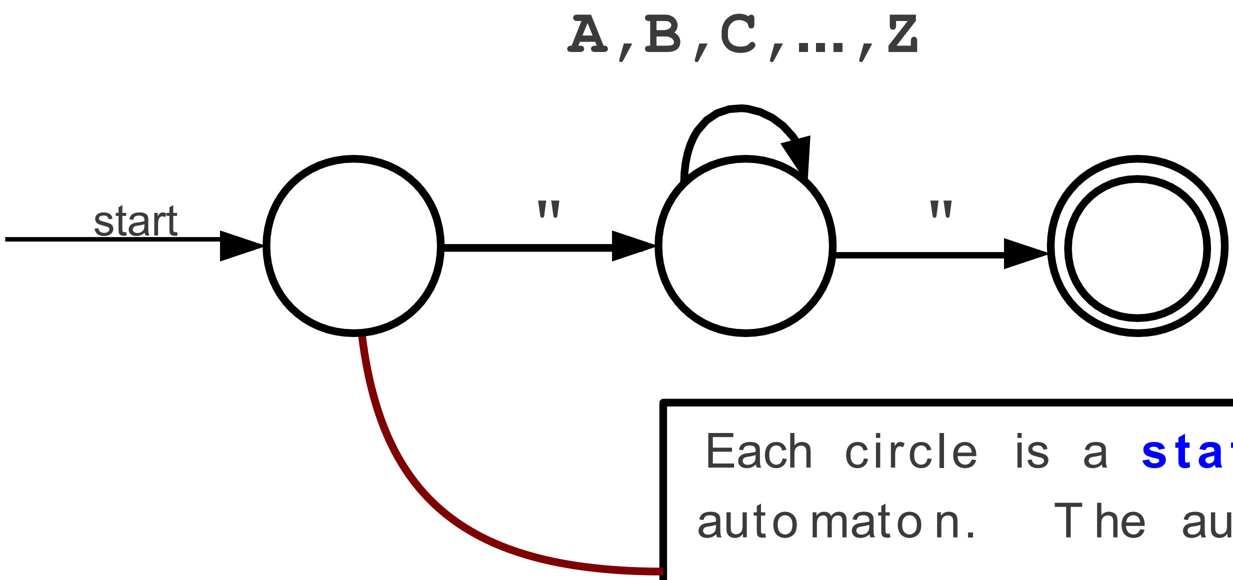


# A Simple Automaton

String = “ [A-Z]* ”

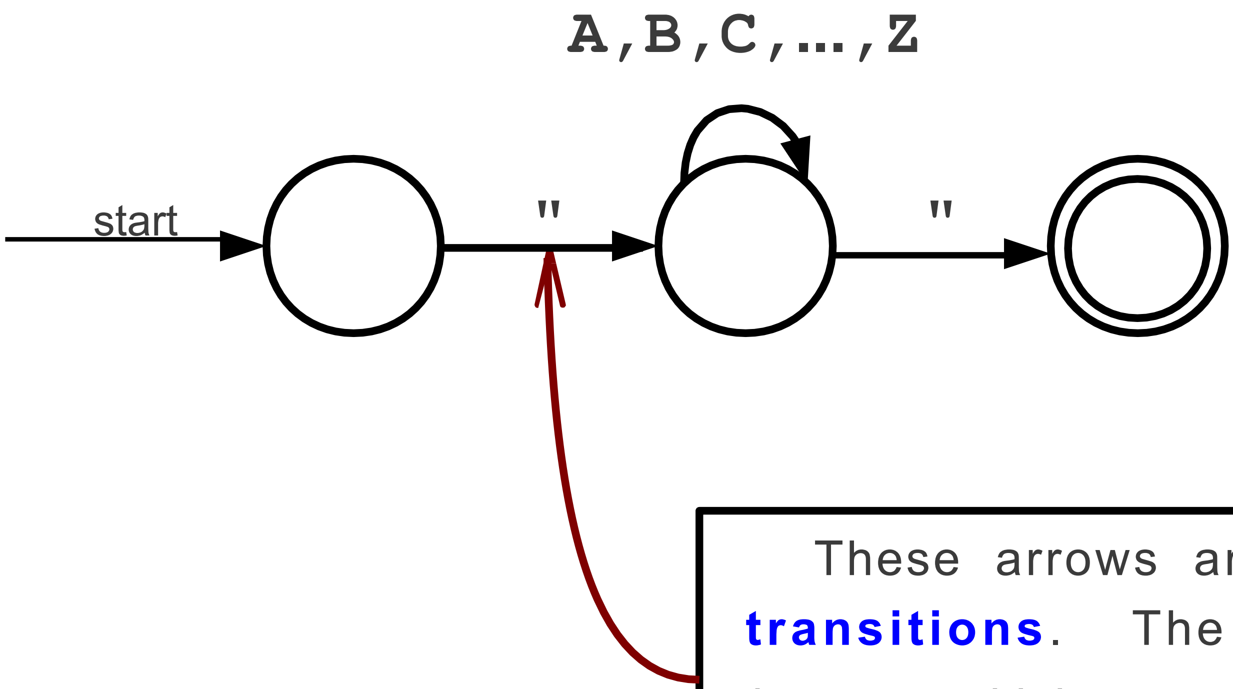


# A Simple Automaton



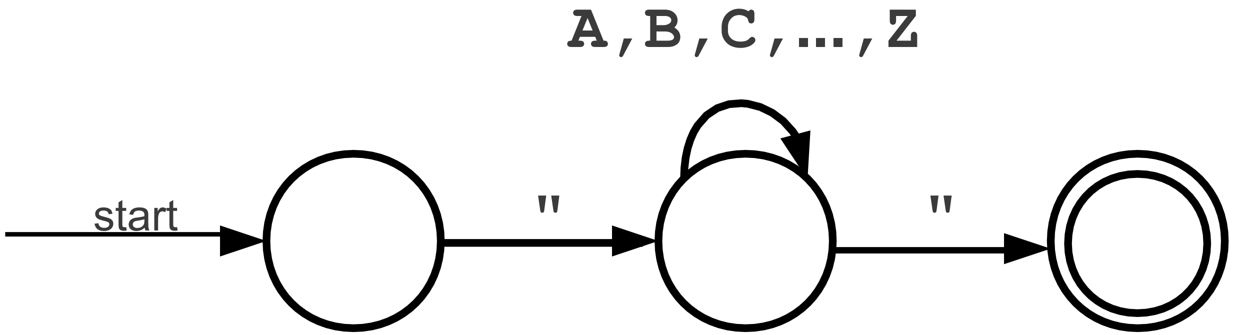

Each circle is a **state** of the automaton. The automaton's configuration is determined by what state(s) it is in.

# A Simple Automaton



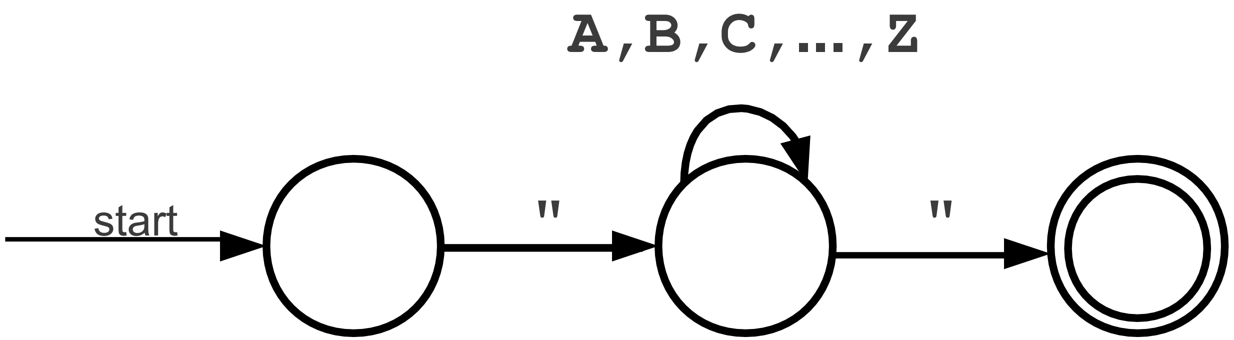

These arrows are called **transitions**. The automaton changes which state(s) it is in by following transitions.

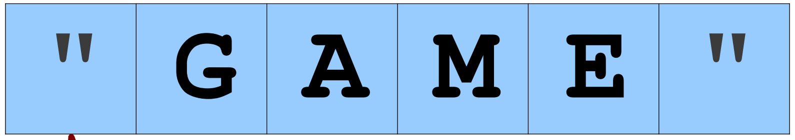
# A Simple Automaton




| " | G | A | M | E | " |
|---|---|---|---|---|---|

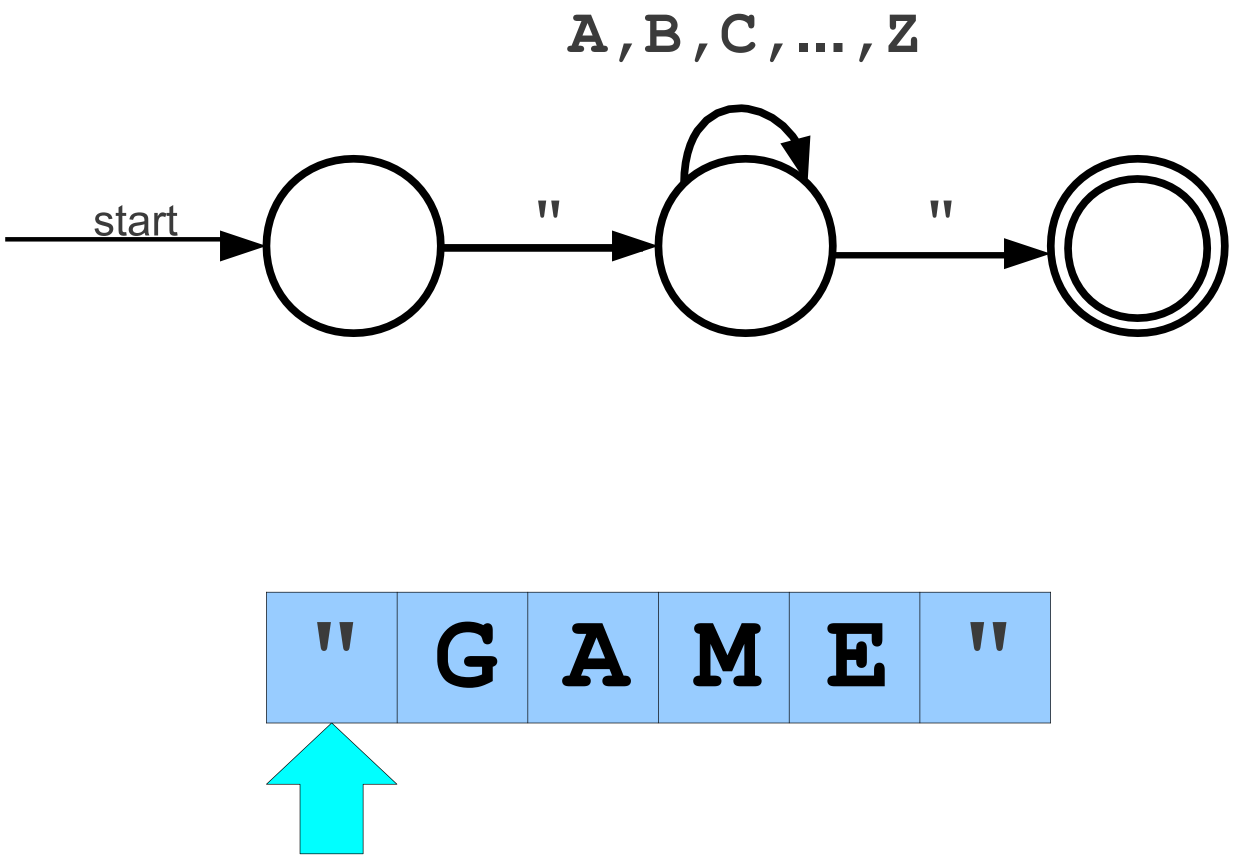
# A Simple Automaton



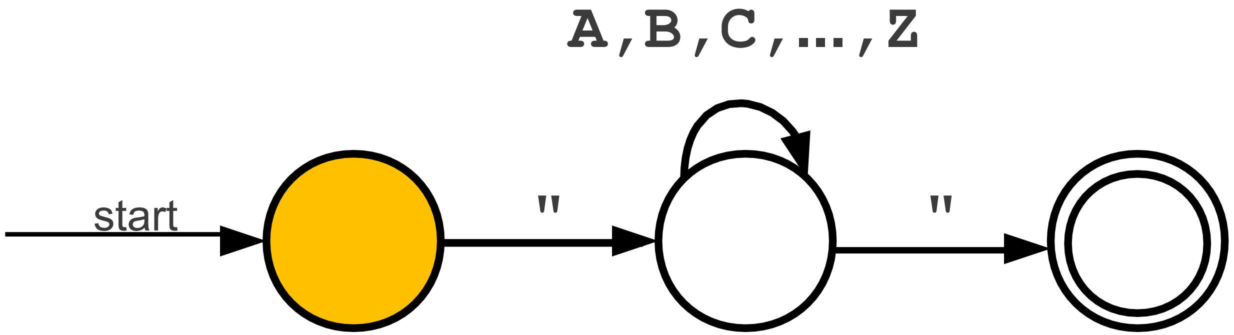



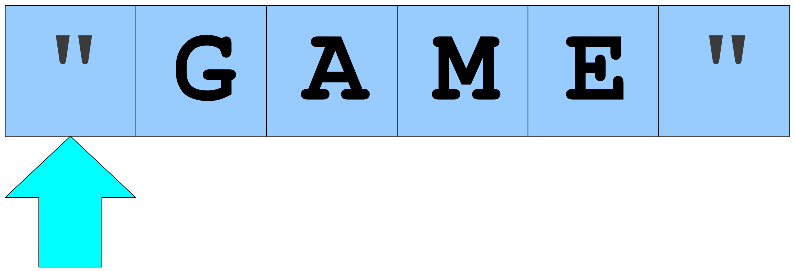

The automaton takes a string as input and decides whether to accept or reject the string.

# A Simple Automaton

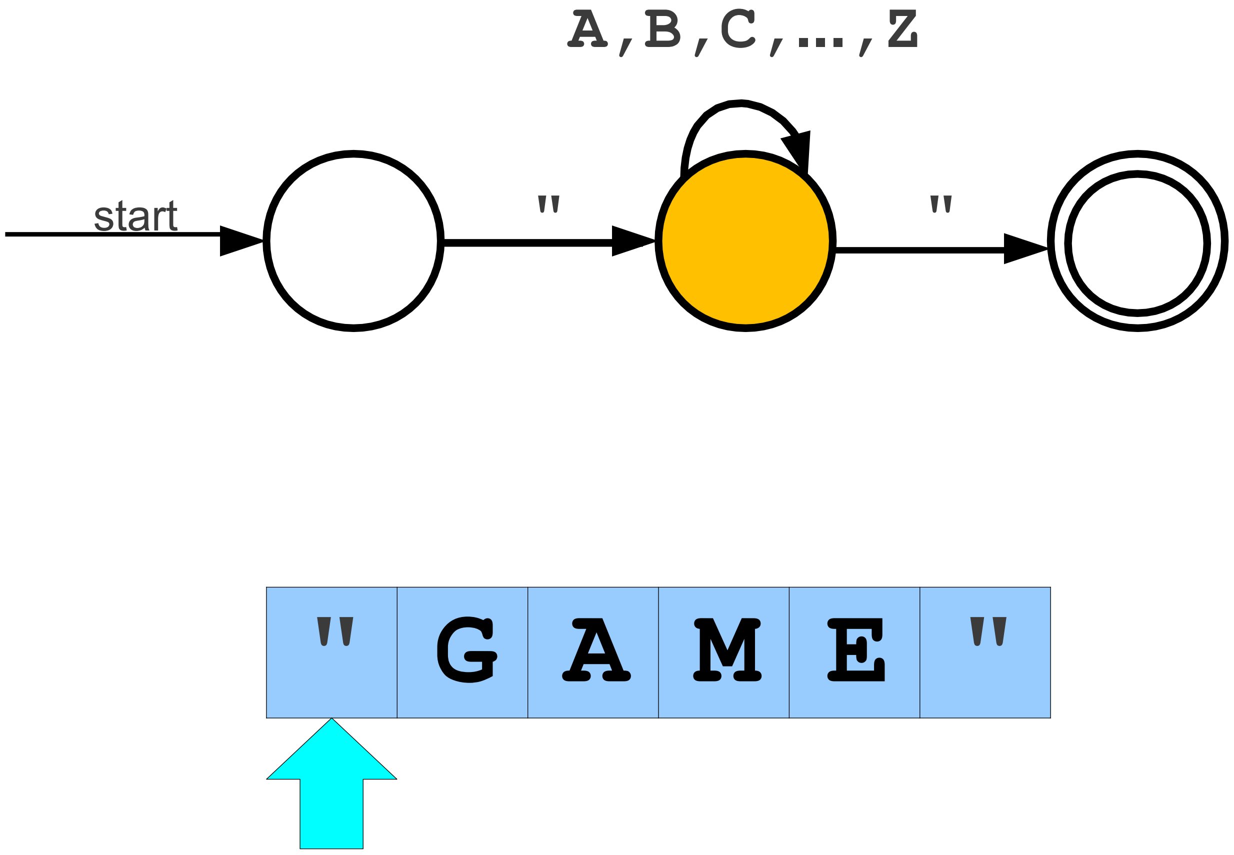


# A Simple Automaton

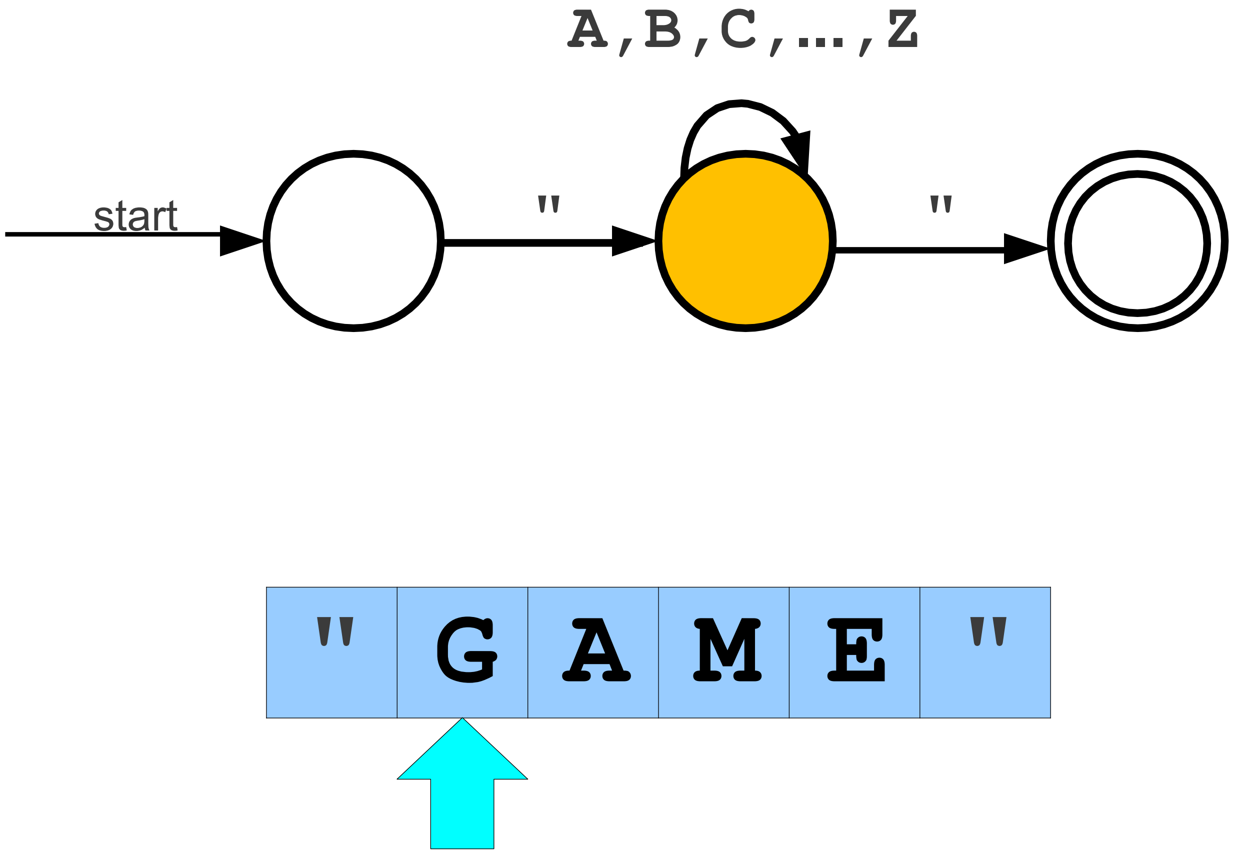


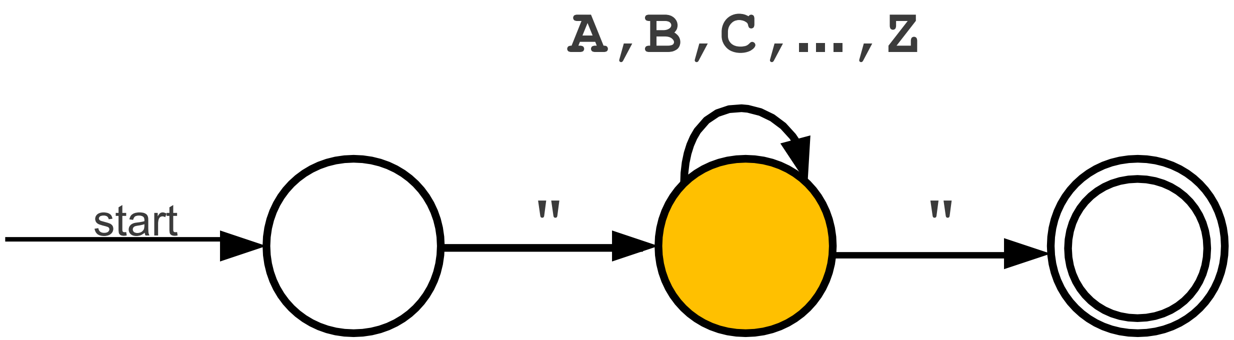
# A Simple Automaton

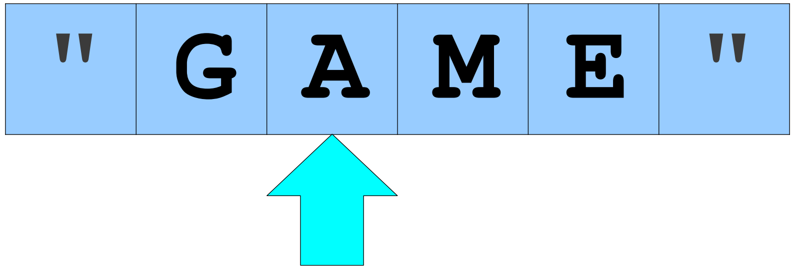


# A Simple Automaton

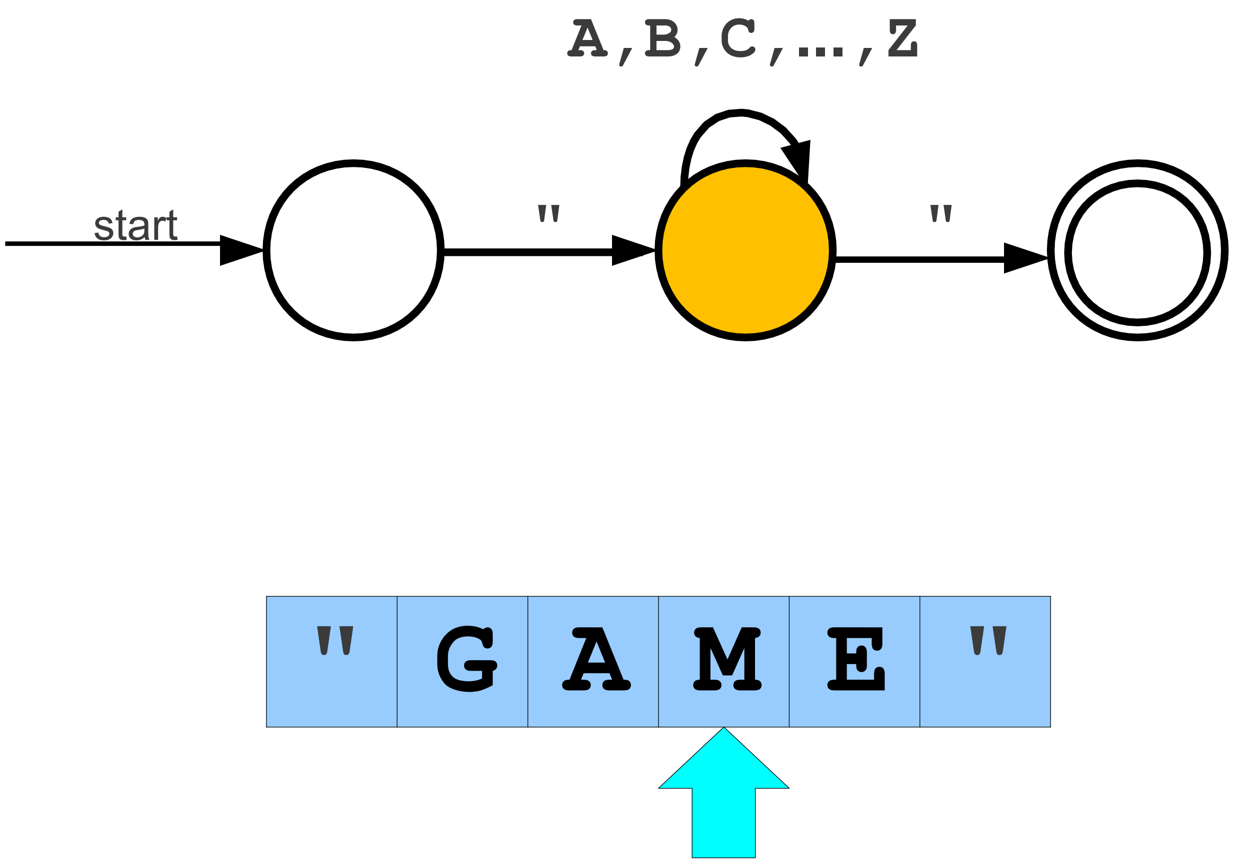


# A Simple Automaton

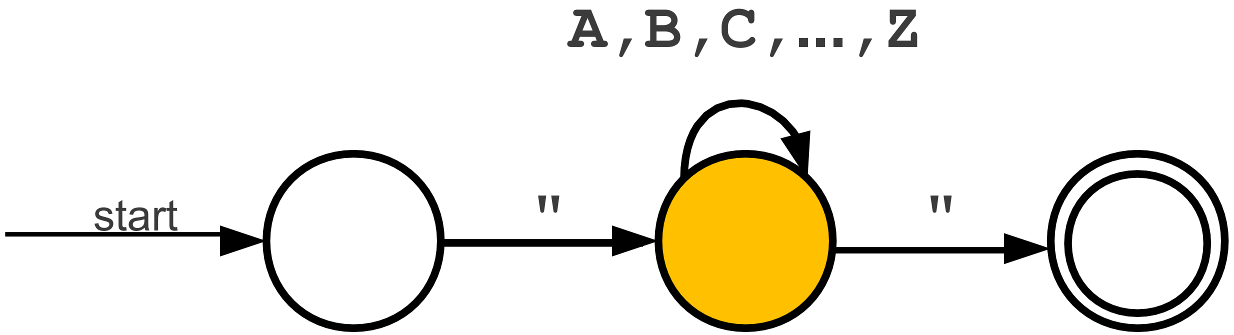


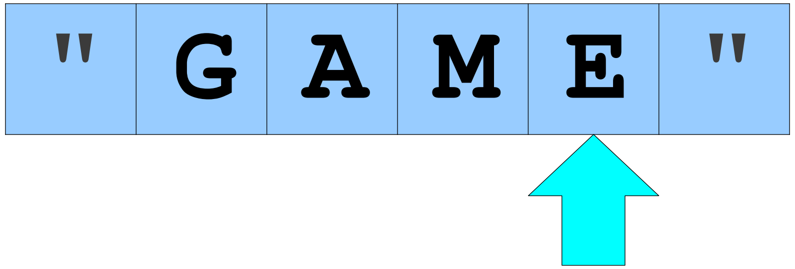



# A Simple Automaton

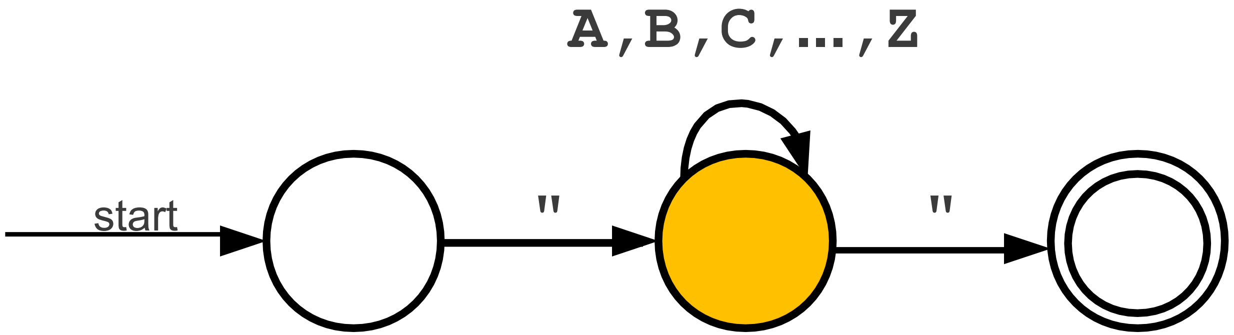


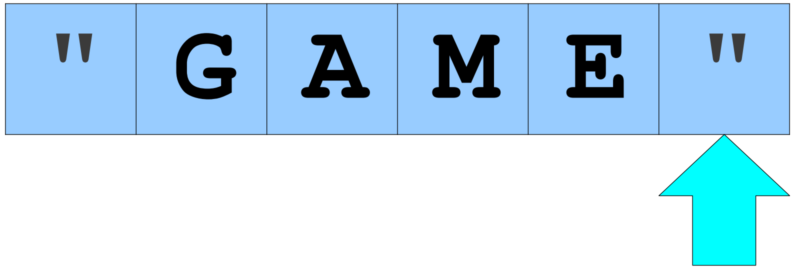
# A Simple Automaton



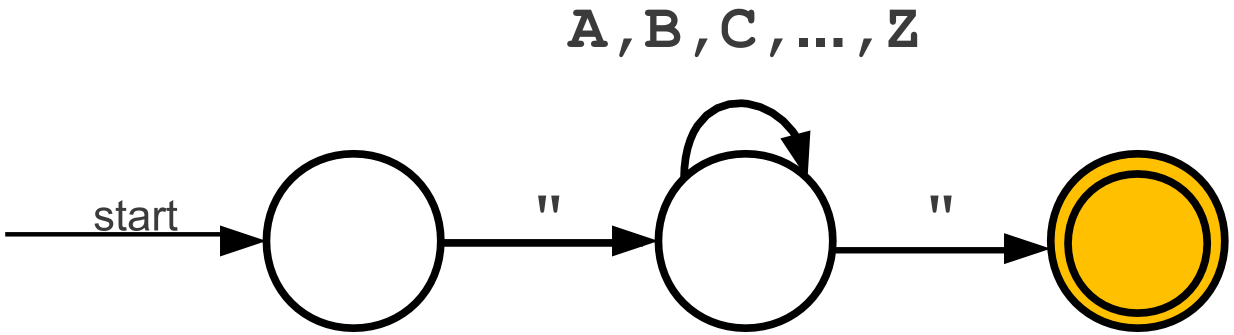



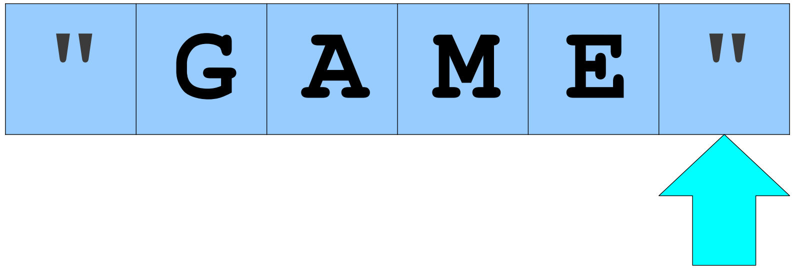
# A Simple Automaton



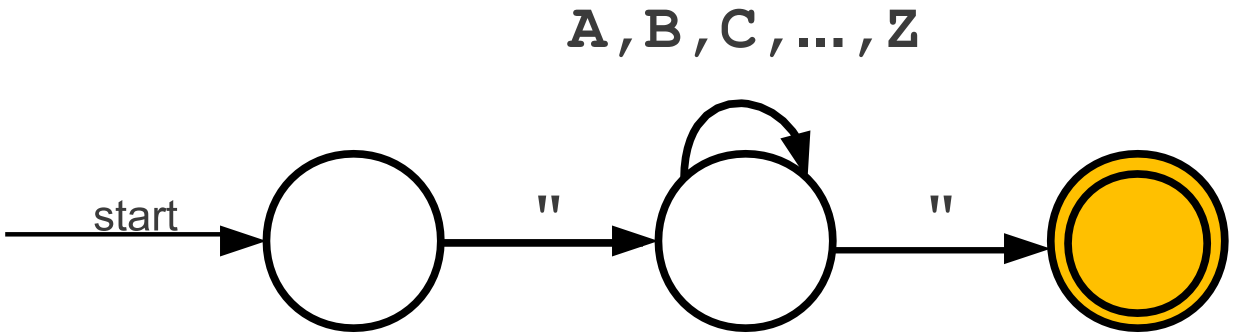



# A Simple Automaton



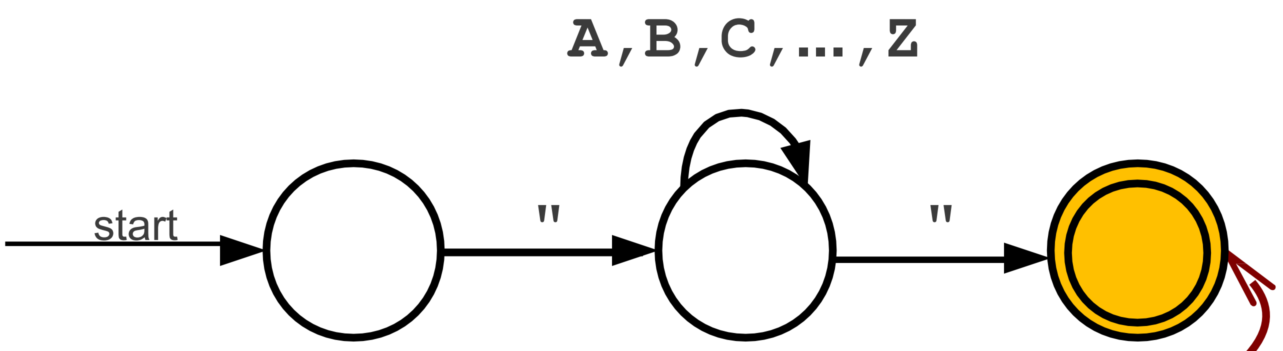



# A Simple Automaton




| | | | | | |
|---|---|---|---|---|---|
| " | G | A | M | E | " |

# A Simple Automaton

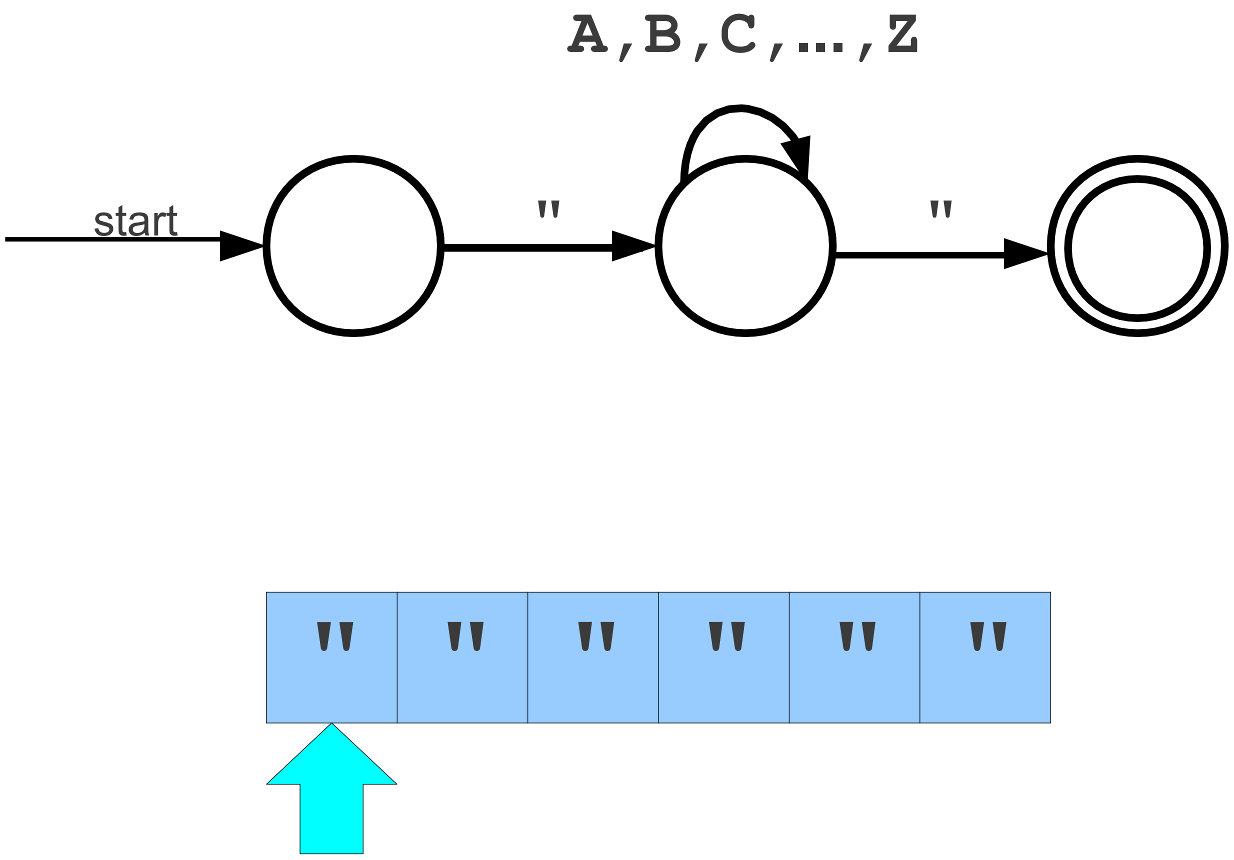



| " | G | A | M | E | " |
|---|---|---|---|---|---|

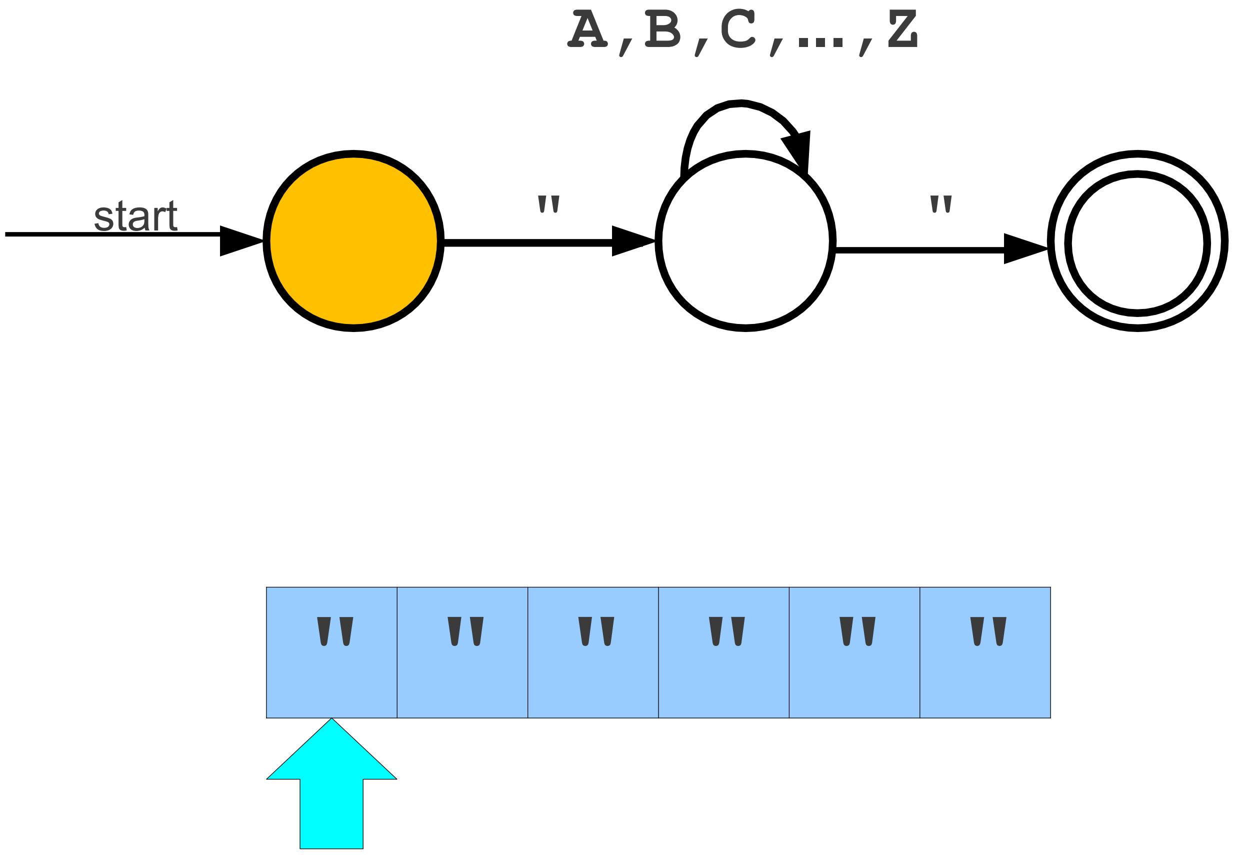
The double circle indicates that this state is an **accepting state**.

The automaton accepts the string if it ends in an accepting state.

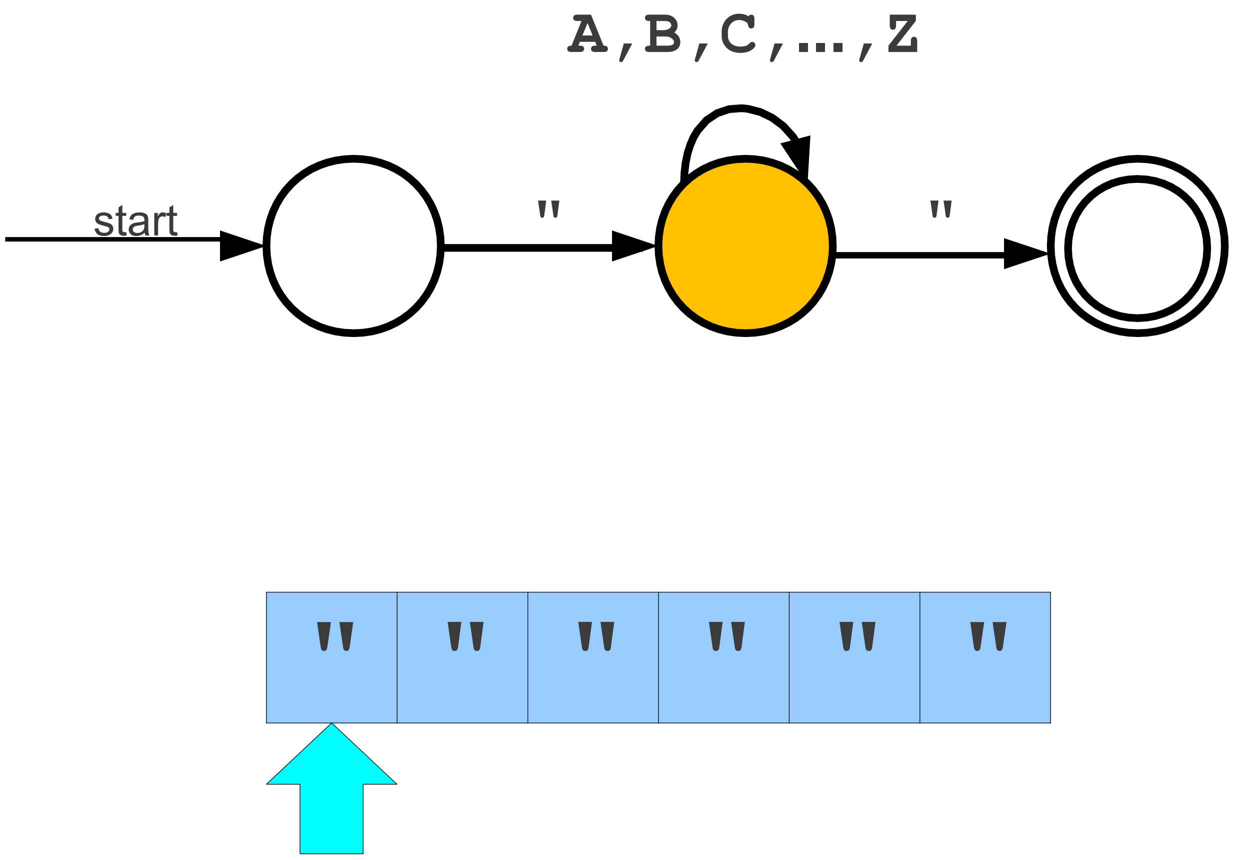
# A Simple Automaton



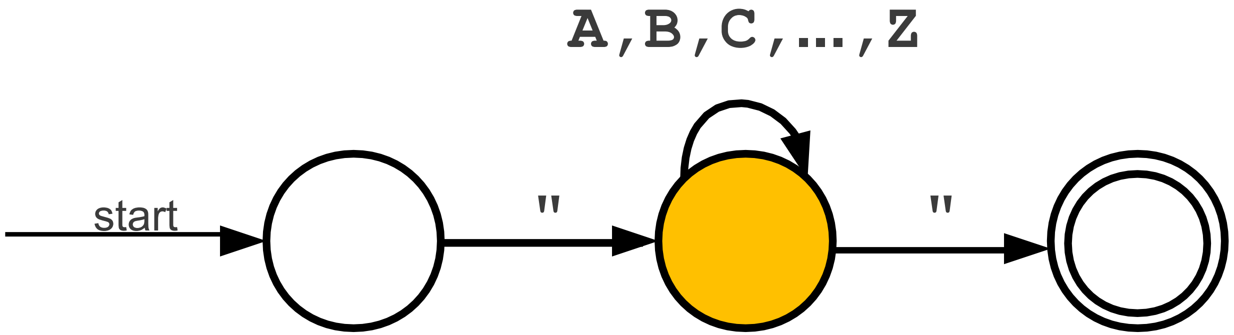
# A Simple Automaton

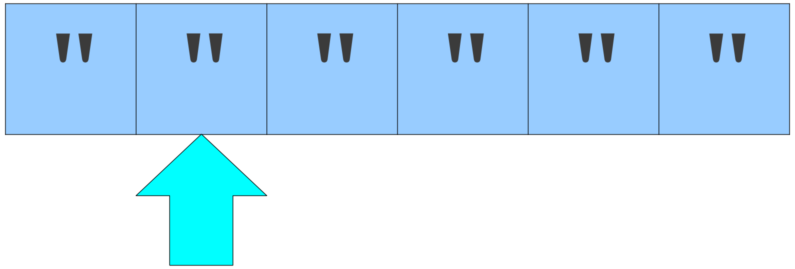


# A Simple Automaton

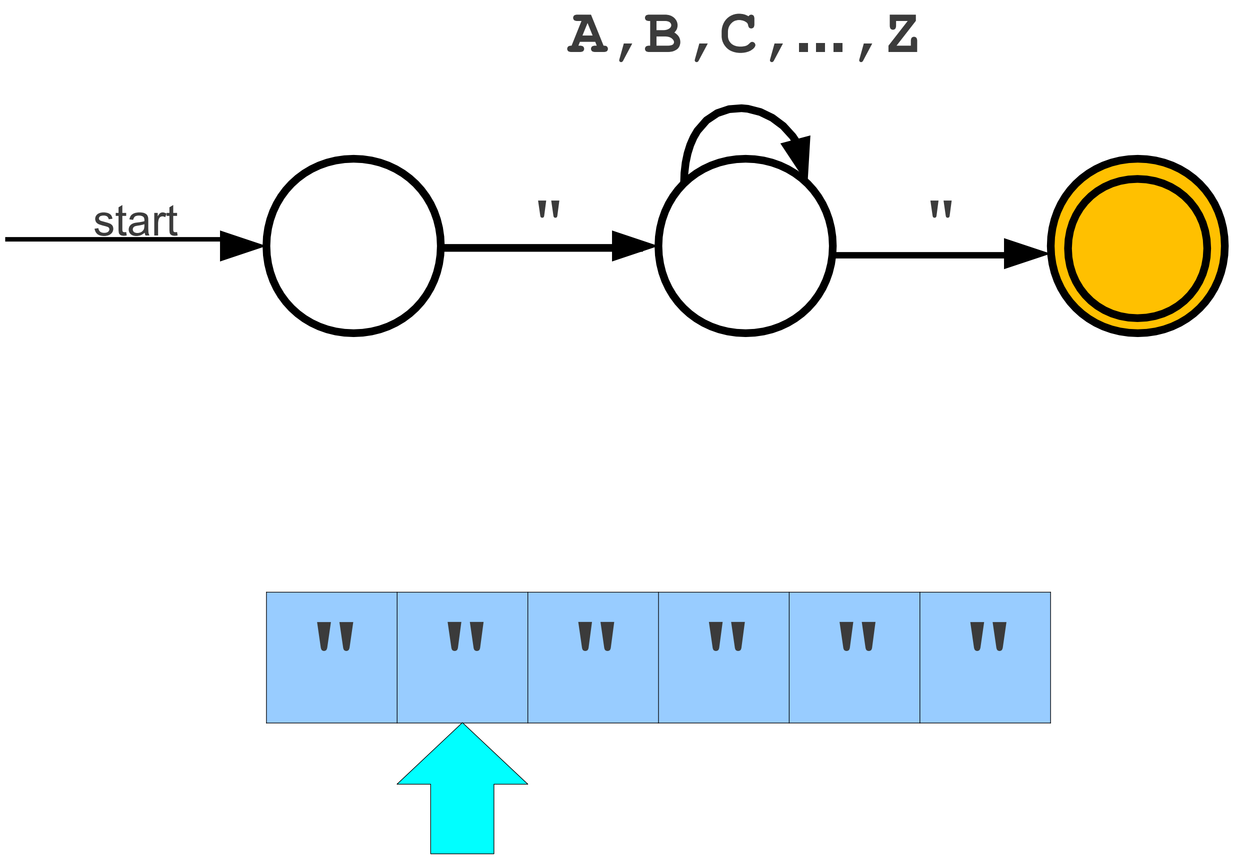


# A Simple Automaton

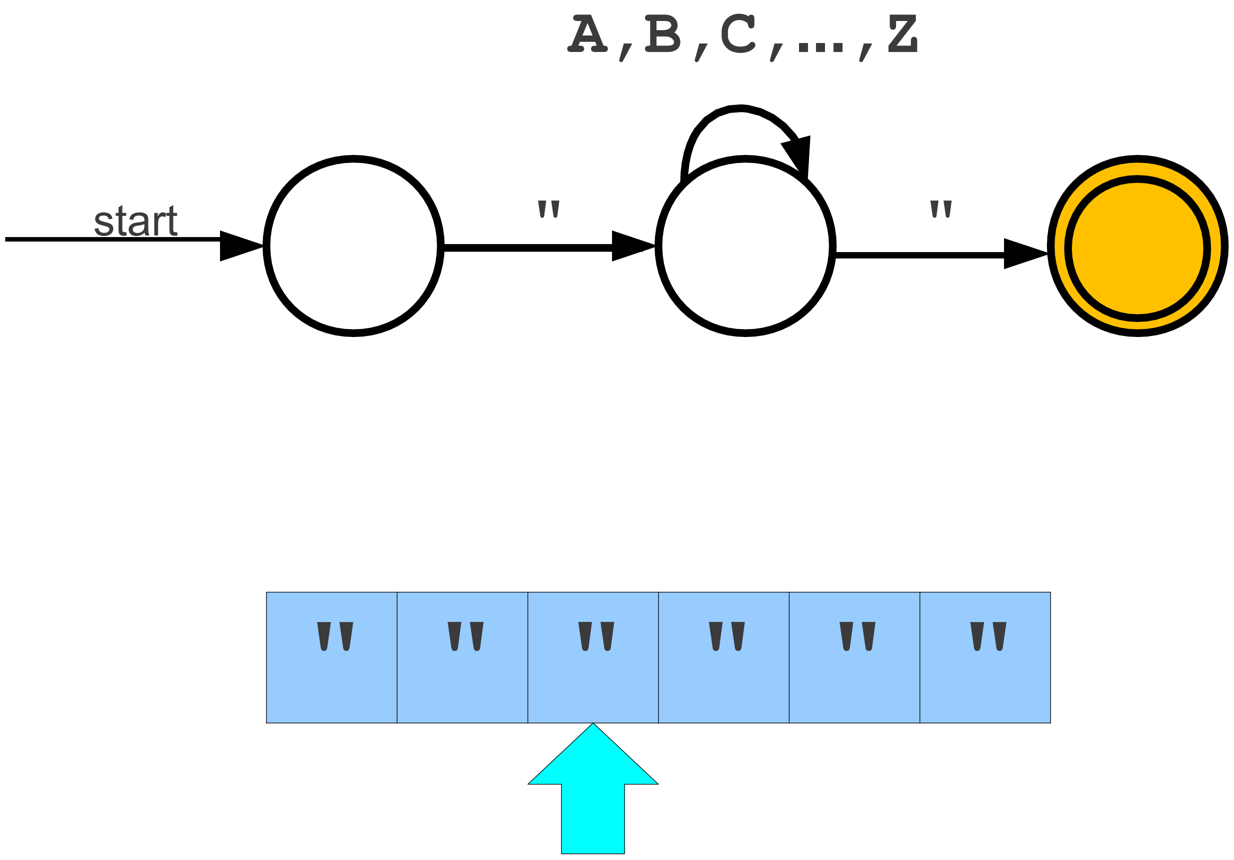


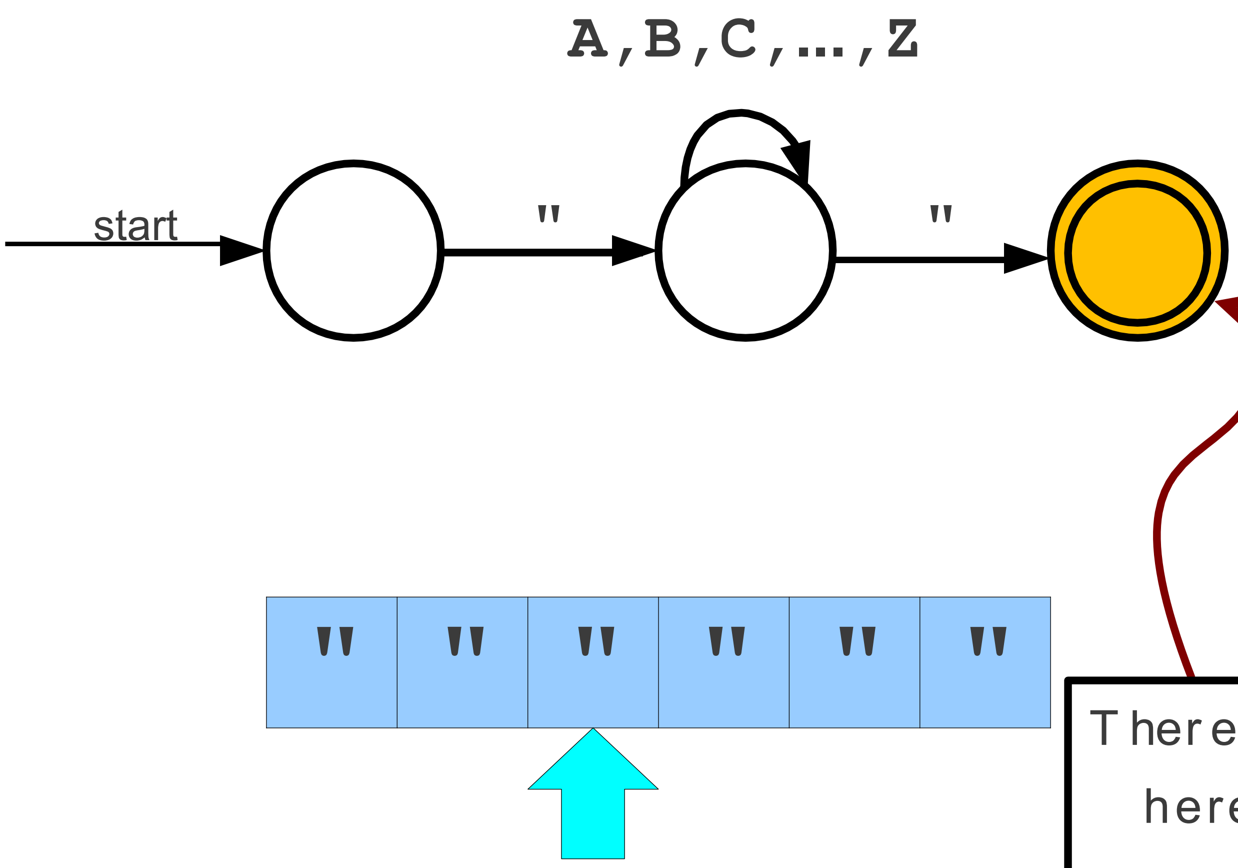
# A Simple Automaton



# A Simple Automaton

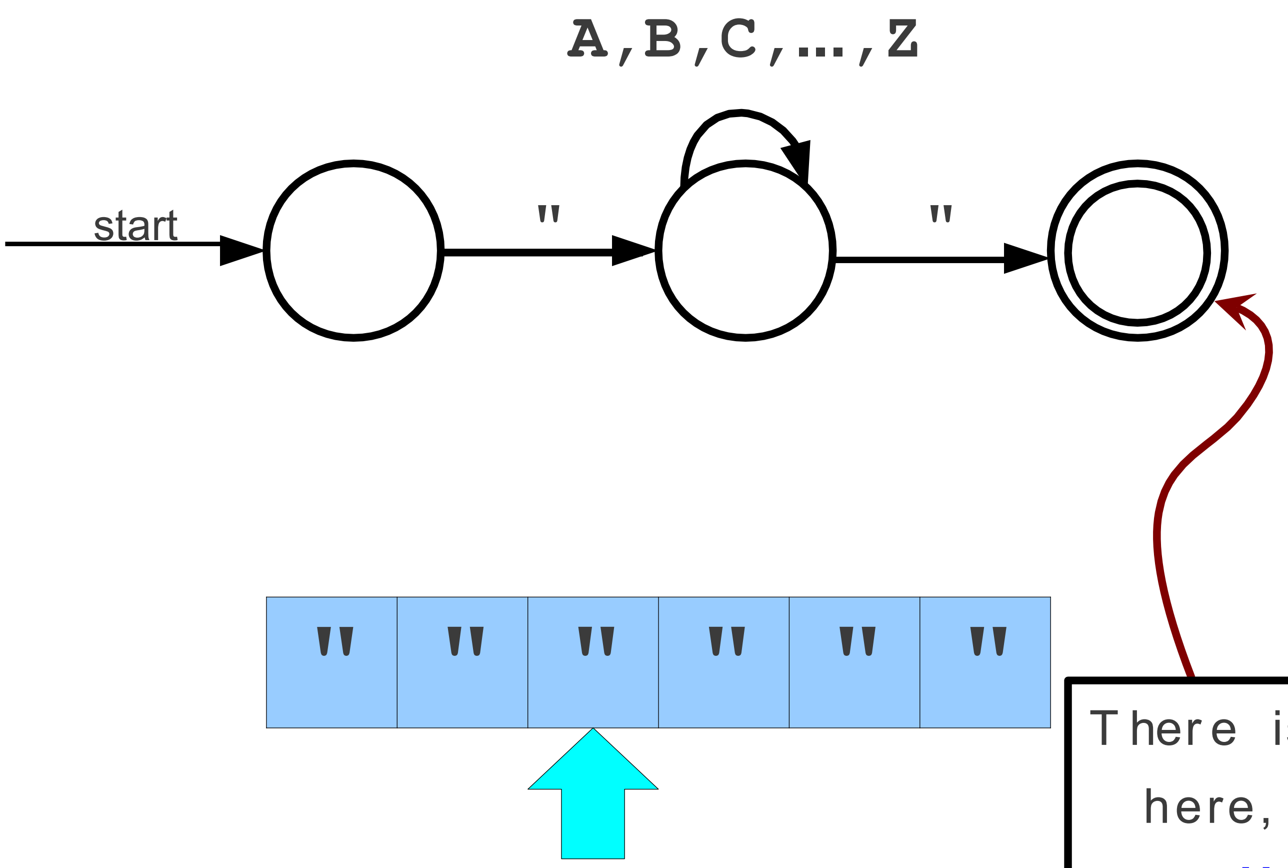


# A Simple Automaton



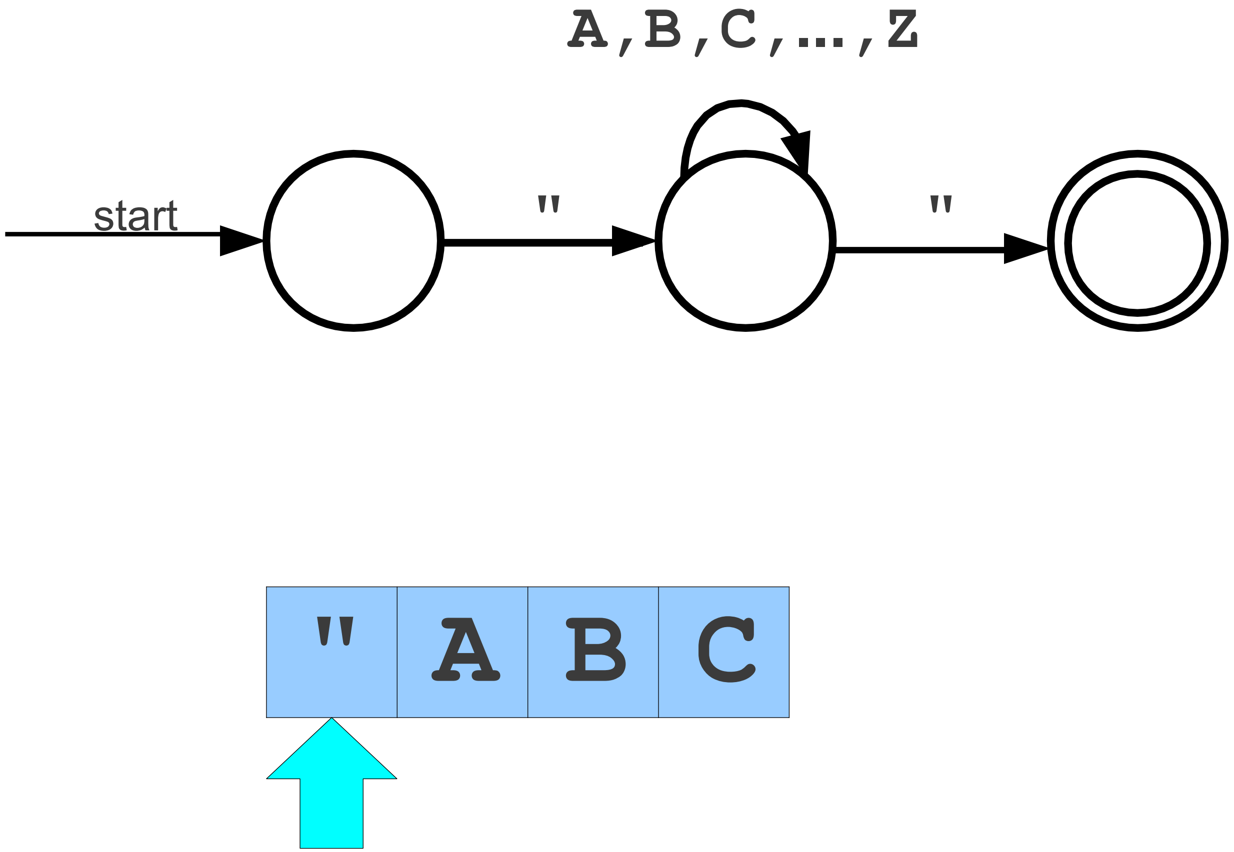

There is no transition on " here, so the automaton **dies** and rejects.

# A Simple Automaton

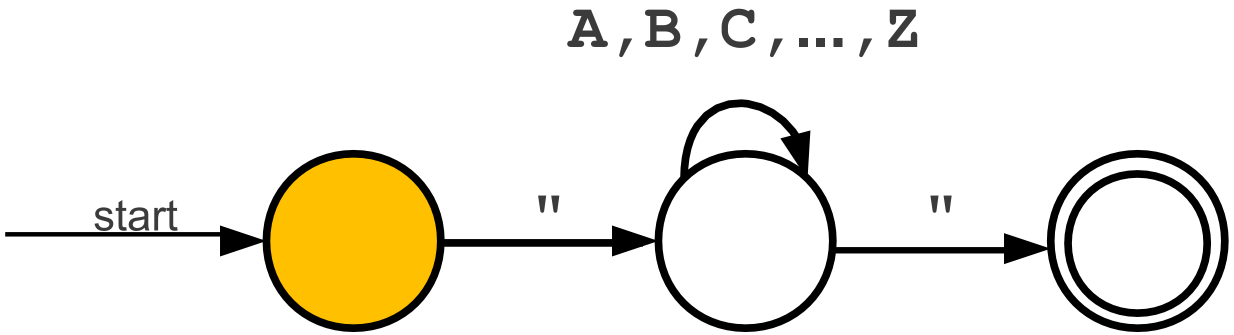


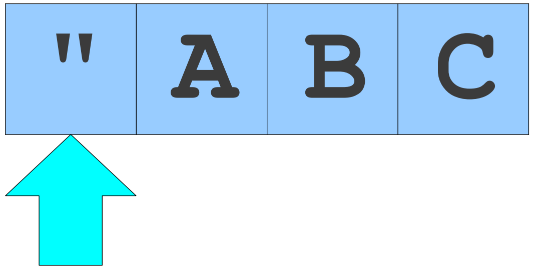

There is no transition on " here, so the automaton **dies** and rejects.

# A Simple Automaton

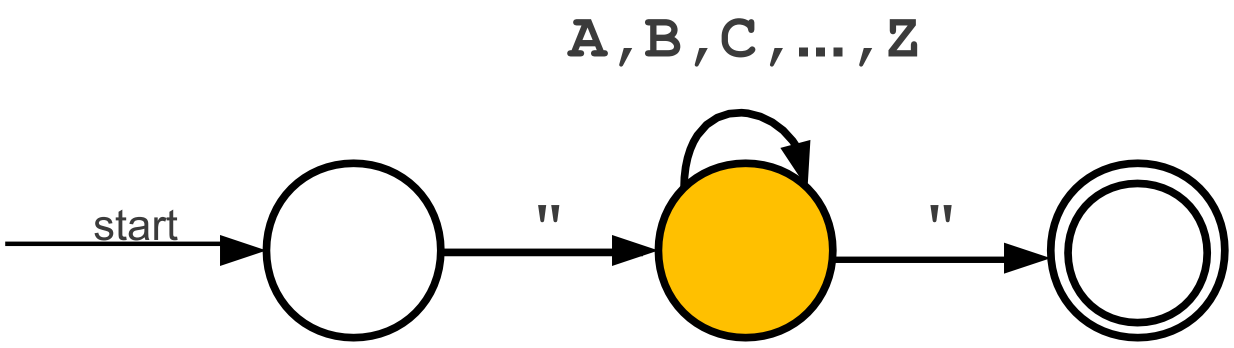


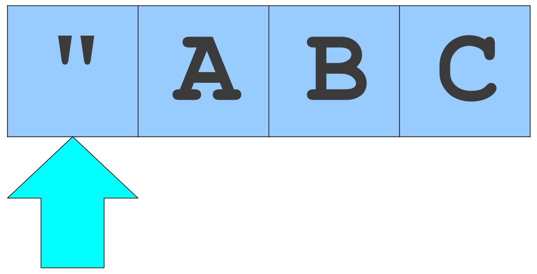
# A Simple Automaton



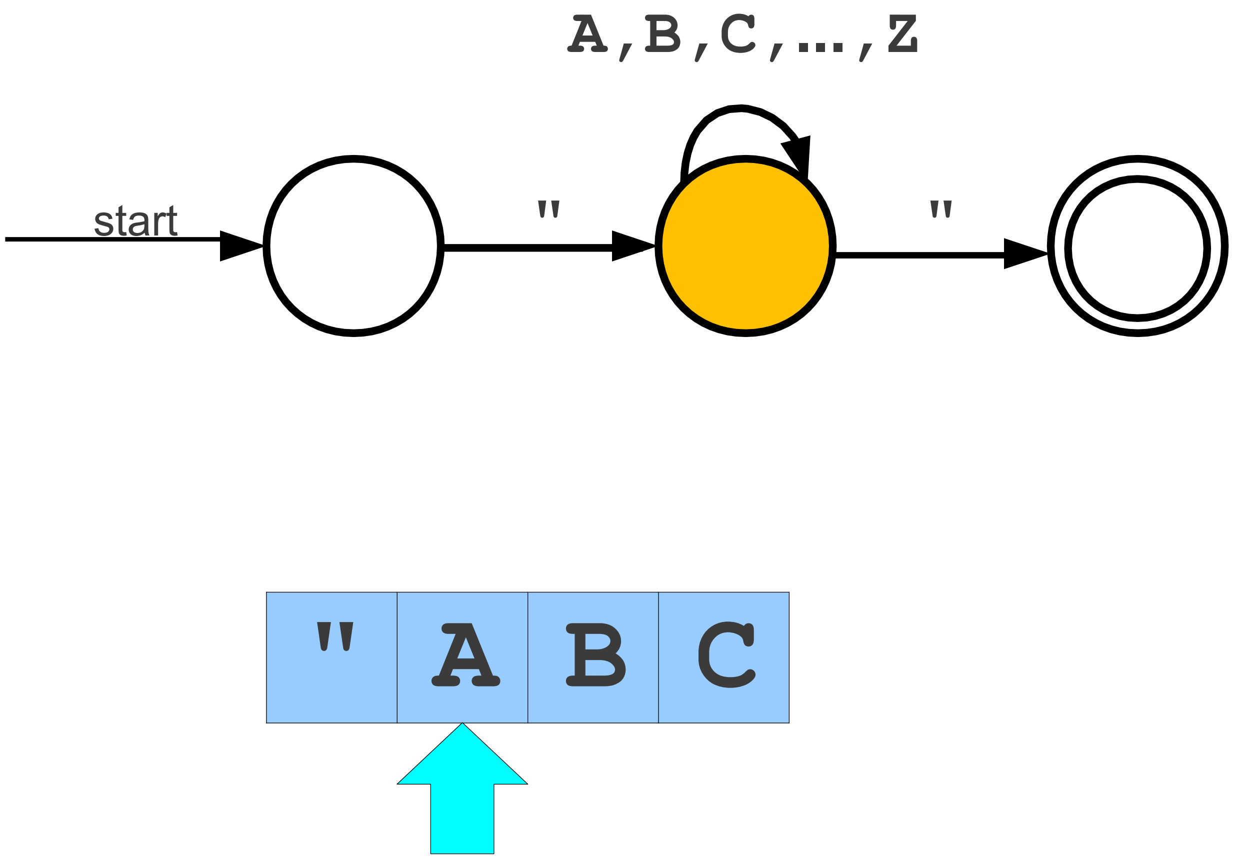



# A Simple Automaton

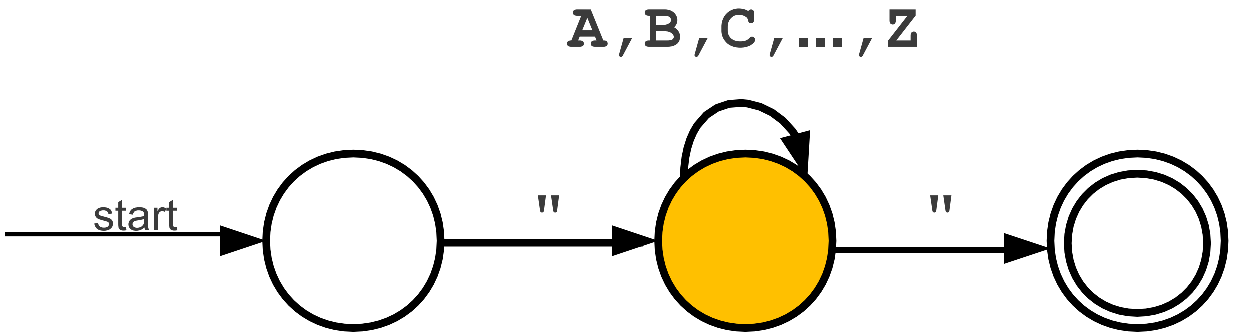


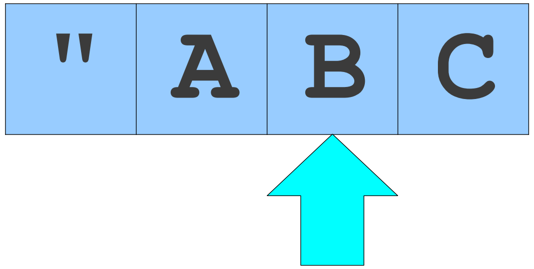



# A Simple Automaton

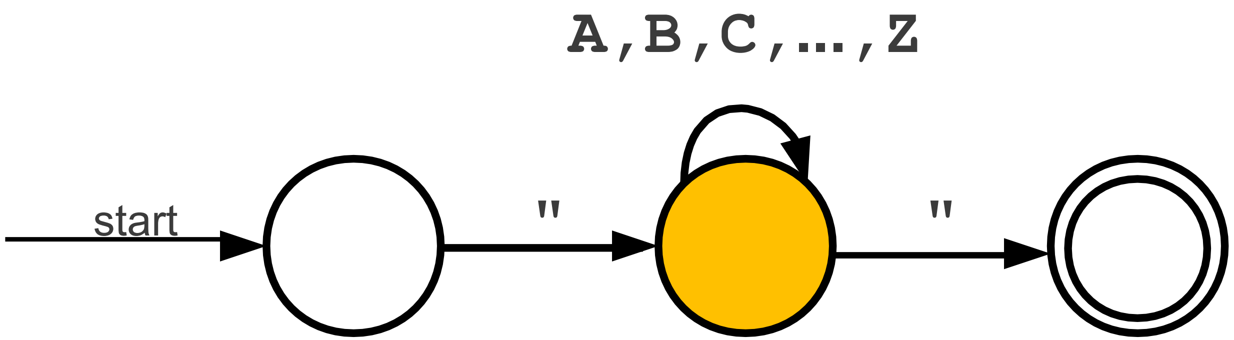


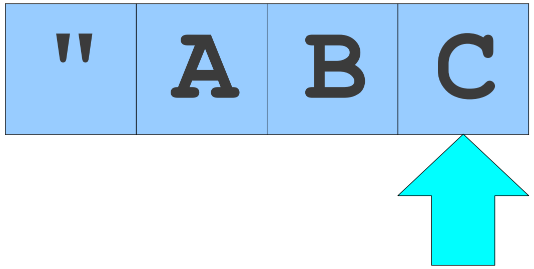
# A Simple Automaton



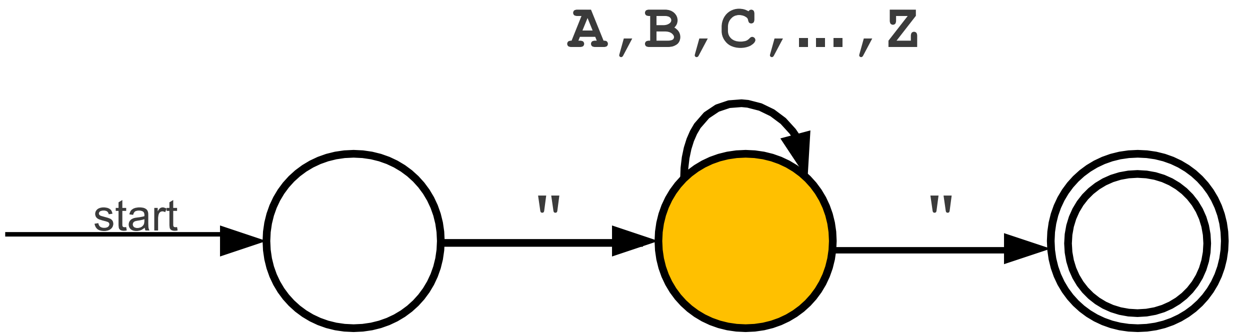



# A Simple Automaton



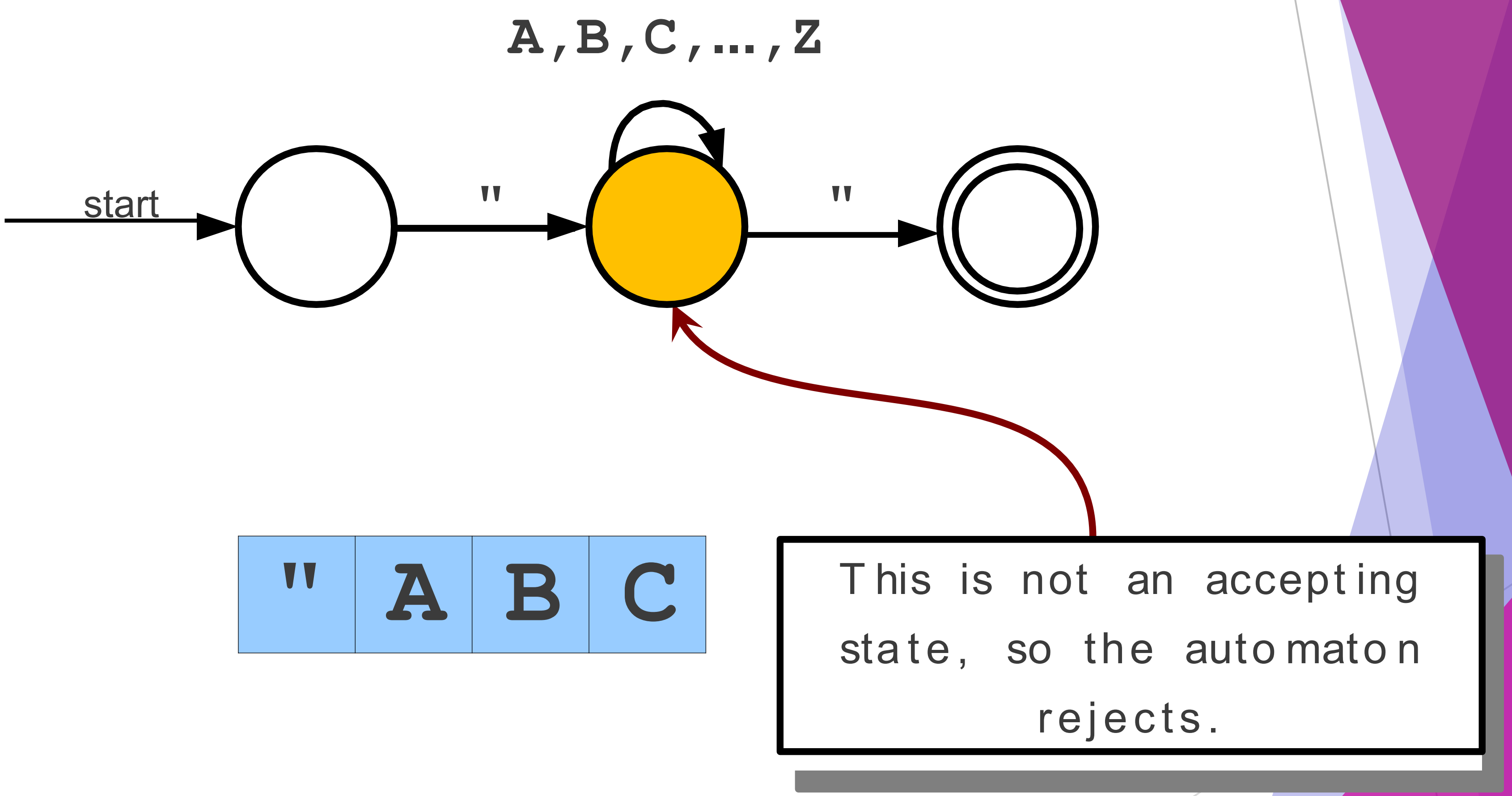



# A Simple Automaton

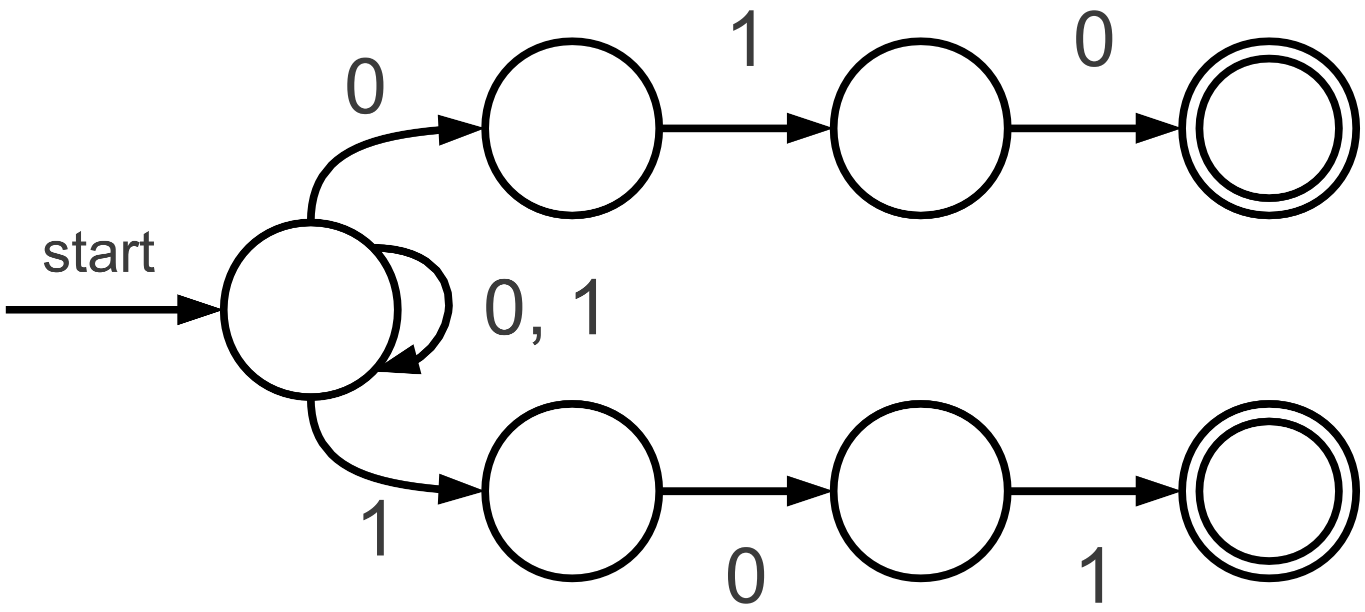



| | | | |
|---|---|---|---|
| " | A | B | C |

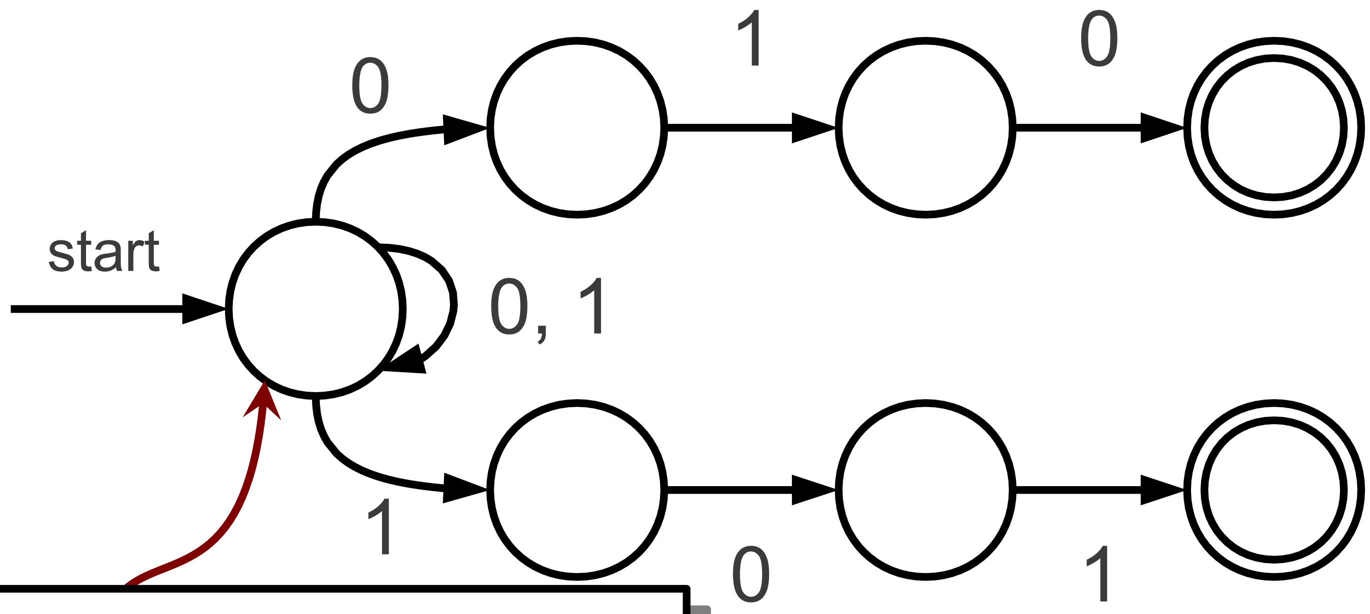
# A Simple Automaton



# A More Complex Automaton

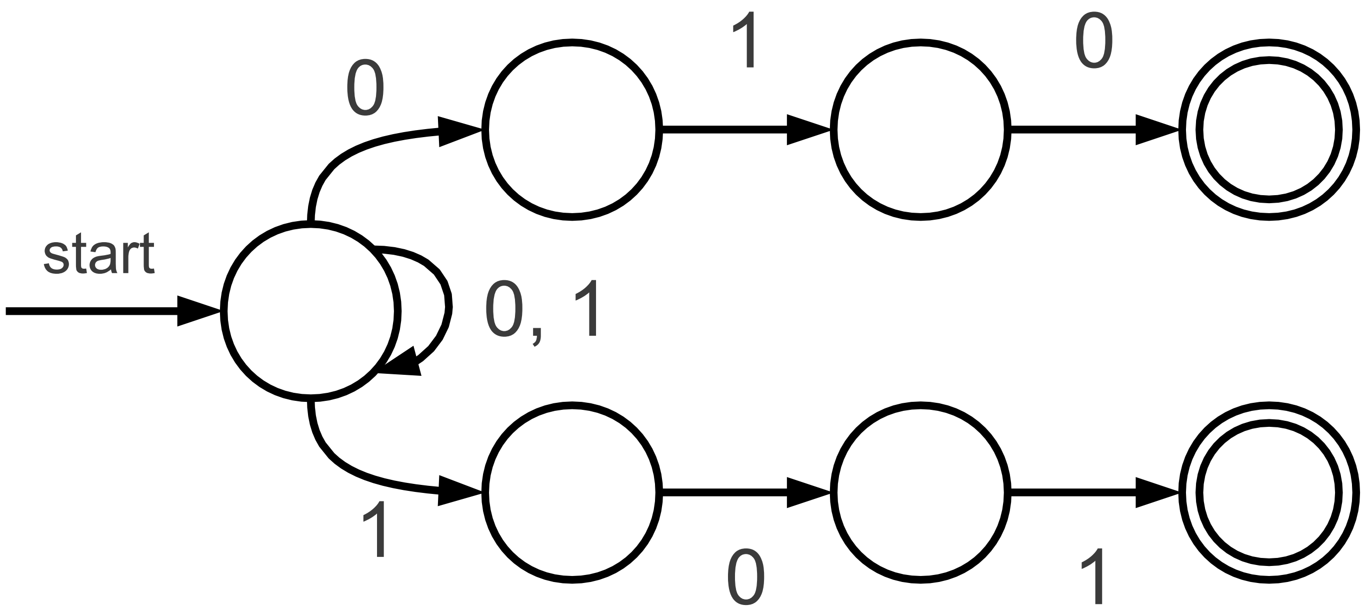


# A More Complex Automaton



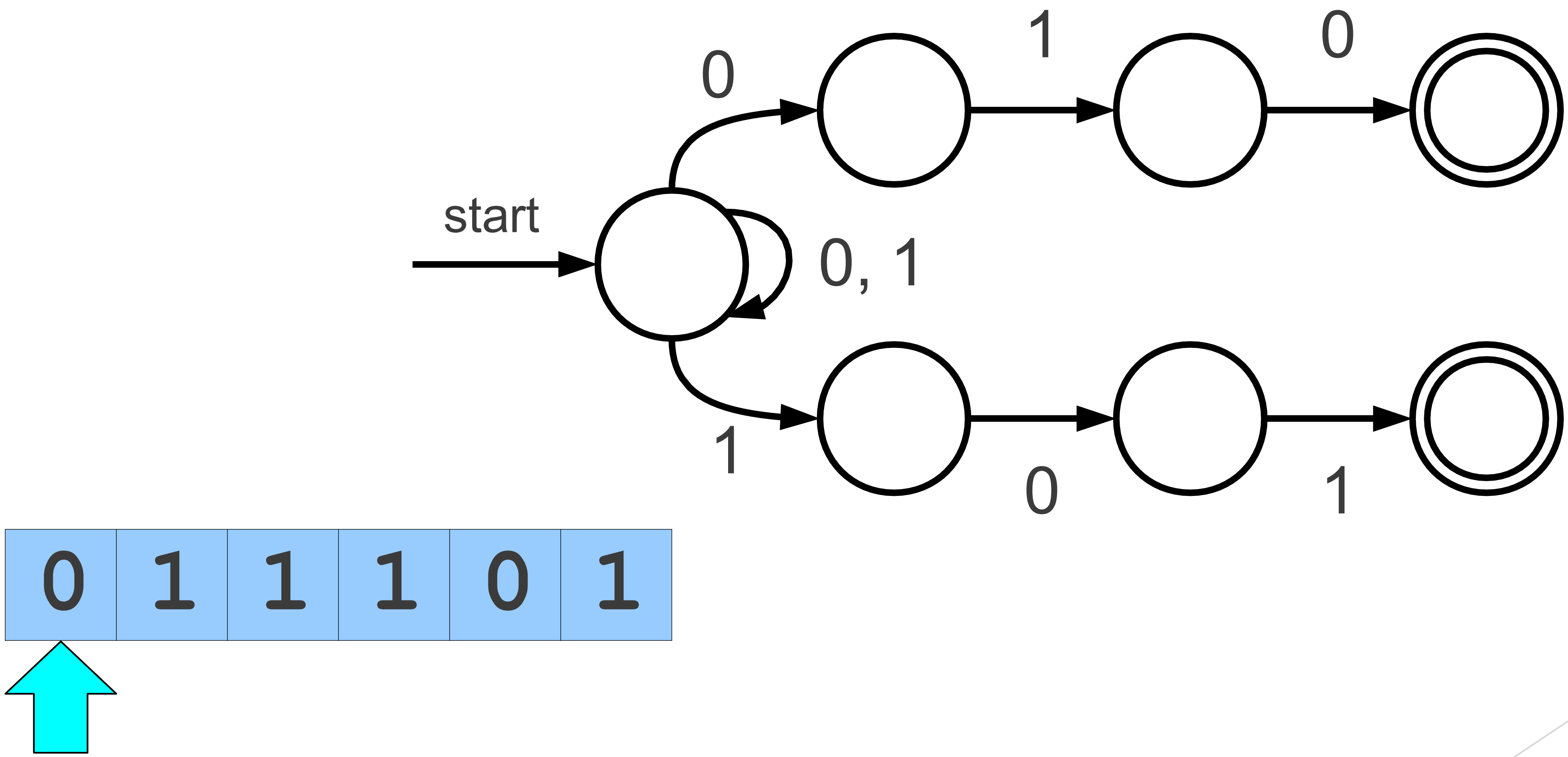

Notice that there are multiple transitions defined here on 0 and 1. If we read a 0 or 1 here, we follow *both* transitions and enter multiple states.

# A More Complex Automaton

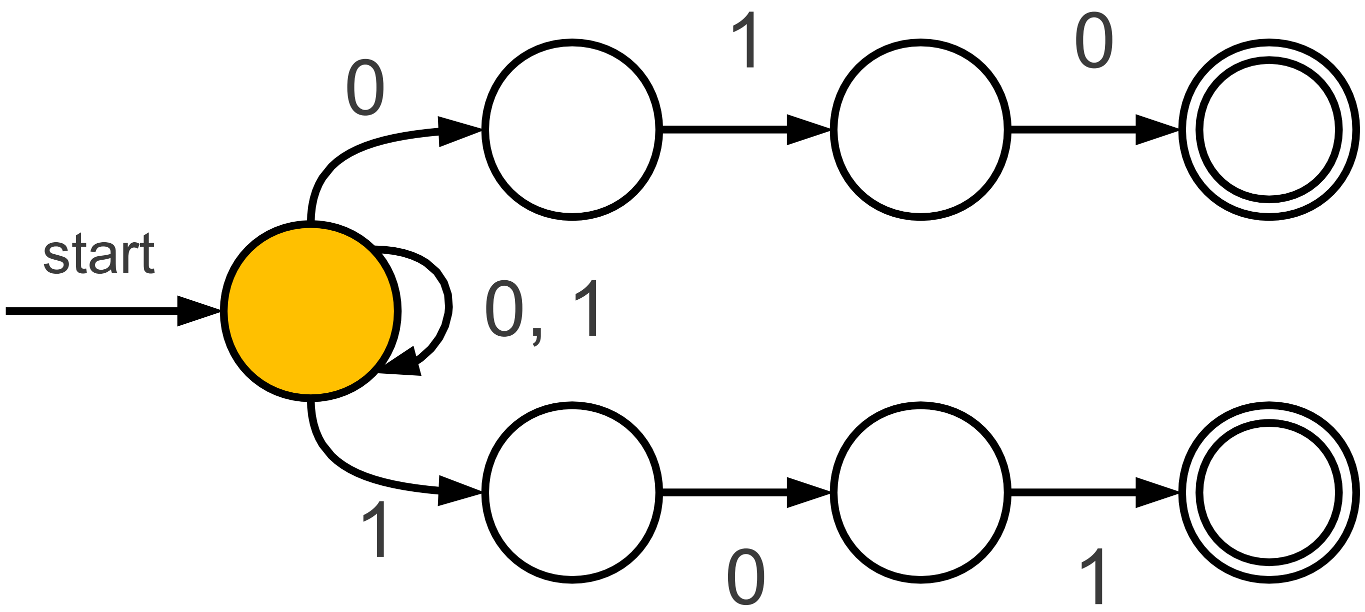



| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 |

# A More Complex Automaton

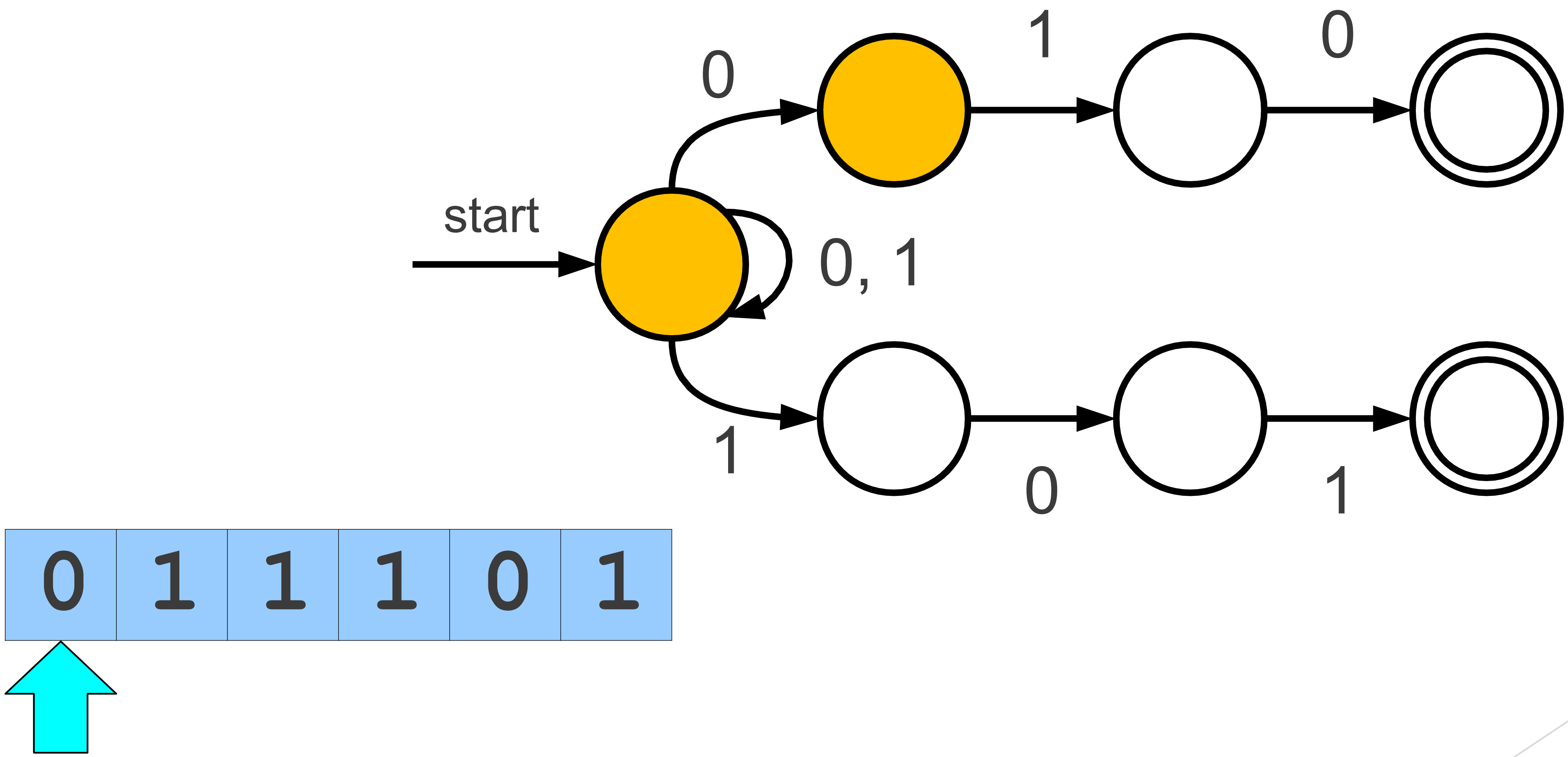


# A More Complex Automaton

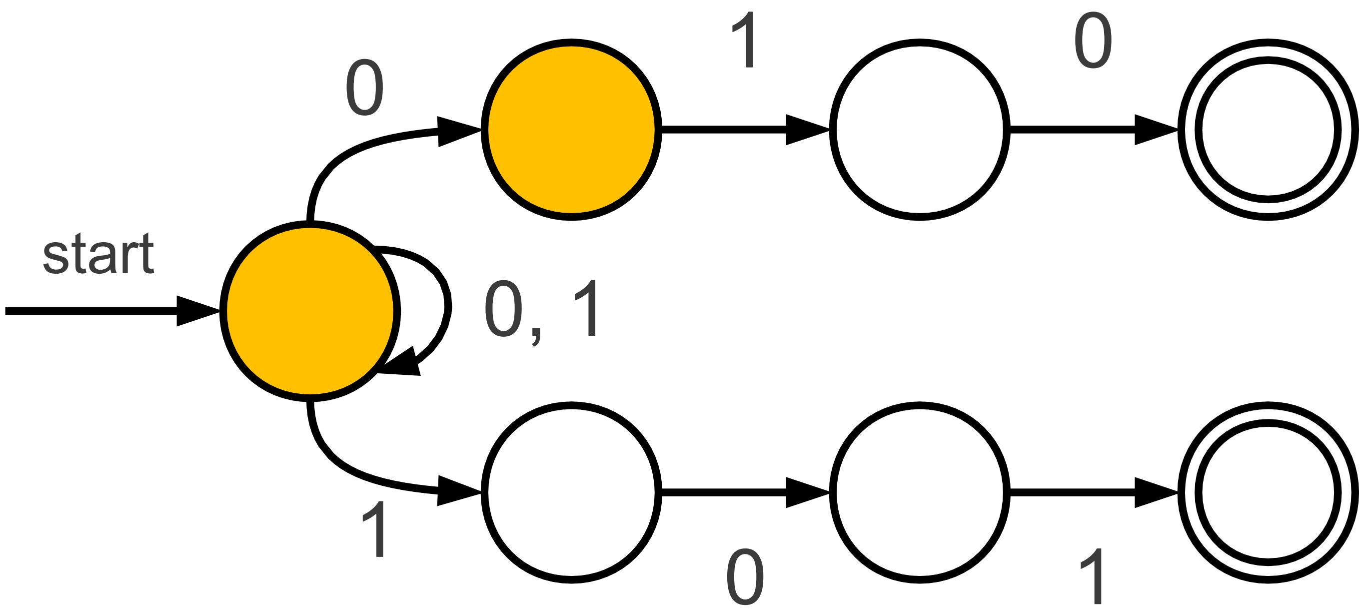


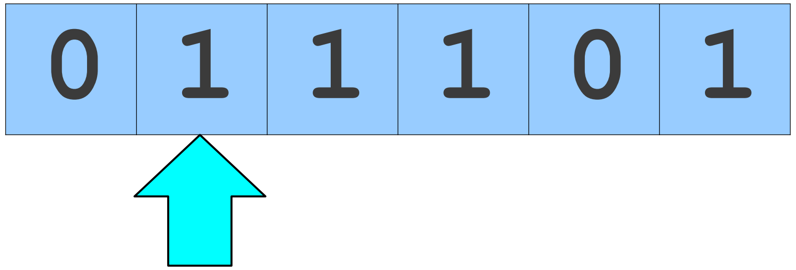

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 |

# A More Complex Automaton

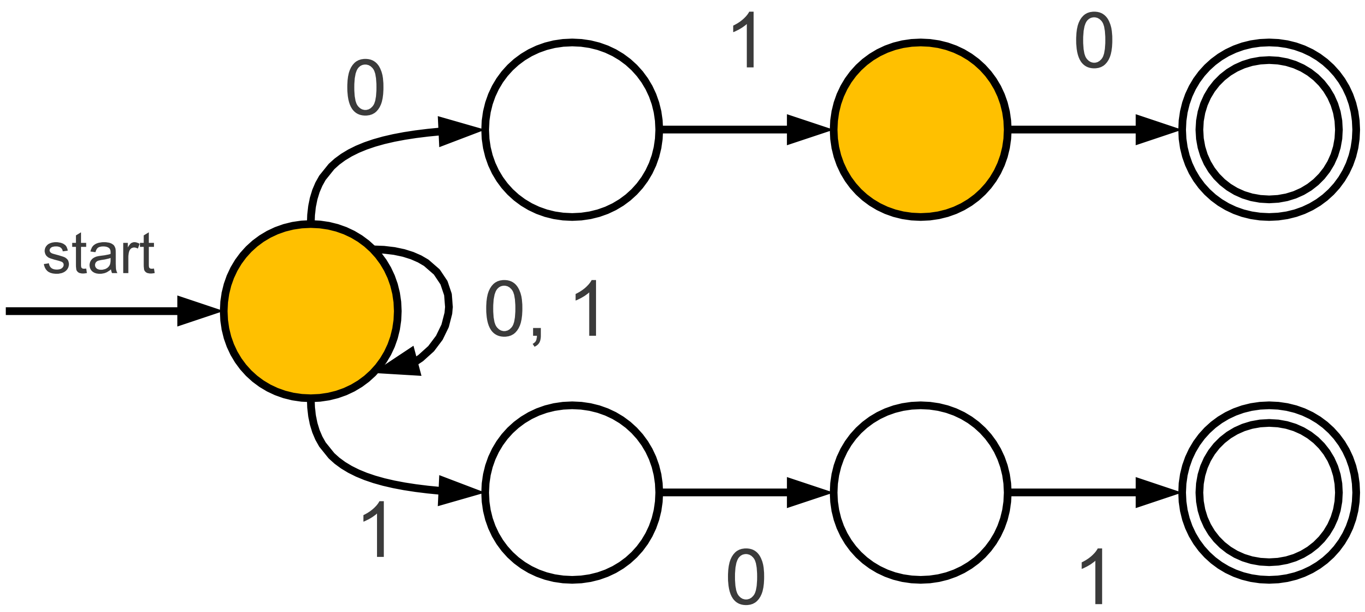


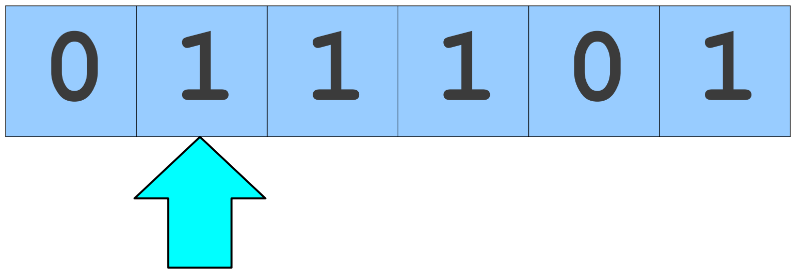
# A More Complex Automaton



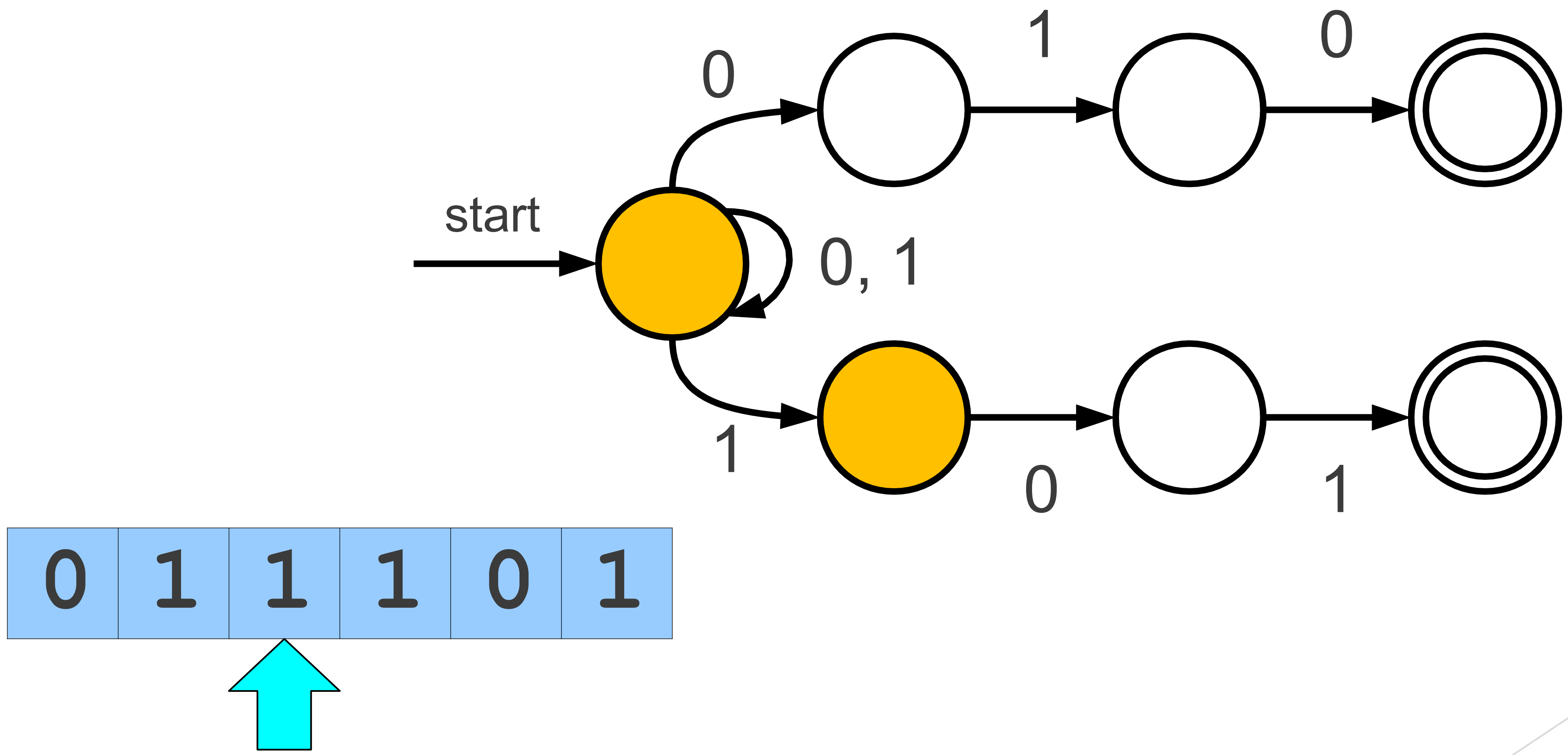



# A More Complex Automaton

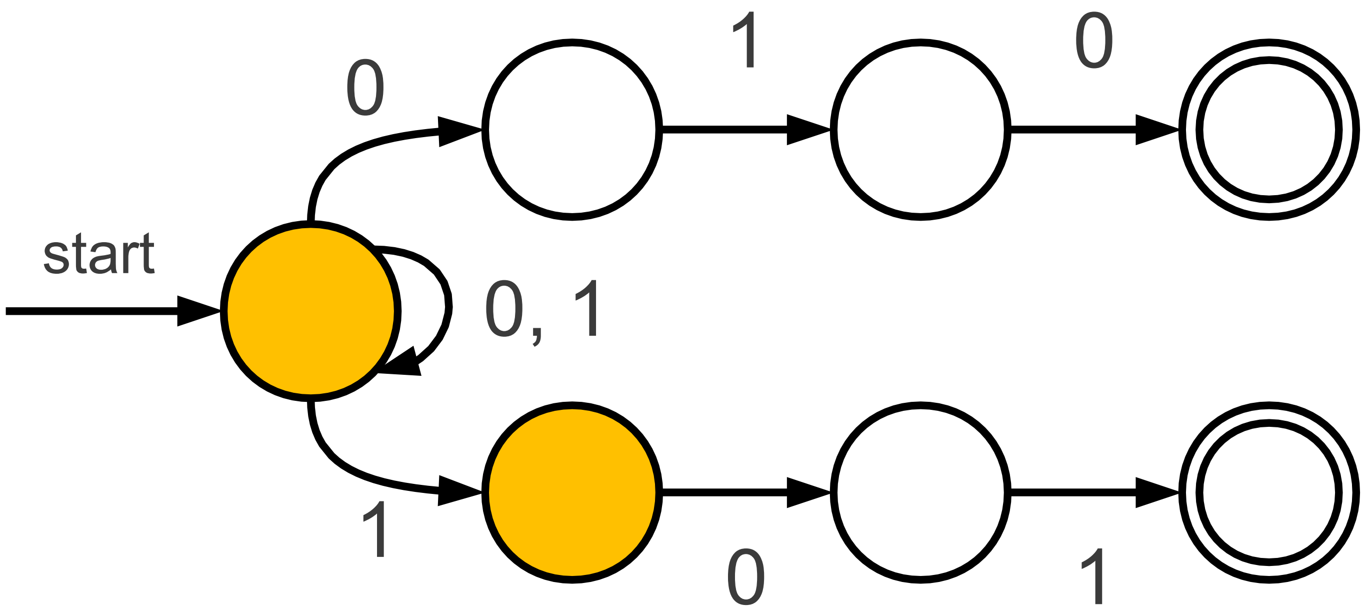


# A More Complex Automaton

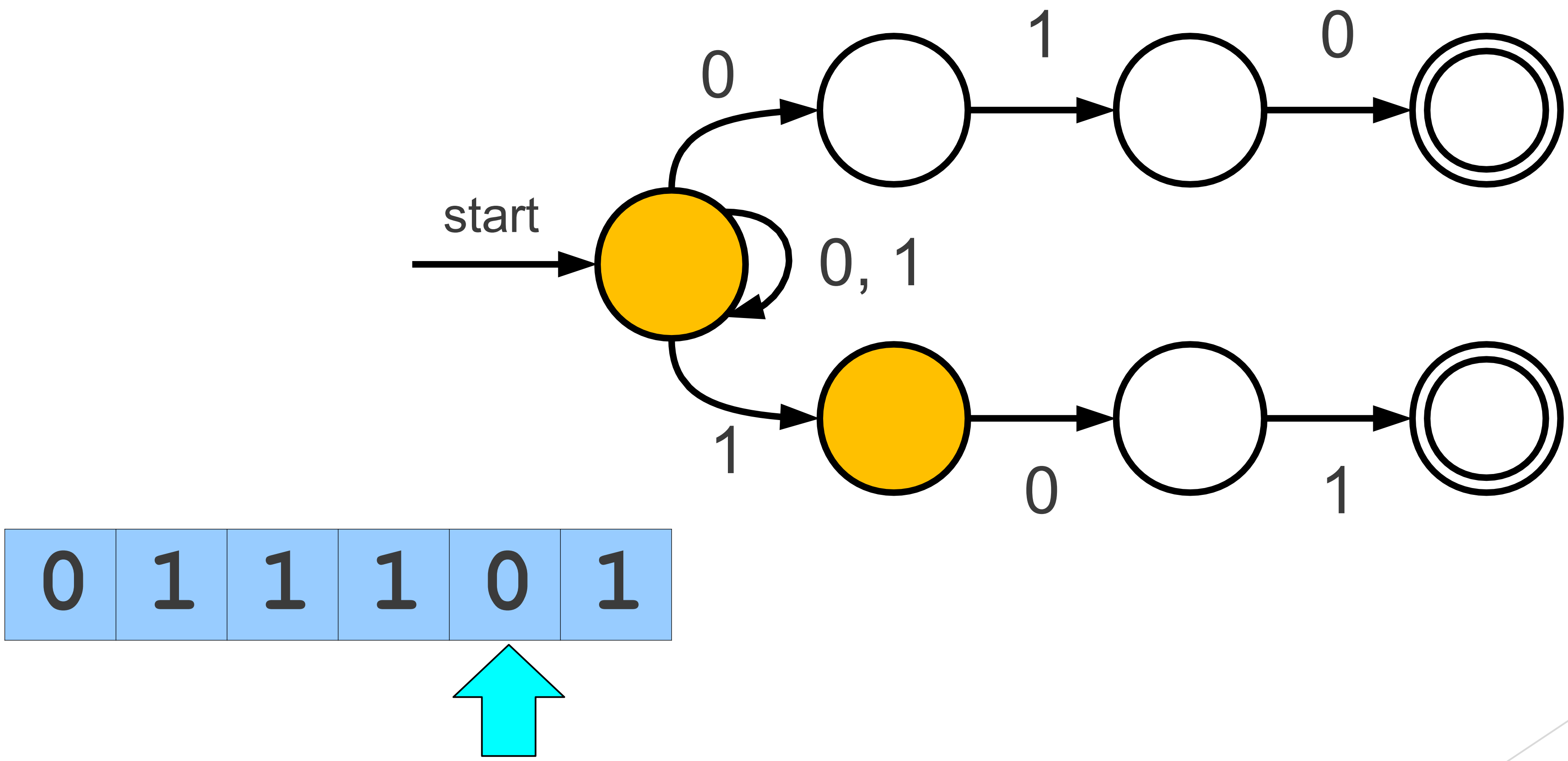


# A More Complex Automaton

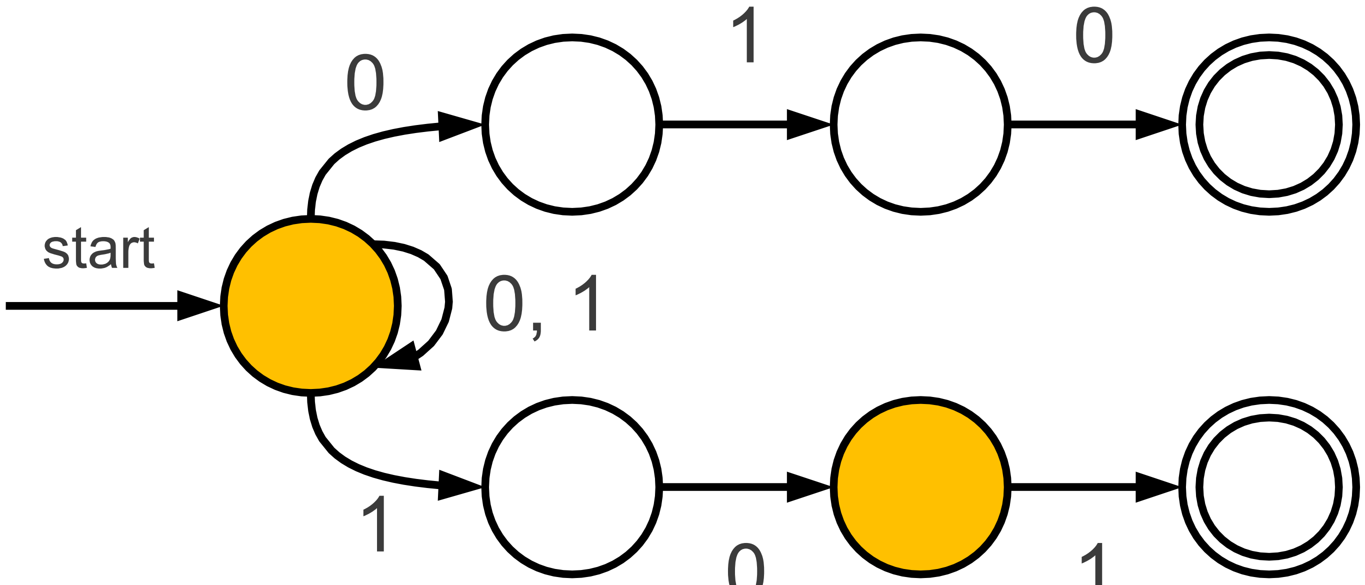



| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 |

# A More Complex Automaton

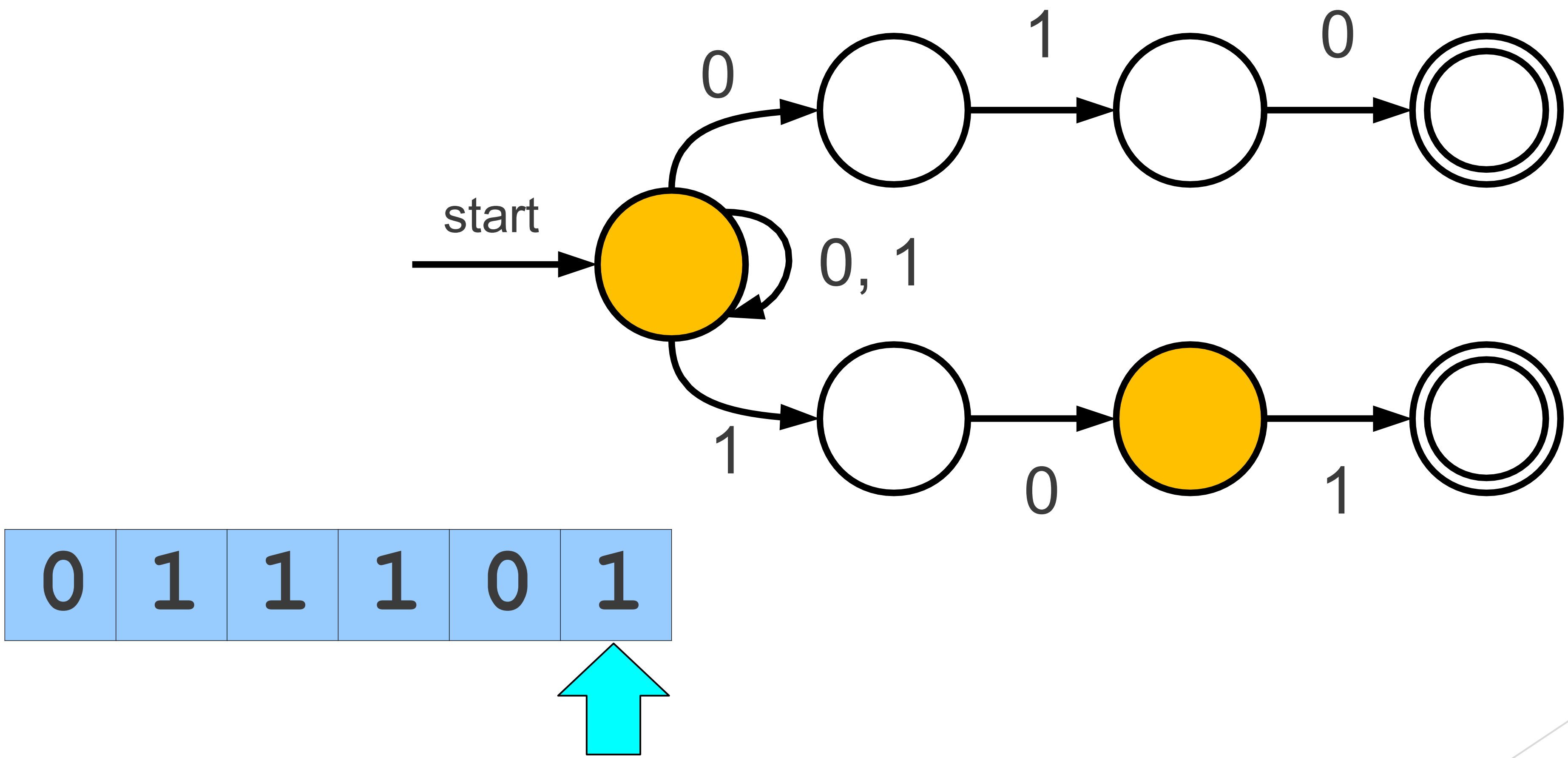


# A More Complex Automaton

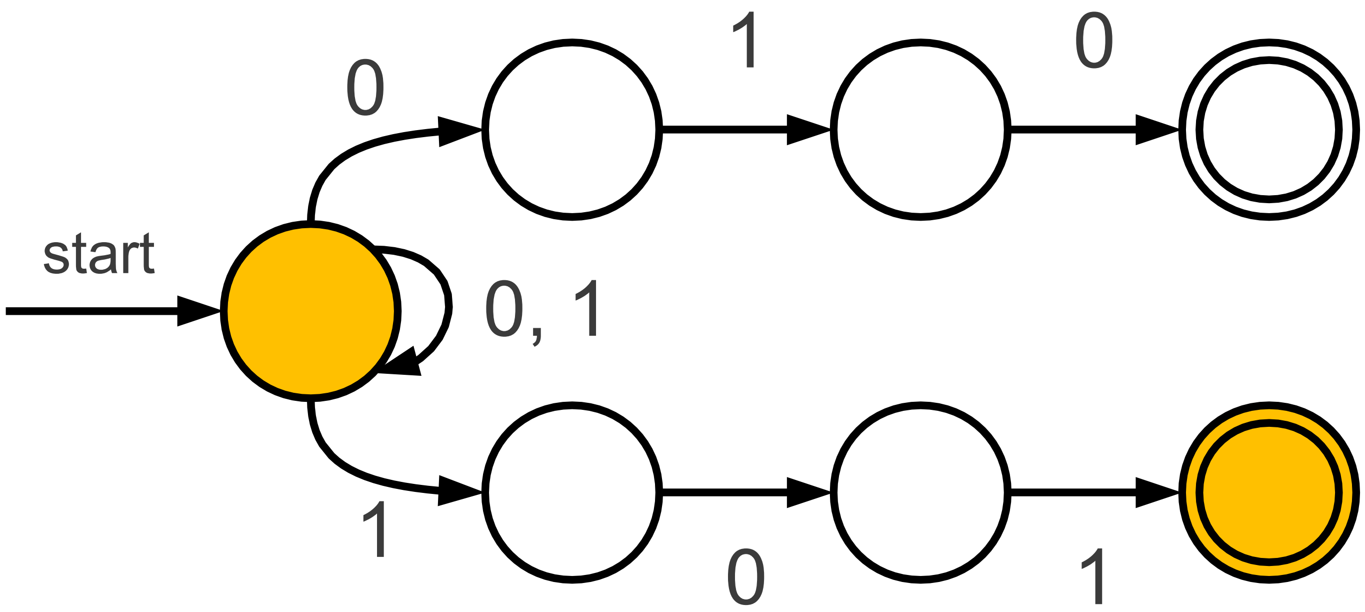



| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 |

# A More Complex Automaton

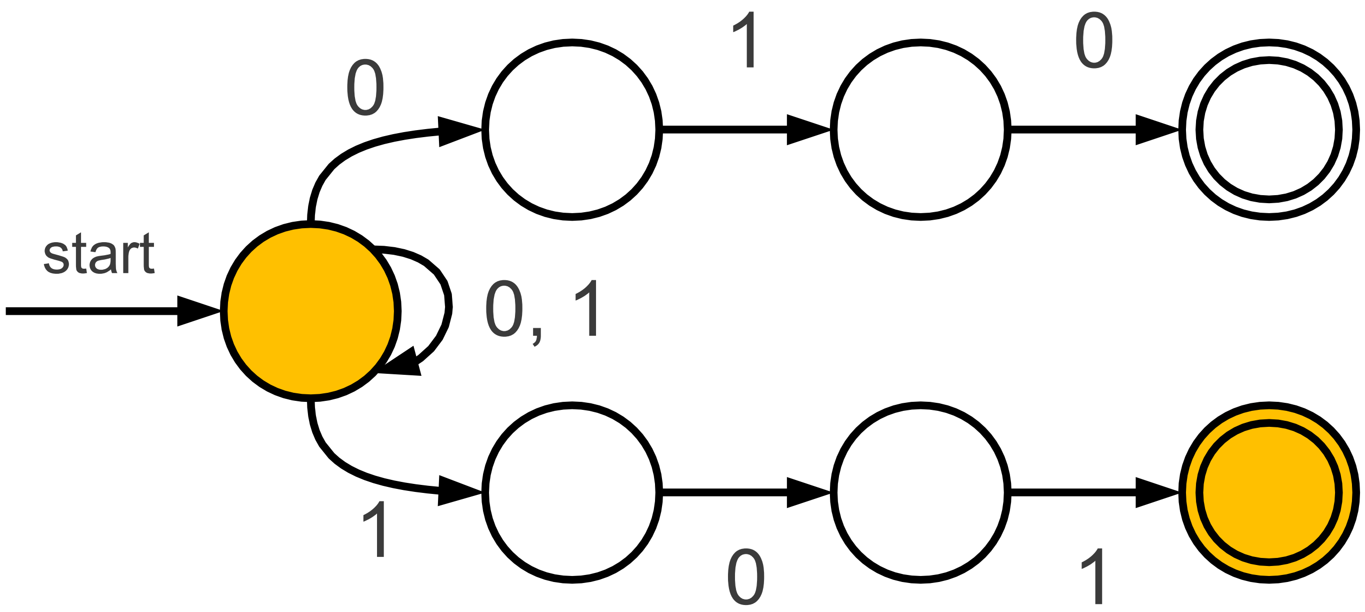


# A More Complex Automaton



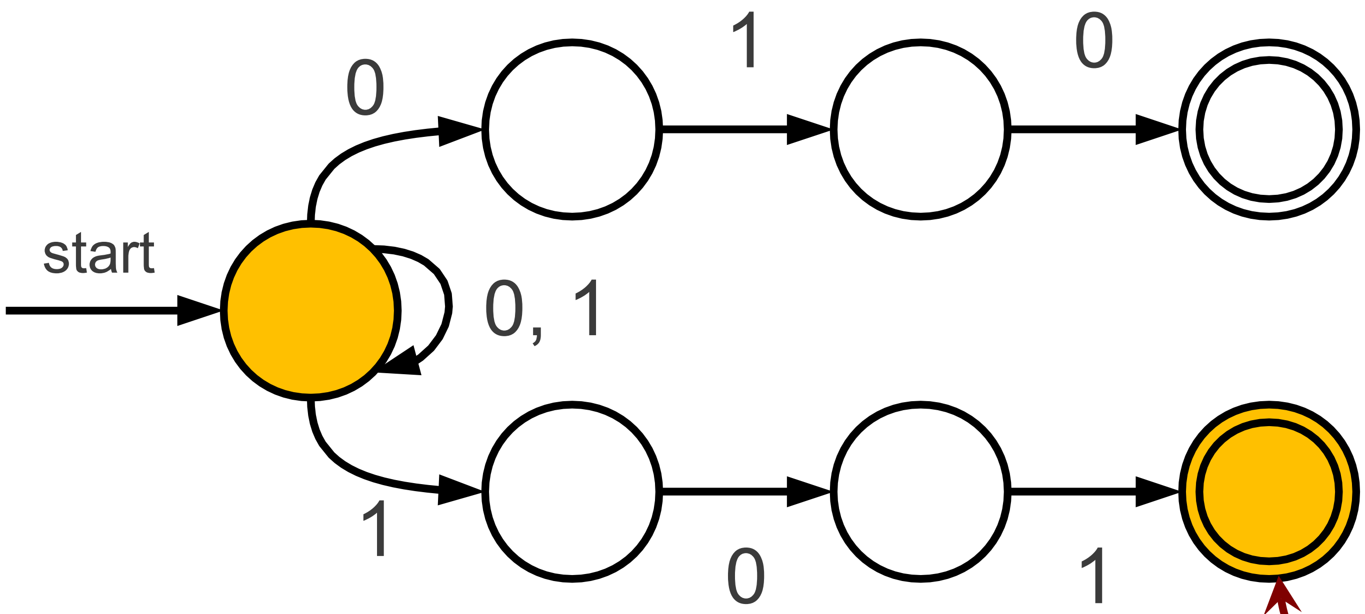

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 |

# A More Complex Automaton



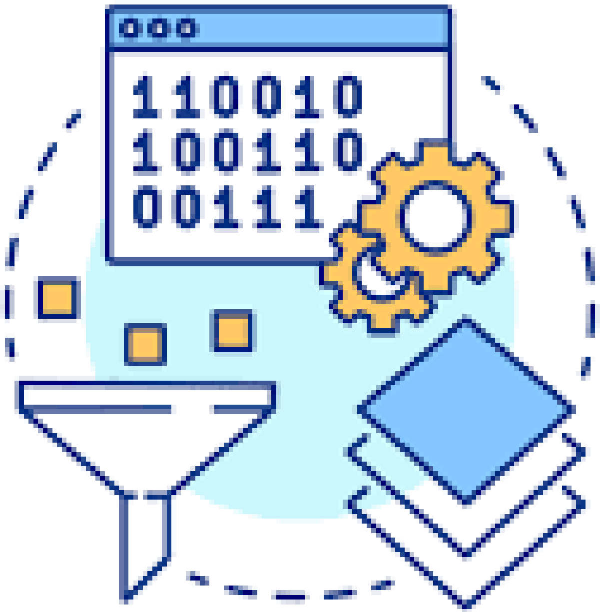

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 |

# A More Complex Automaton




| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 |

Since we are in at least one accepting state, the automaton accepts.




# Deterministic Finite Automata (DFA)

# The Concept of DFA

DFA: Automata where the next state is uniquely given by the current state and the current input character.

**Definition of a DFA:**

A DFA (Deterministic Finite Automation) M consists of

1. A set of states Q
2. an alphabet $\Sigma$
3. a transition function $\delta : Q \times \Sigma \rightarrow Q$
4. a start state $q_0 \in Q$
5. and a set of accepting states $F \subset Q$

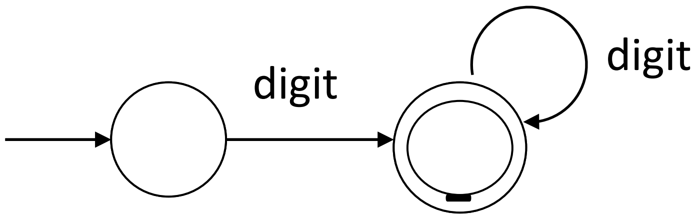
# Examples of DFA: Example 1

digit = [0-9]

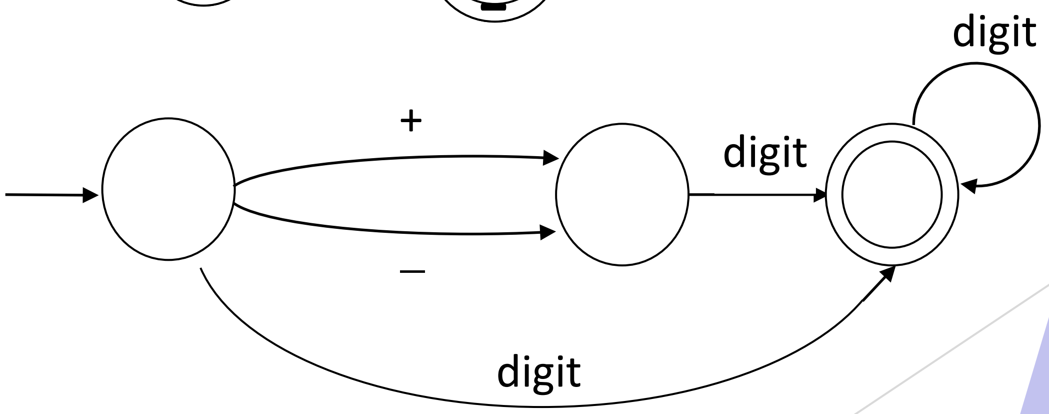
nat = digit+

signedNat = (+|-)? nat

A DFA of **nat**:




A DFA of **signedNat**:



# Examples of DFA: Example 2

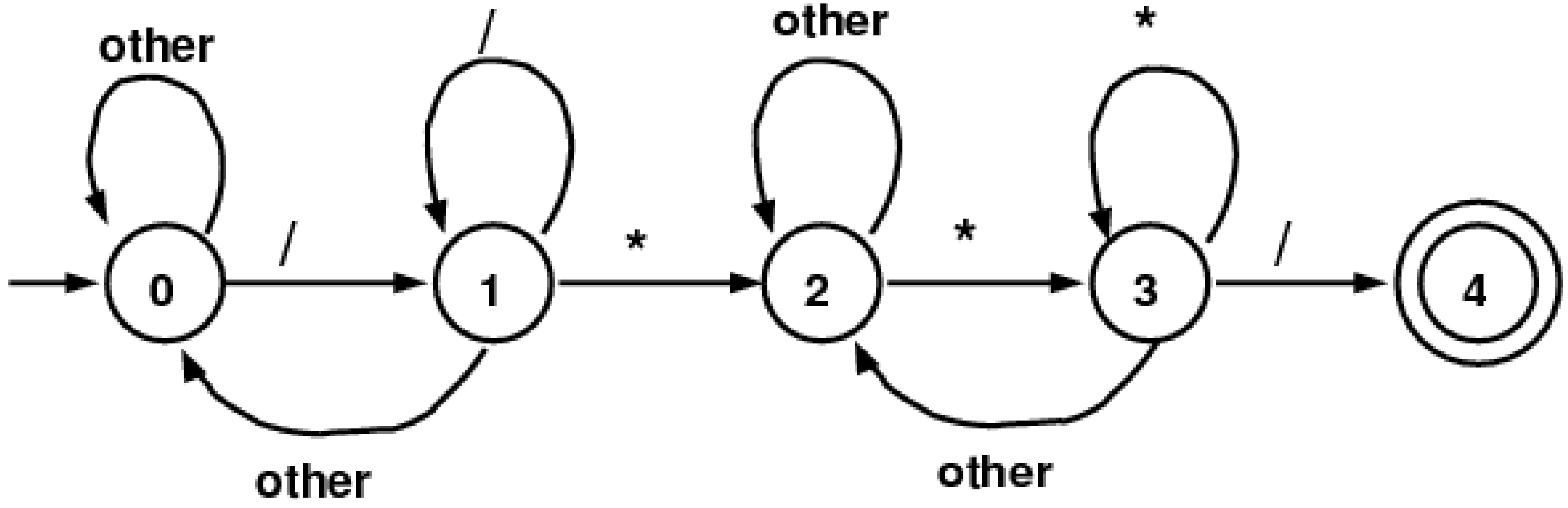
- Draw a DFA of C++ Long Comments

  /* This is a long comment in C++ */

# Examples of DFA

- Draw a DFA of C++ Long Comments

/* This is a long comment in C++ */




Note : It is easier than writing it down as a regular expression.

# Accept/Reject

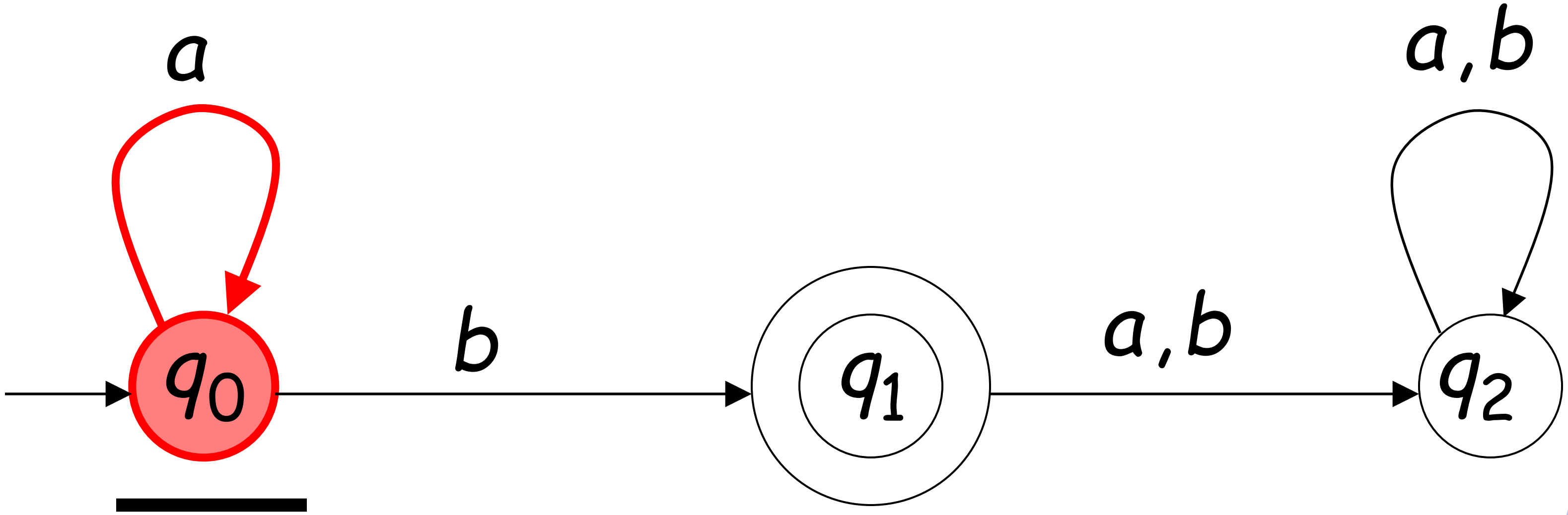
## To accept a string:

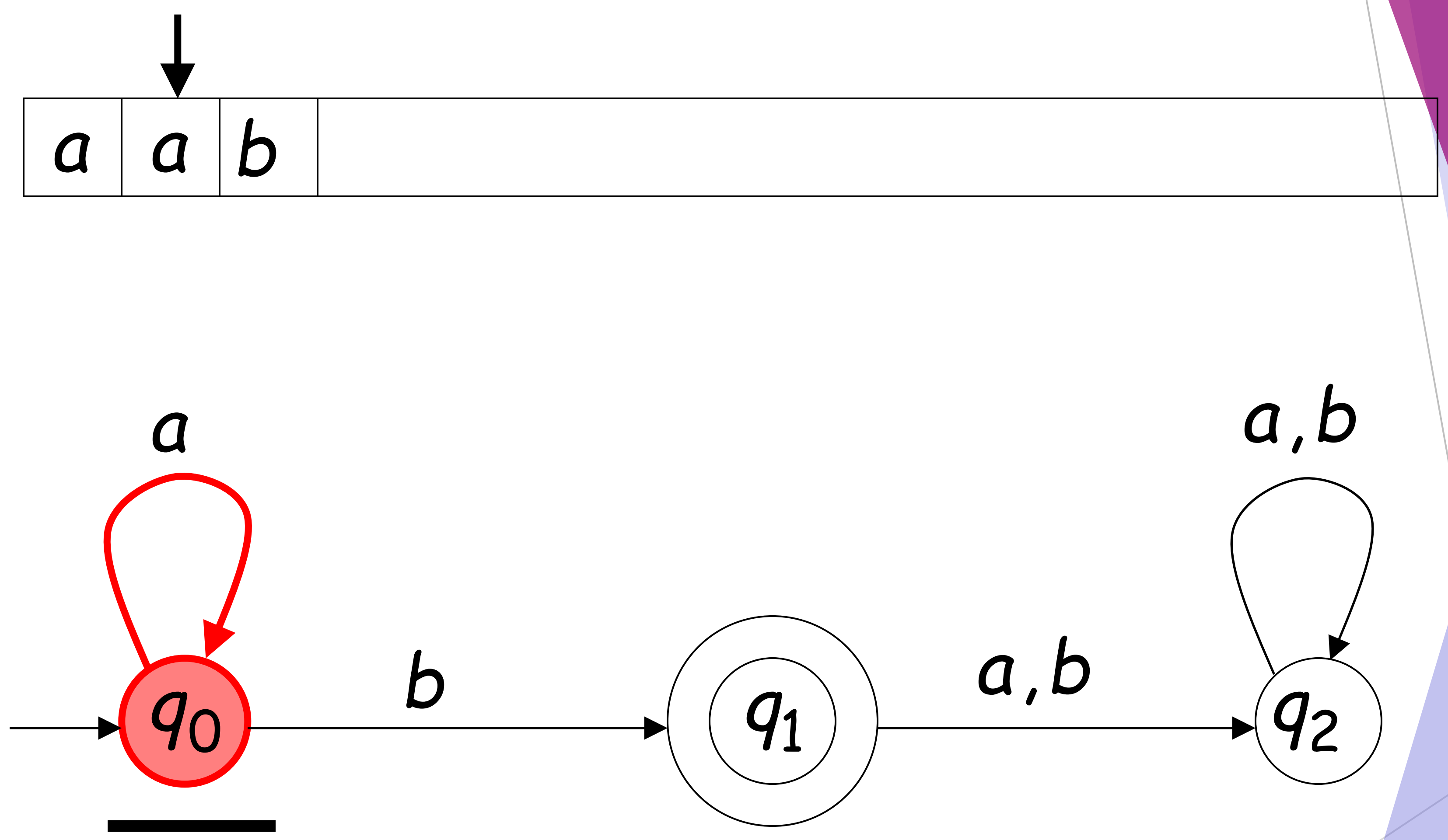
all the input string is scanned and the last state is accepting

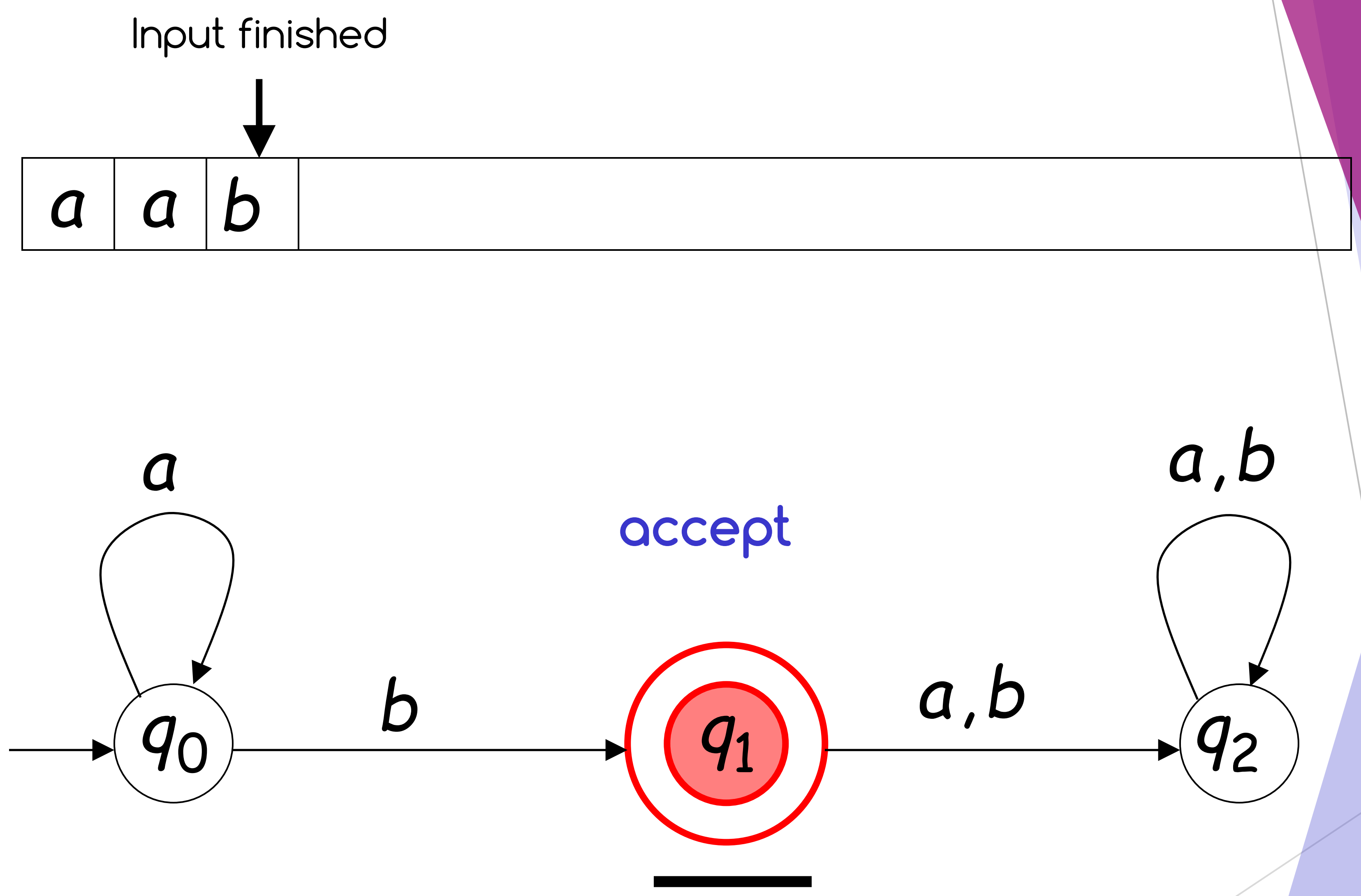
## To reject a string:

all the input string is scanned and the last state is non-accepting

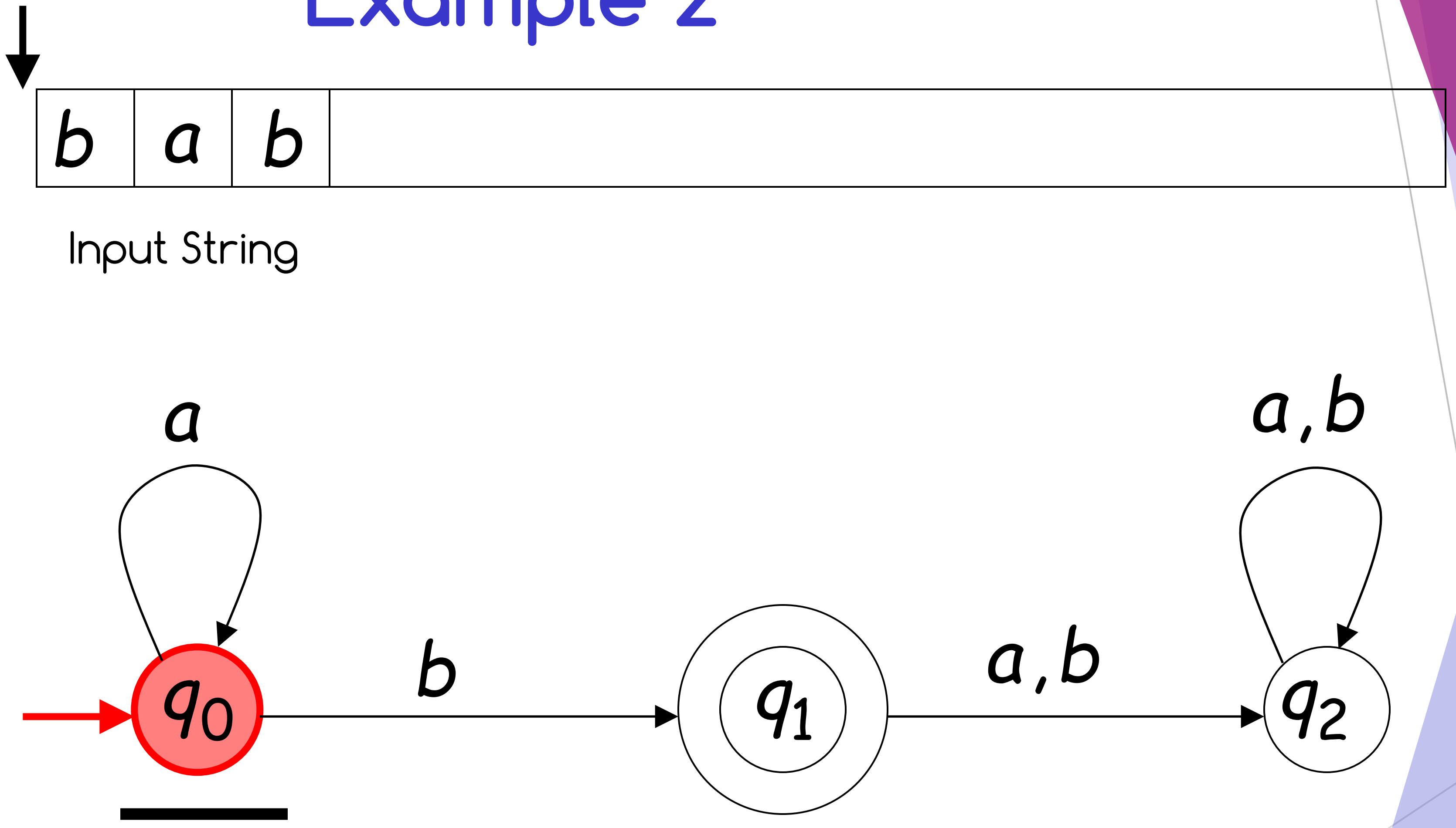
# Example 1

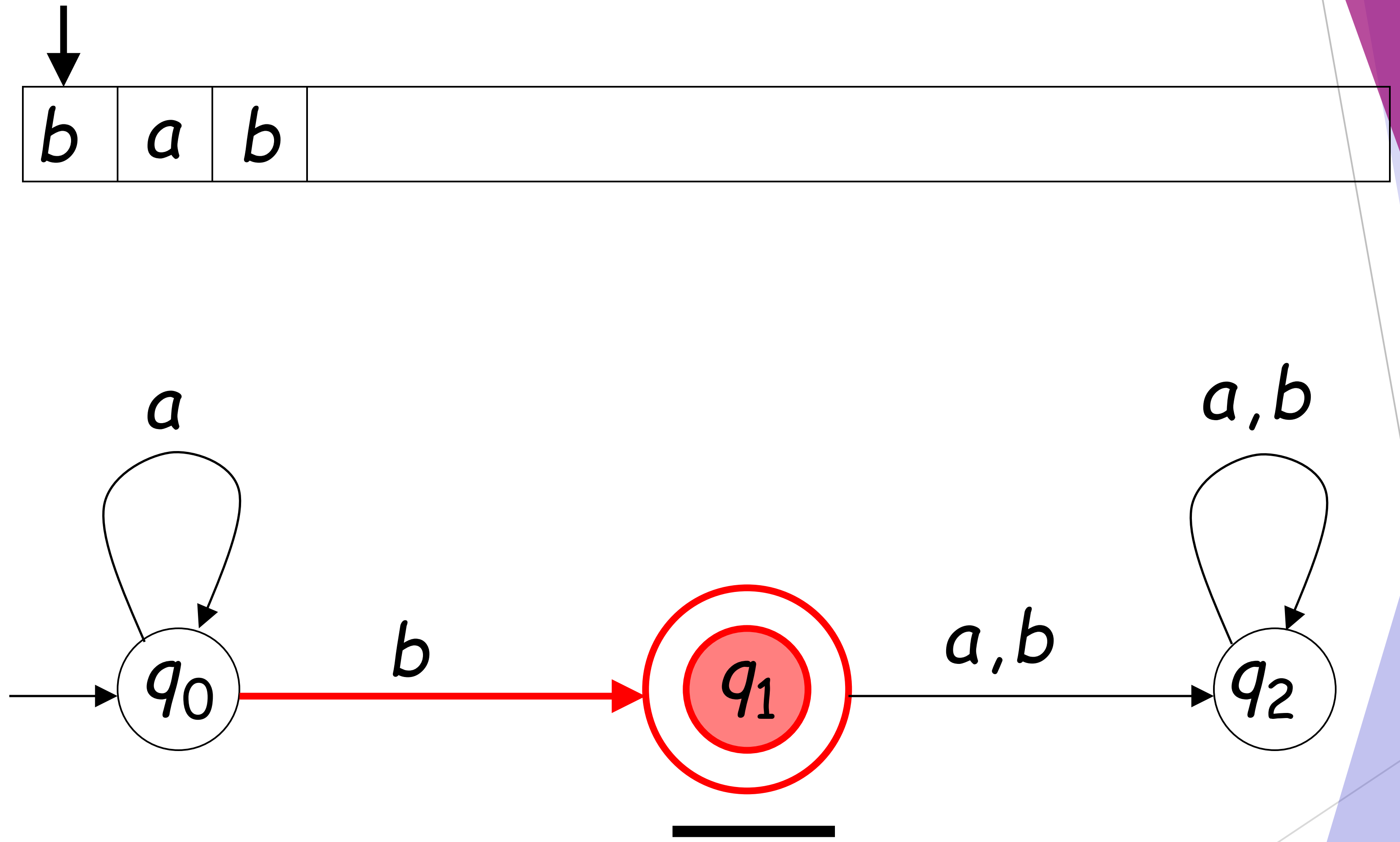



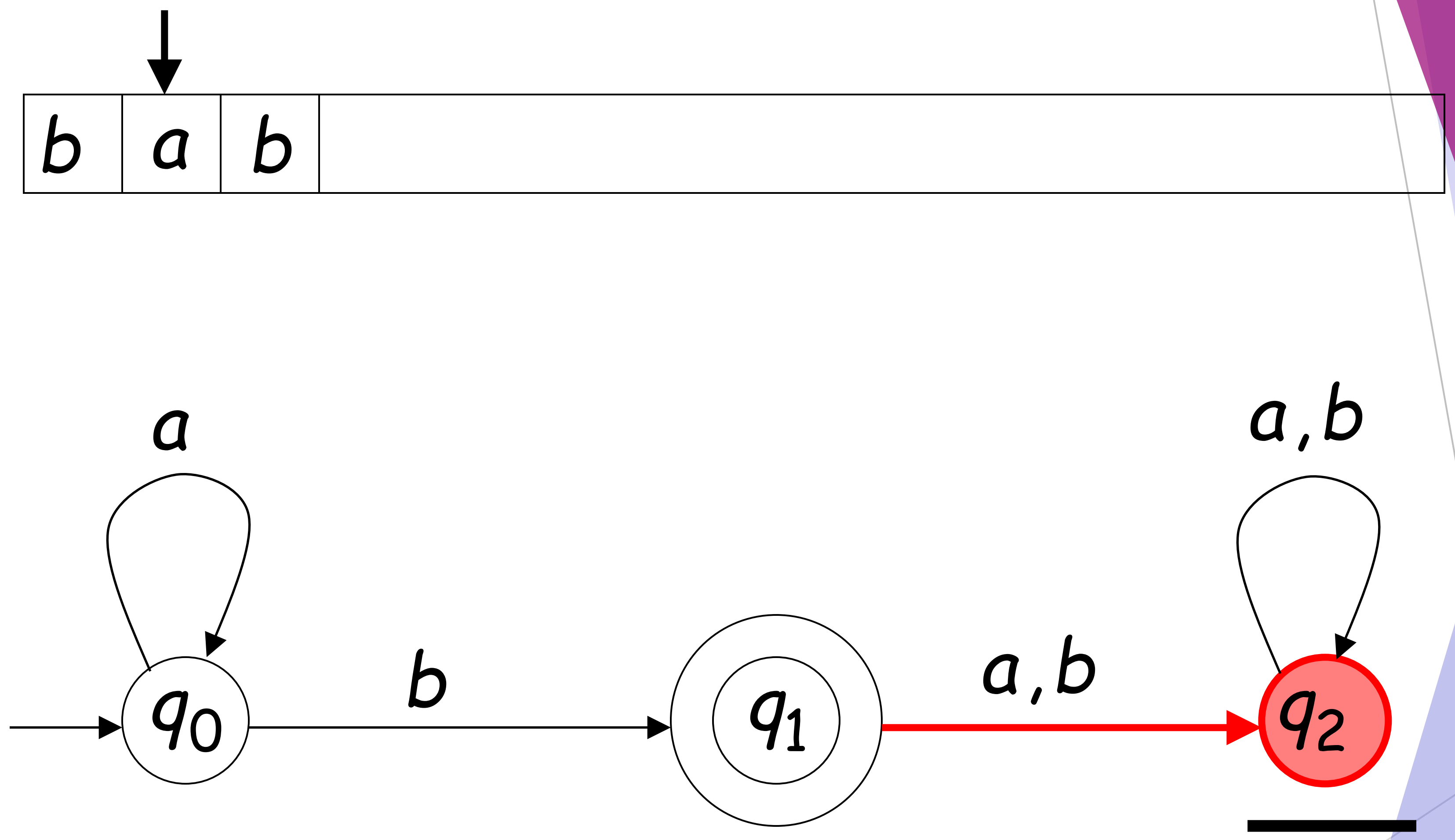

a
a
b
a
b
a,b
a,b
$q_0$
$q_1$
$q_2$

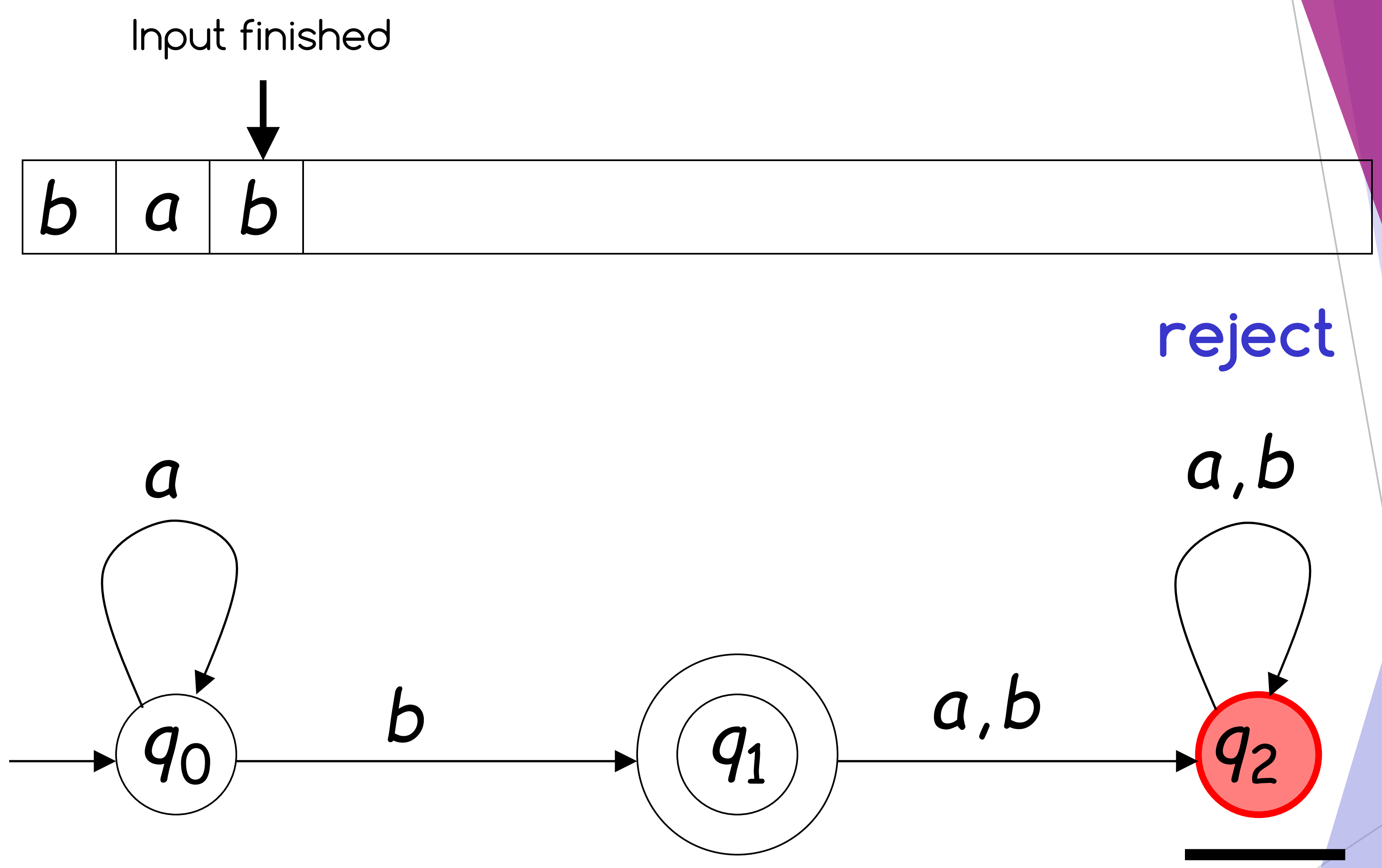

Input finished
a
a
b
a
accept
a,b
b
$q_0$
$q_1$
a,b
$q_2$

# Example 2



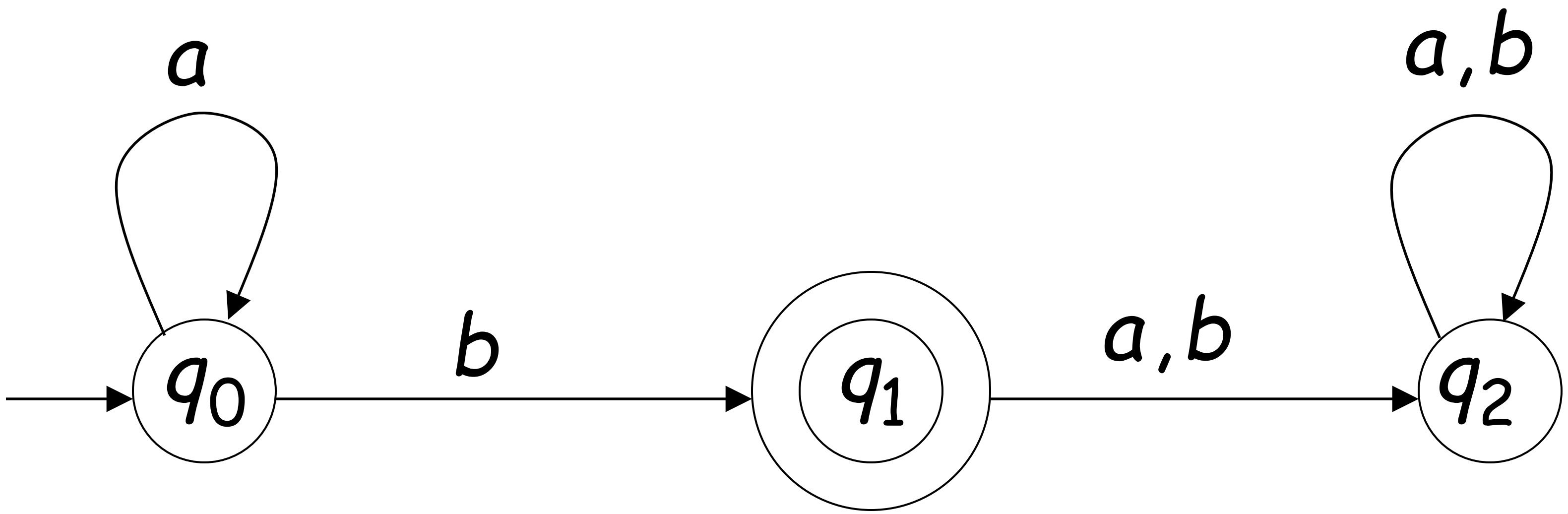

b
a
b
a
a,b
b
$q_0$
$q_1$
a,b
$q_2$


b
a
b
a
a,b
b
a,b
$q_0$
$q_1$
$q_2$


Input finished
b
a
b
reject
a
a,b
b
a,b
$q_0$
$q_1$
$q_2$

# What is the language accepted by this DFA?

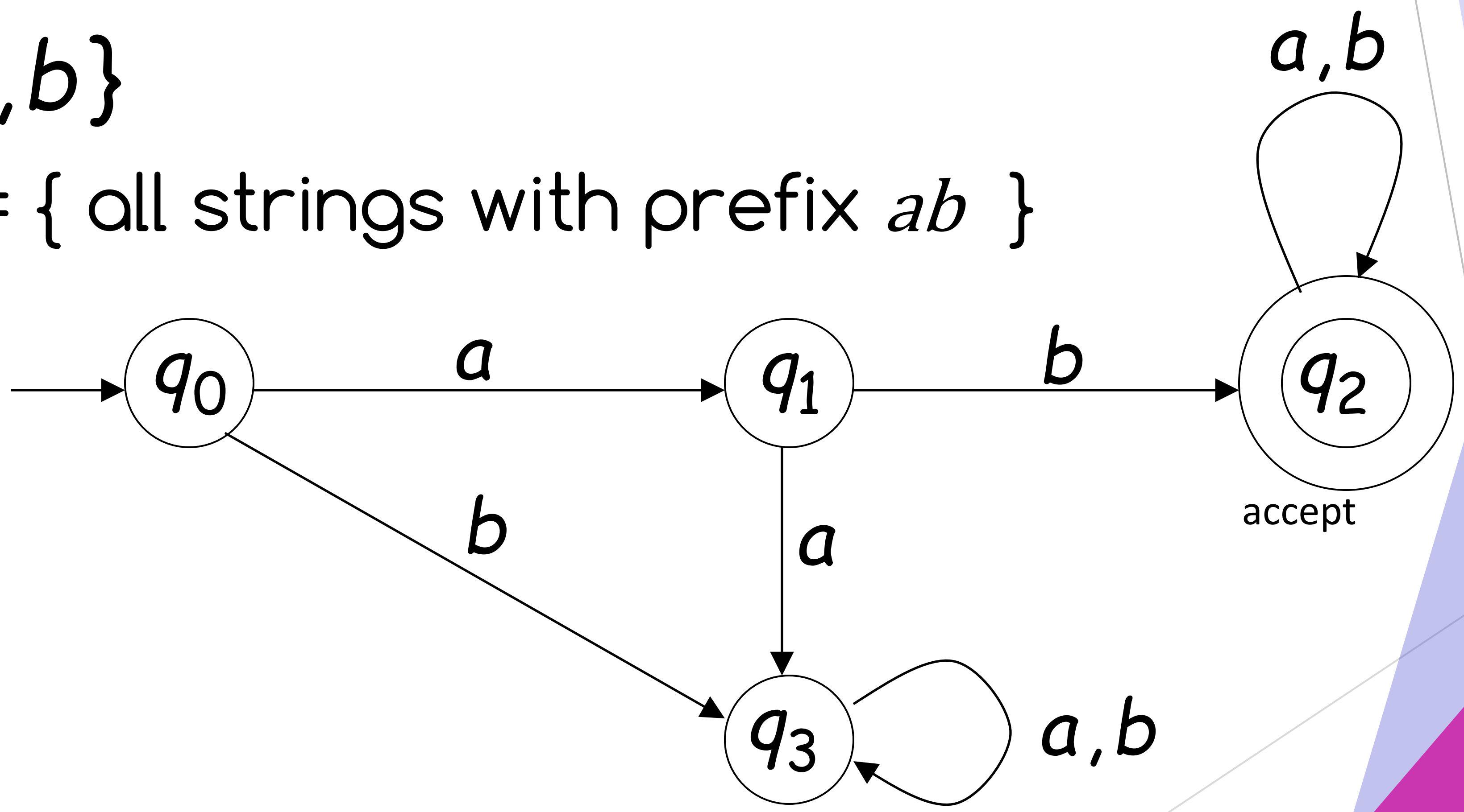
$$L = \{a^n b : n \geq 0\}$$



# What is the language accepted by this DFA?

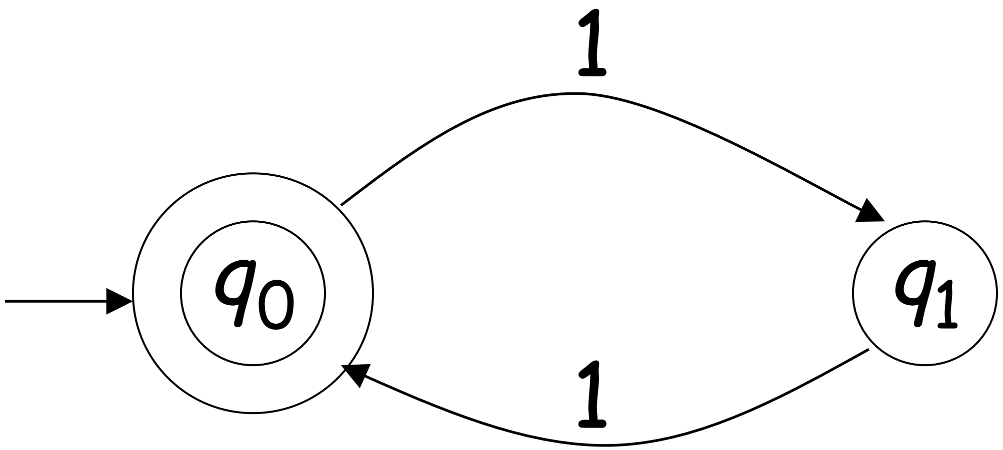
$$\Sigma = \{a, b\}$$

$L(M)$ = { all strings with prefix $ab$ }



# Design a DFA that accepts strings with even number of 1's

Alphabet: $\Sigma = \{1\}$

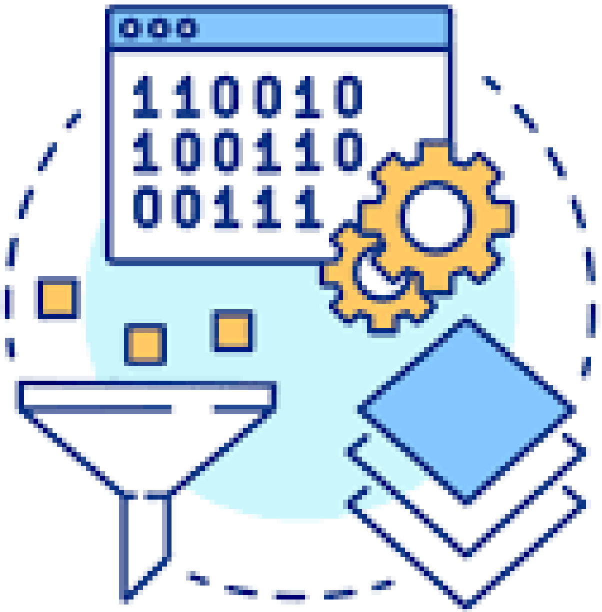



Language Accepted:

$$EVEN = \{x : x \in \Sigma^* \text{ and } x \text{ is even}\}$$

$$= \{\lambda, 11, 1111, 111111, \ldots\}$$


Q
A




# Nondeterministic Finite Automata (NFA)

# What is Determinism?

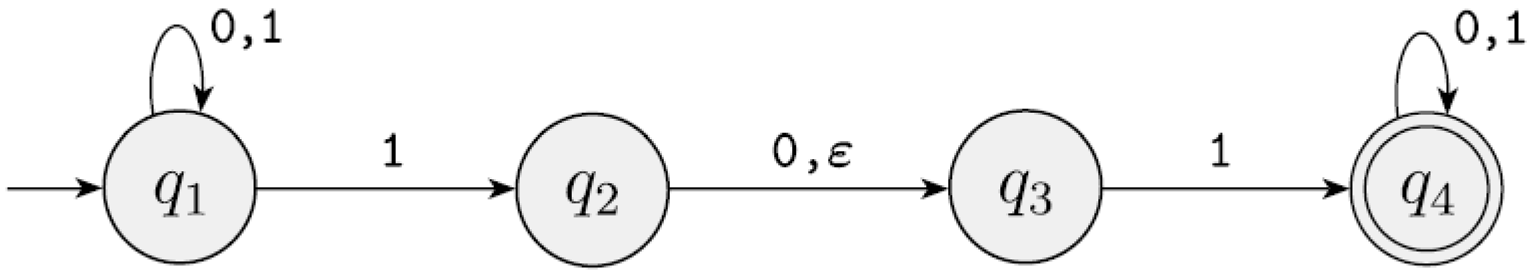
- Are computers deterministic?
- "If the current state is known, and the current inputs are known, then the future state is also known."
  - No Choices
  - No Randomness
  - No Oracles
  - No Errors/Cheating

# What is Nondeterminism?

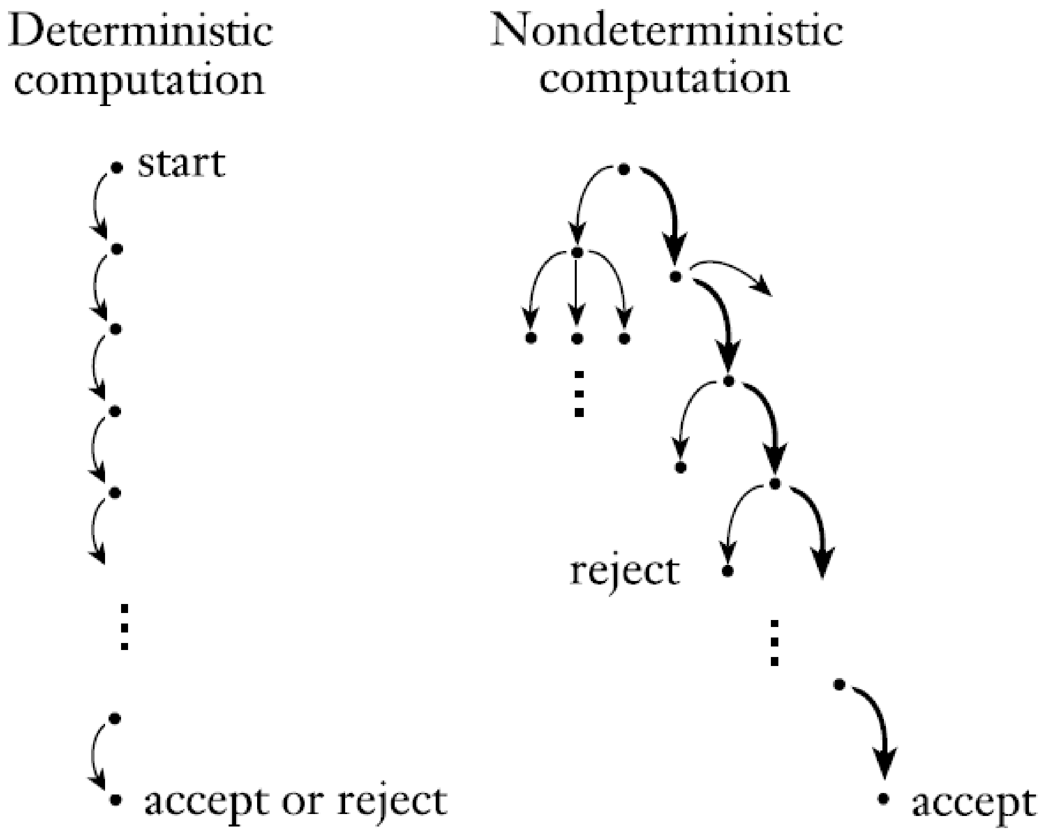
- "If the current state is known, and the current inputs are known, there may be **multiple** possible future states."
  - Random choice?
  - Parallel choice and simultaneous execution?

# Relaxing The Requirements

- Multiple edges with the same label ε (optional) edges
- Only need one path to an accept state
- How do we know which path to try?
  - Try them all
  - Always make the right choice



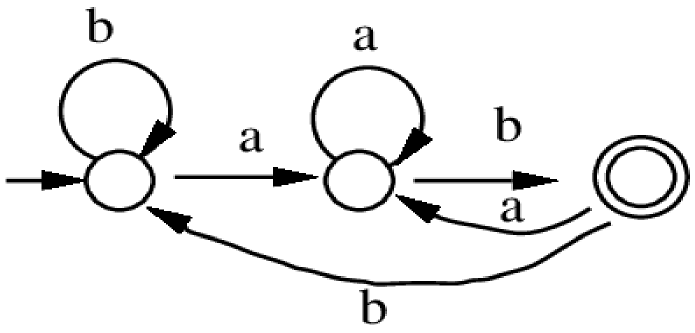
# Deterministic and nondeterministic computations with an accepting branch



# Example

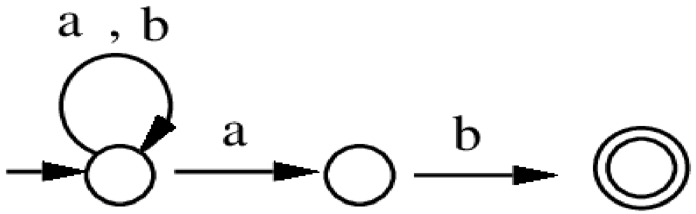
NFA is easier to construct but harder to implement compared to DFA

- L(M) accepts any string that ends with "ab"




DFA

for $M[p]$

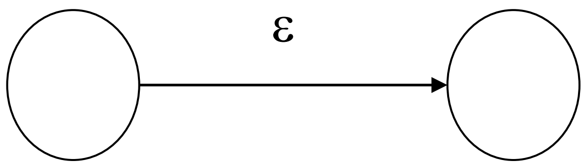



NFA

for $M[p]$

# ε-transition

- A transition that may occur without consulting the input string (and without consuming any characters)

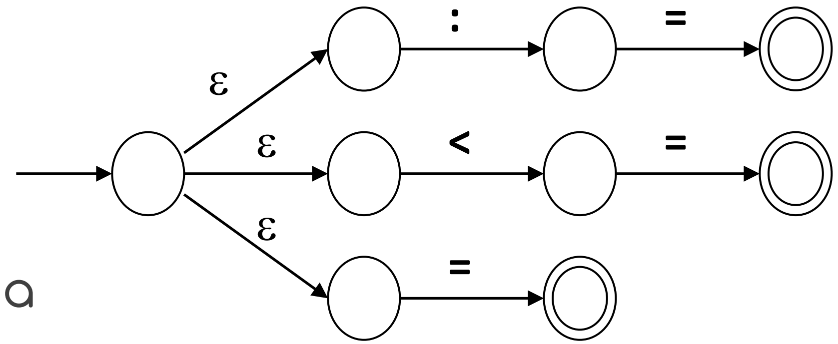



- It may be viewed as a "match" of the empty string.

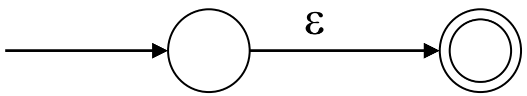
(This should not be confused with a match of the character ε in the input)

# ε-Transitions Used in Two Ways

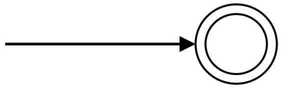
- **First:** to **express a choice of alternatives** in a way without combining states
  - Advantage: keeping the original automata intact and only adding a new start state to connect them




- **Second:** to explicitly **describe a match of the empty string**.

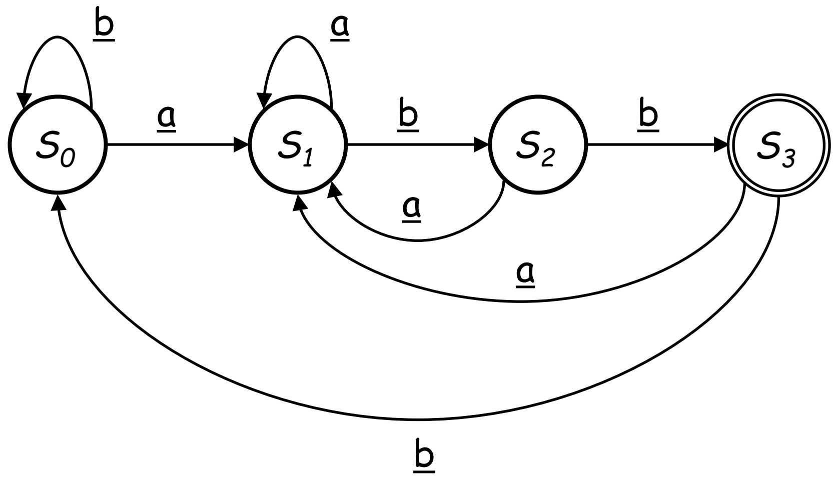



- Equivalent to the following 1-state **DFA**



# Non-deterministic Finite Automata

What about a RE such as $(\underline{a} \mid \underline{b})^* \underline{abb}$ ?

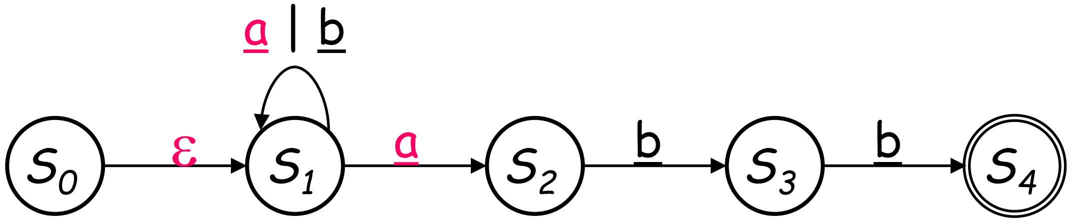



Each RE corresponds to a *deterministic finite automaton* (DFA)

- We know a DFA exists for each RE
- The DFA may be hard to build directly

# Non-deterministic Finite Automata

Here is a simpler presentation for ( a | b )* abb




This recognizer is more intuitive

- Structure seems to follow the RE's structure

This recognizer is not a DFA

- $S_0$ has a transition on ε
- $S_1$ has two transitions on a

This is a *non-deterministic finite automaton* (NFA)

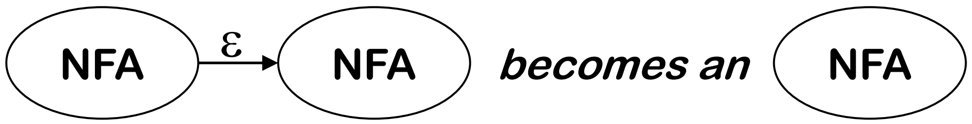
# Non-deterministic Finite Automata

An NFA accepts a string $x$ *iff* $\exists$ a path through the transition graph from $s_0$ to a final state such that the edge labels spell $x$, ignoring $\varepsilon$'s

- TRANSITIONS ON $\varepsilon$ CONSUME NO INPUT
- To "run" the NFA, start in $s_0$ and *guess* the right transition at each step
  - Always guess correctly
  - If some sequence of correct guesses accepts x then accept

## Why study NFAs?

- They are the key to automating the RE $\rightarrow$ DFA construction
- We can paste together NFAs with $\varepsilon$-transitions



# Some Notes

An NFA does not represent an algorithm.

- However, it can be simulated by an algorithm that backtracks through every non-deterministic choice.

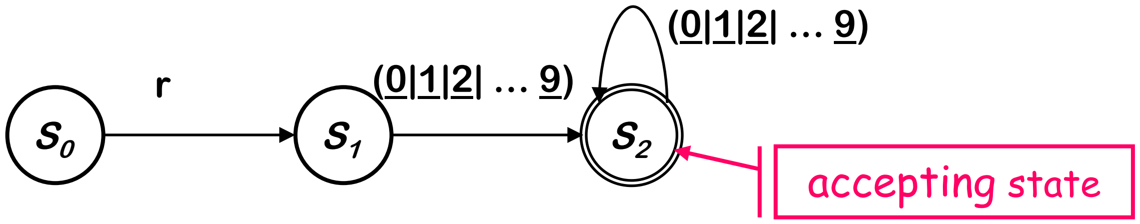
# Example 1

Consider the problem of recognizing ILOC register names

*Register* → r (0|1|2| … | 9) (0|1|2| … | 9)$^*$

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



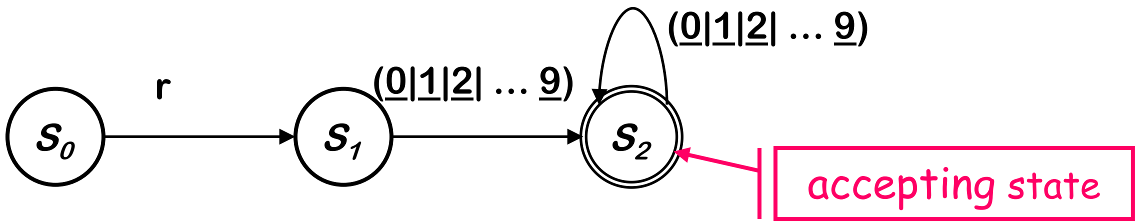

**Recognizer for *Register***

*Transitions on other inputs go to an error state, $s_e$*

# Example 1

DFA operation

- Start in state $S_0$ & make transitions on each input character
- DFA accepts a word $\underline{x}$ *iff* $\underline{x}$ leaves it in a final state ($S_2$)



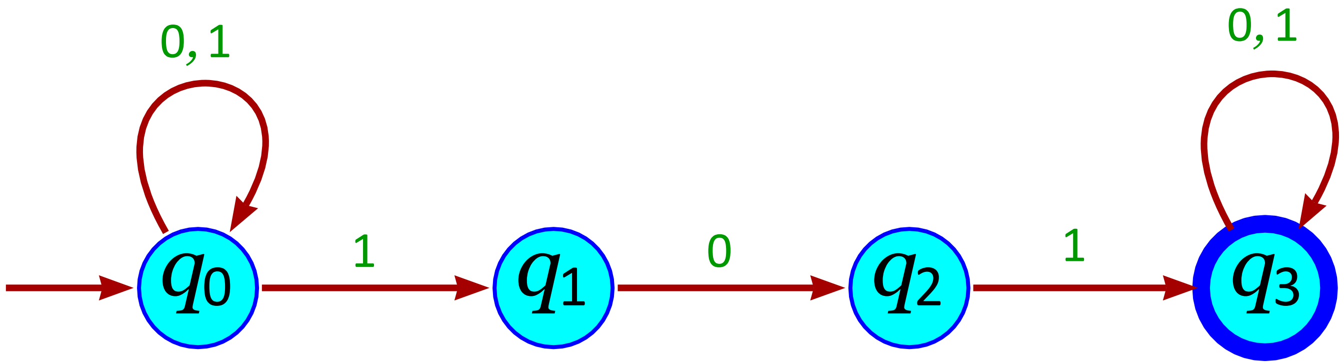

**Recognizer for *Register***

So,

- r17 takes it through $s_0$, $s_1$, $s_2$ and accepts
- r takes it through $s_0$, $s_1$ and fails
- a takes it straight to $s_e$

# Example 2

- Design an NFA that recognizes the language $L = \{w | w \in \{0, 1\}^* \text{ and } w \text{ has } 101 \text{ as a substring } \}$.

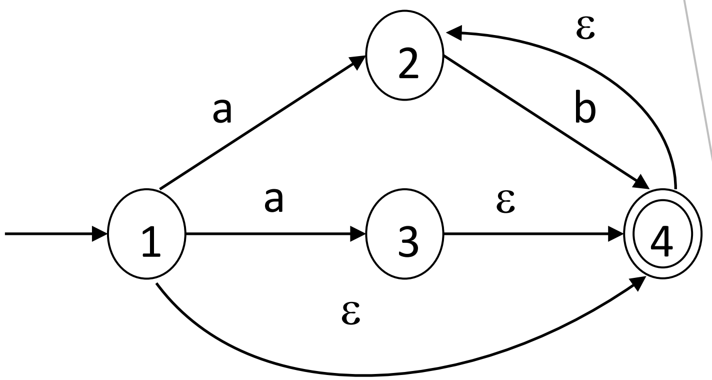


# Example 3

- The string **abb** can be accepted by either of the following sequences of transitions:

$$\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

$$\rightarrow 1 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

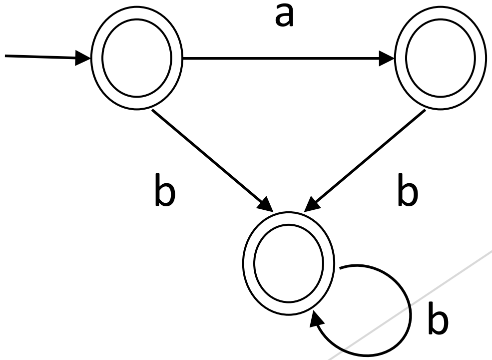



- This NFA accepts the languages as follows:

  regular expression: (a|ε) b*

  ab+|ab*|b*

- This DFA accepts the same language.



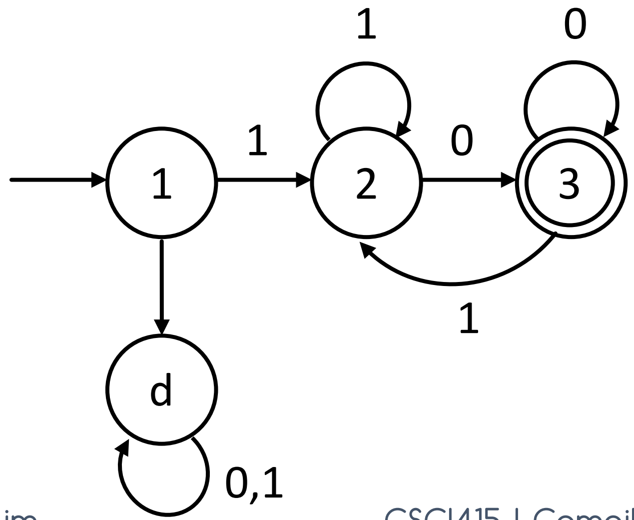
# Example 4

L = {w | w begins with a *1* and ends with a *0*}

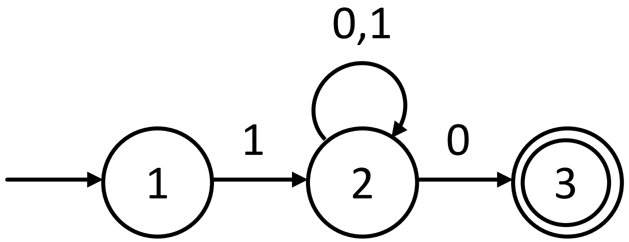
## Regex

$1\ \Sigma^*\ 0$

### DFA

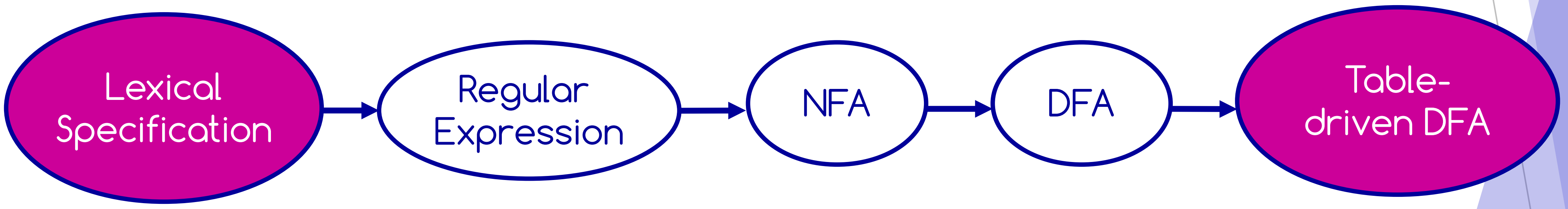



### NFA



# Where are we going?

The steps of implementing the lexical analyzer



# Where are we going?

- Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
  - Easy to build in an algorithmic way
  - Requires ε-transitions to combine regular subexpressions
- Construct a deterministic finite automaton (DFA) to simulate the NFA
  - Use a set-of-states construction
- Minimize the number of states in the DFA

Optional, but worthwhile

  - Hopcroft state minimization algorithm
- Generate the scanner code
  - Additional specifications needed for the actions

# Automating Scanner Construction

To convert a specification into code:

1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code


Q
A

# Relationship between NFAs and DFAs

DFA is a special case of an NFA

- DFA has no $\varepsilon$ transitions
- DFA's transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

  - *Obviously*

NFA can be simulated with a DFA *(less obvious)*

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream

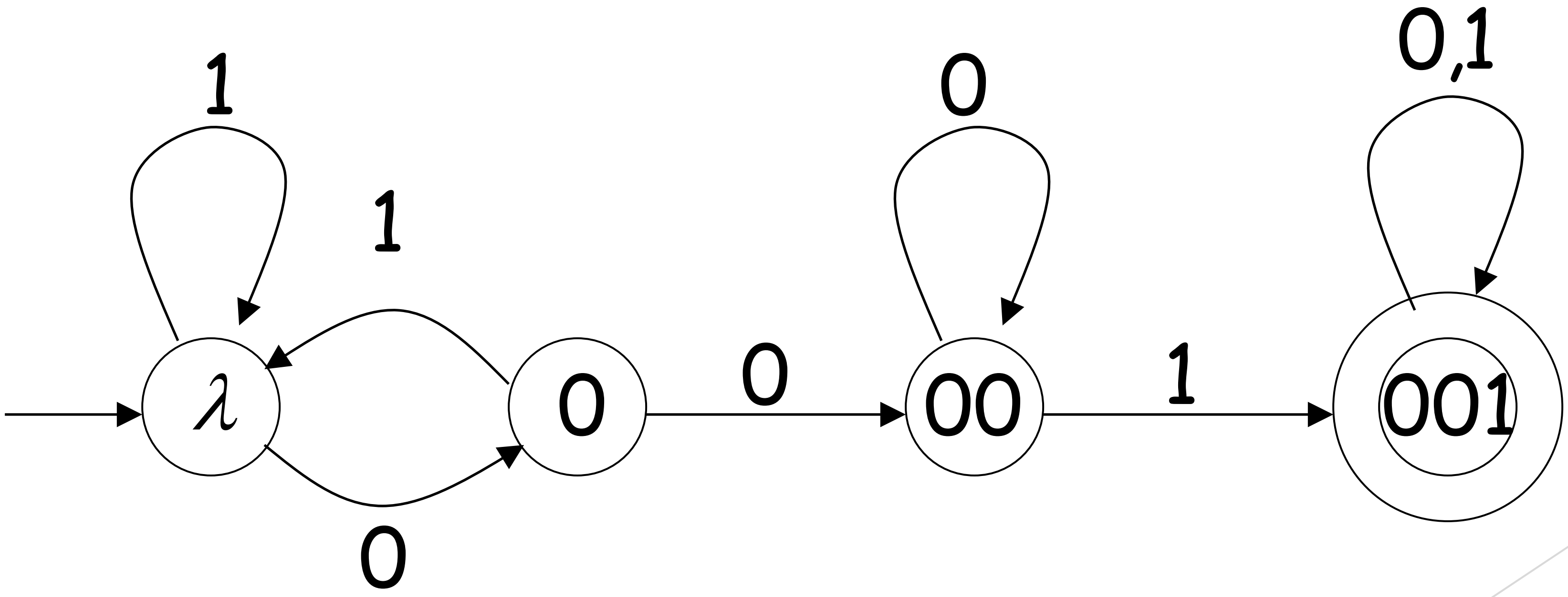
Rabin & Scott, 1959


# Review Questions

# Problem 1

## Define the language of the given DFA

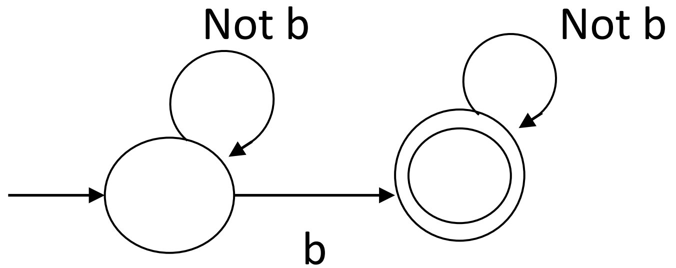
$L(M)$ = { all binary strings containing substring 001 }



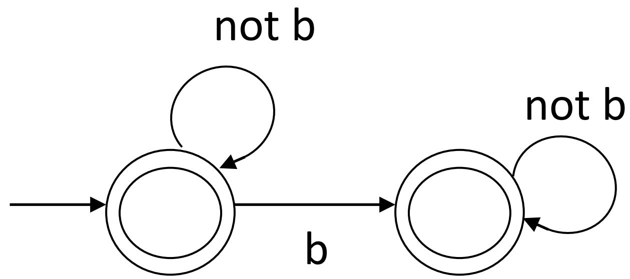
# Problem 2

## Design the DFA for ∑ = {a, b, c, …, z} that

1. Accepts strings containing exactly one b




2. Accepts strings containing at most one b



# Problem 3

Design the DFA of a number

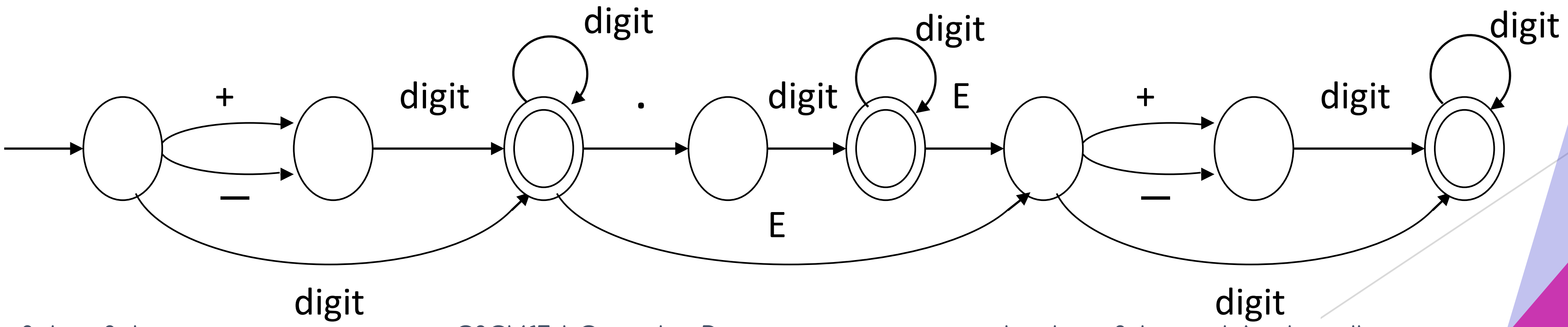
digit = [0-9]

nat = digit+

signedNat = (+|-)? nat

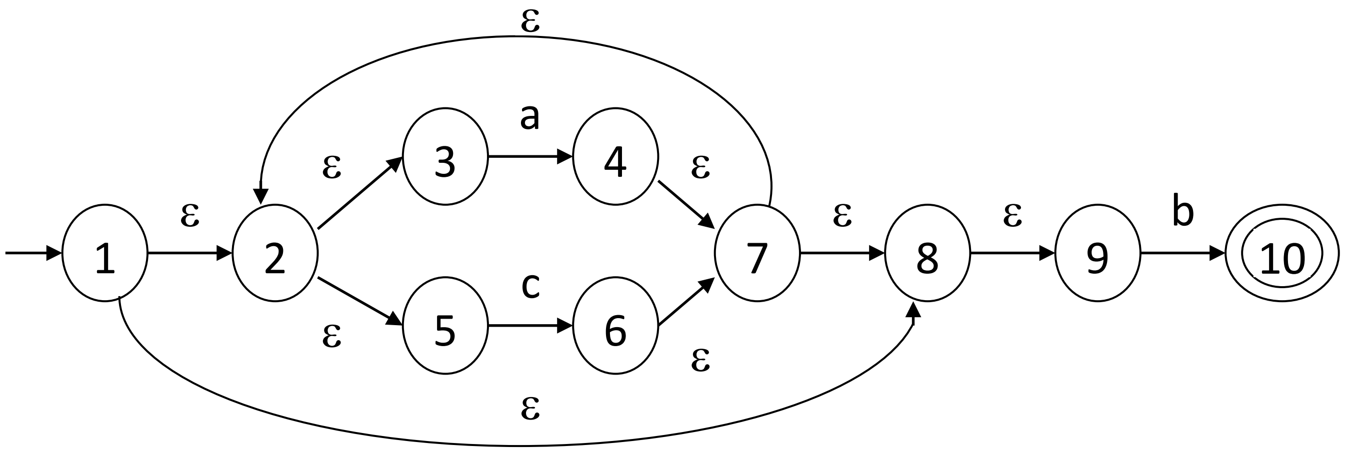
Number = singedNat(“.”nat)?(E signedNat)?

## A DFA of Number:

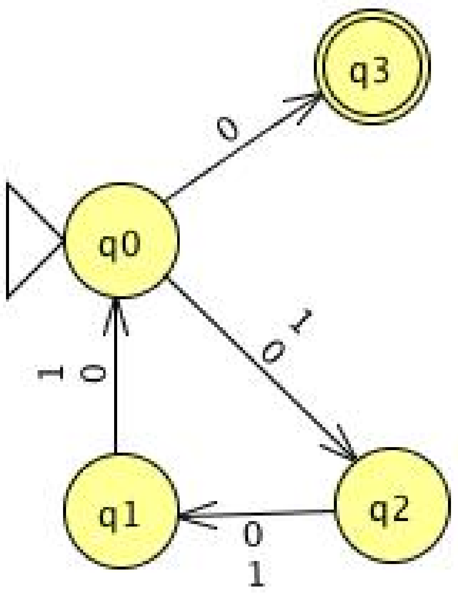


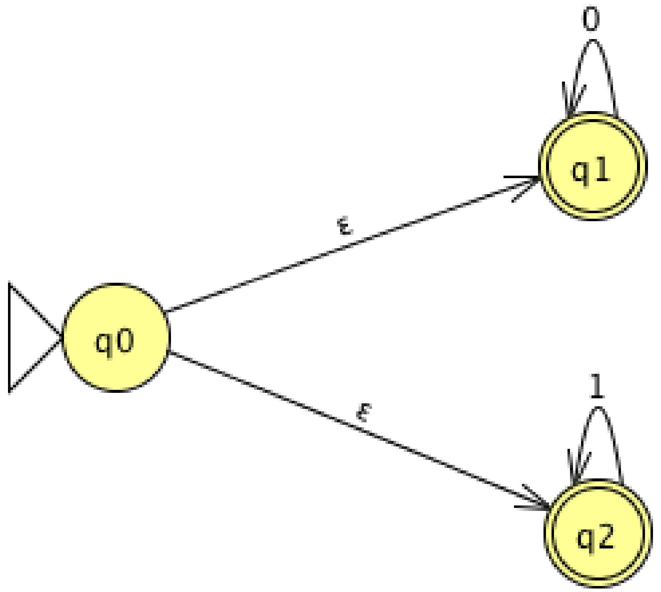
# Problem 4

- It accepts the string acab by making the following transitions:
  - (1)(2)(3)a(4)(7)(2)(5)(6)c(7)(2)(3)a(4)(7)(8)(9)b(10)
- It accepts the same language as that generated by the regular expression : (a | c)* b



# Problem 5: Accepted strings by NFA




Accepts strings that consists of one zero or multiples of 3 symbols and ends with 0

Accepts empty strings or strings that contains any number of zeros OR any number of ones

# References of this lecture

- Presentation slides of the book: COMPILER CONSTRUCTION, Principles and Practice, by Kenneth C. Louden
- Presentation slides of the book: Introduction to the Theory of Computation, Michael Sipser, 2nd edition
  - Prepared by: Ananth Kalyanaraman
- Credits for Dr. Sally Saad, Prof. Mostafa Aref, Dr. Islam Hegazy, and Dr. Abd ElAziz for help in content preparation and aggregation (FCIS-ASU)

# Next Lecture

- NFA to DFA
- DFA Minimization
- Transition Table

# See you next lecture