Finite Automaton

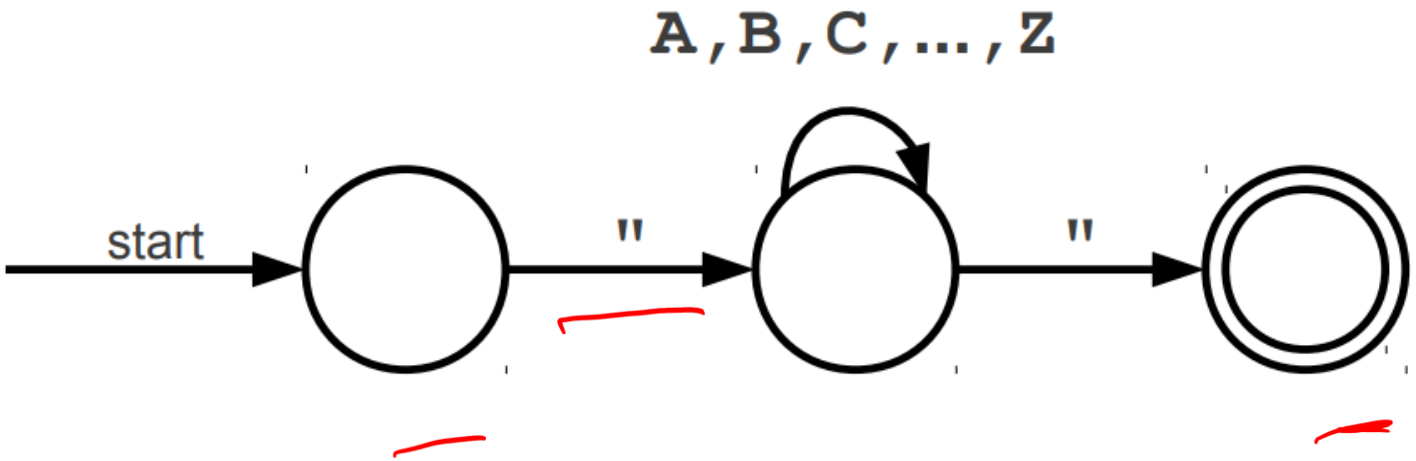
Professor: Suman Saha

Implementing Regular Expression

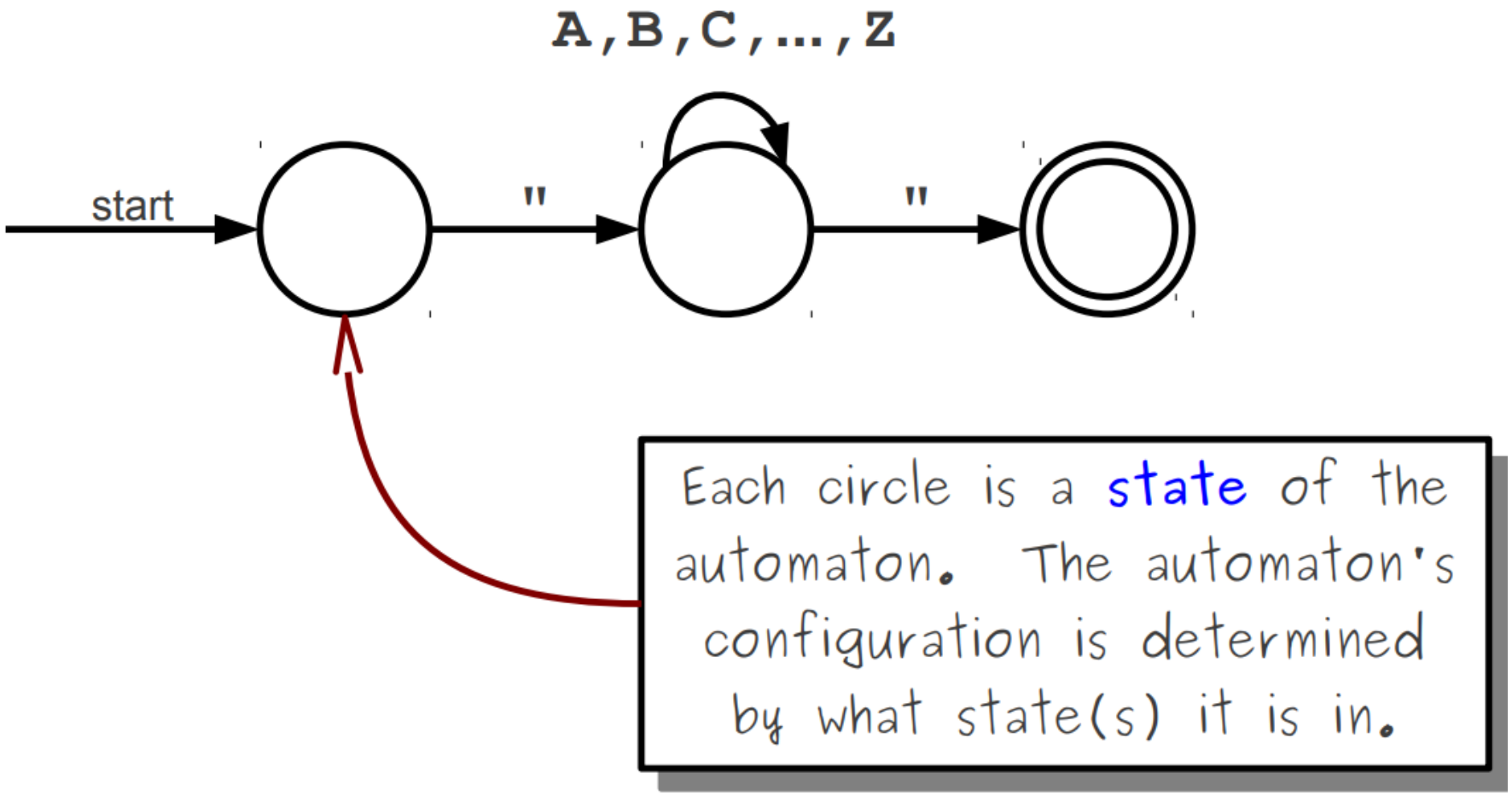
PennState

- Regular expressions are equivalent to **finite automata**
  - NFAs (nondeterministic finite automata)
  - DFAs (deterministic finite automata)
- Finite automata are easily turned into computer programs
- Two methods:
  - Convert the regular expressions to an NFA and simulate the NFA
  - Convert the regular expressions to an NFA, convert the NFA to a DFA, and simulate the DFA.

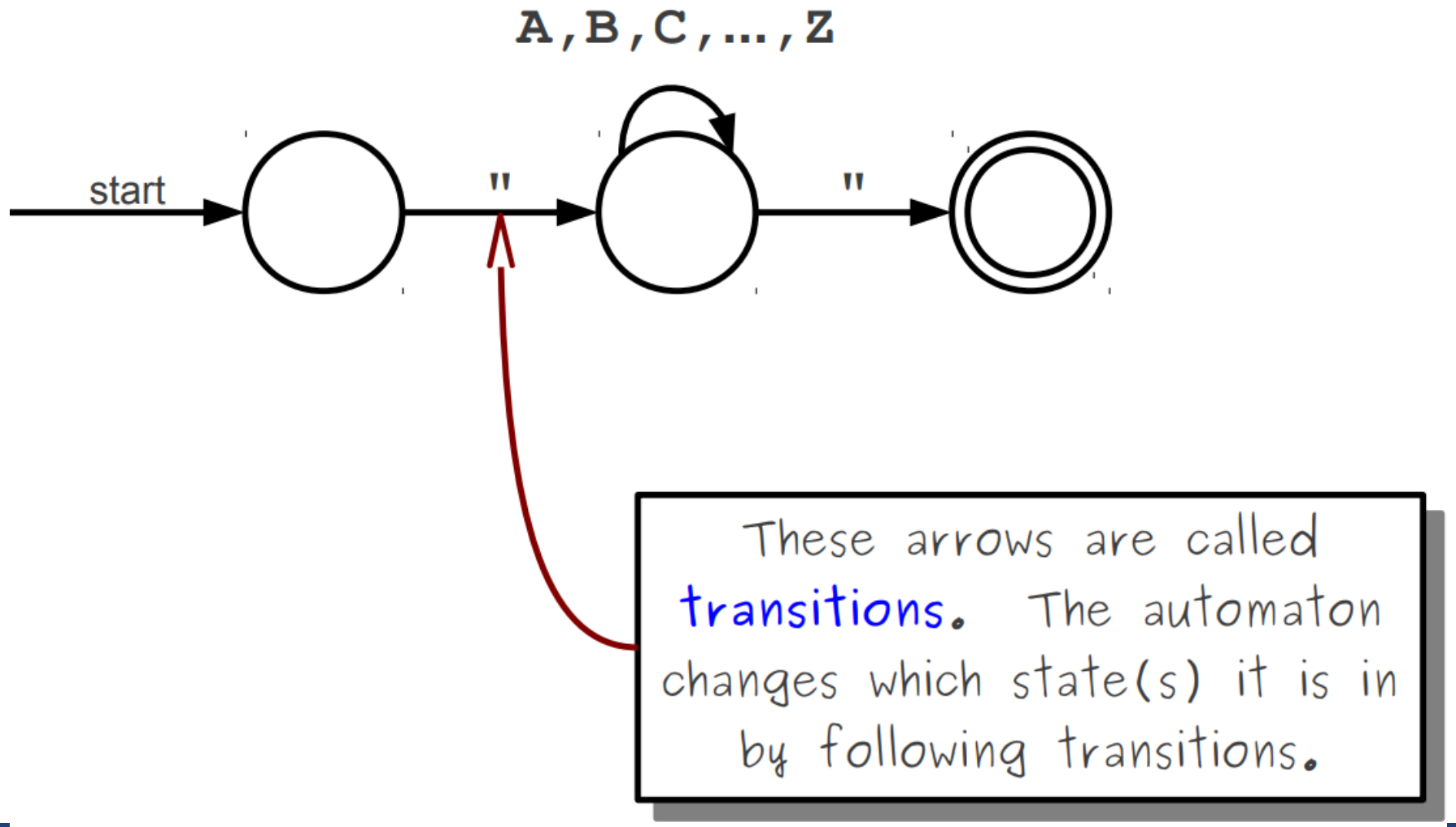
A Simple Automaton



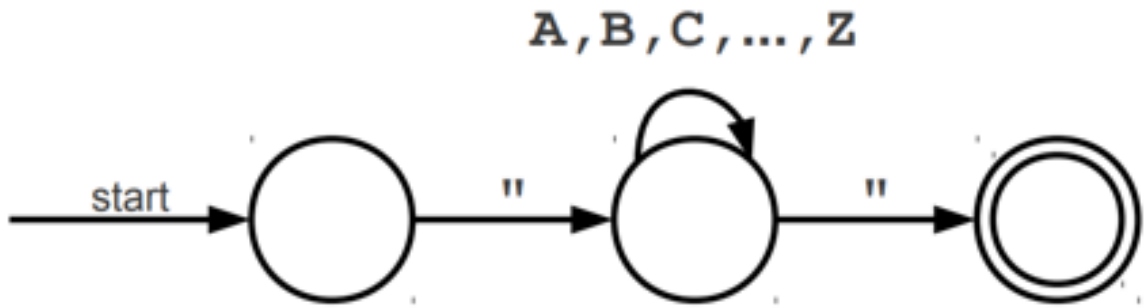
A Simple Automaton

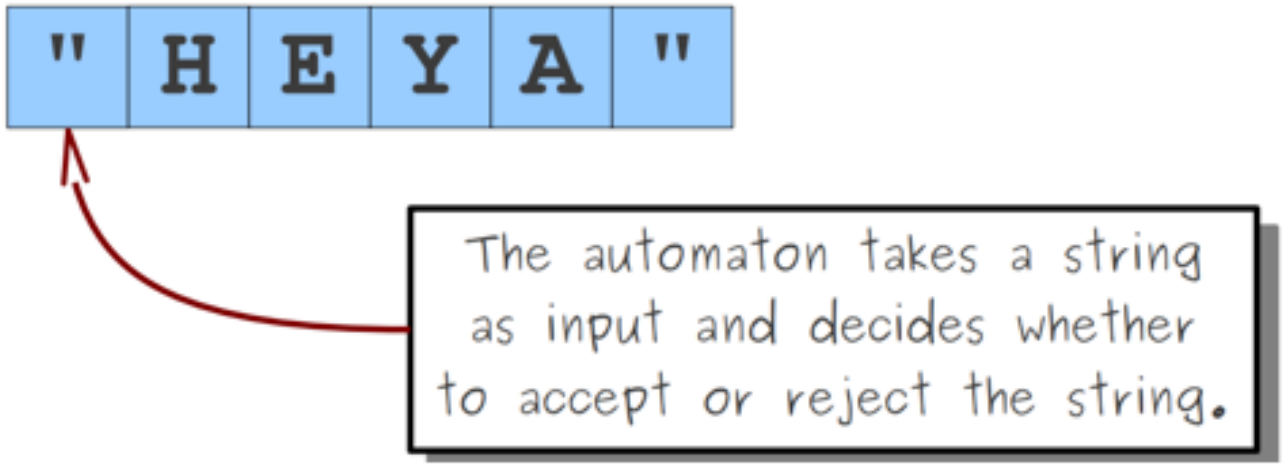


A Simple Automaton

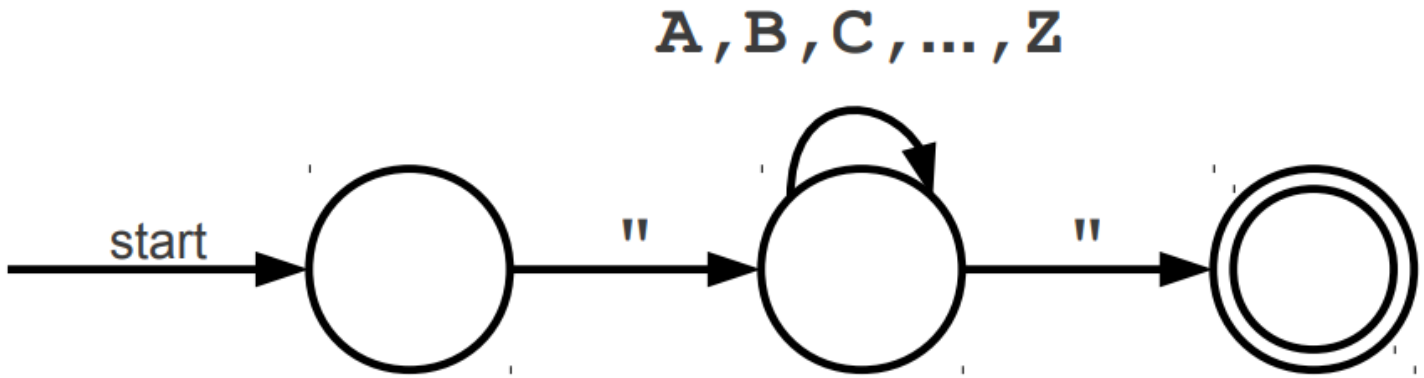


A Simple Automaton

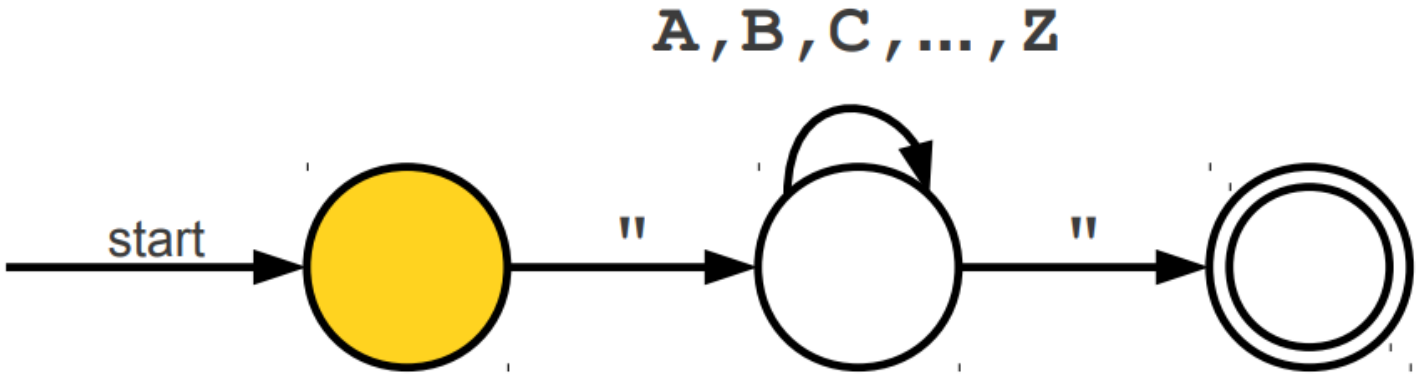




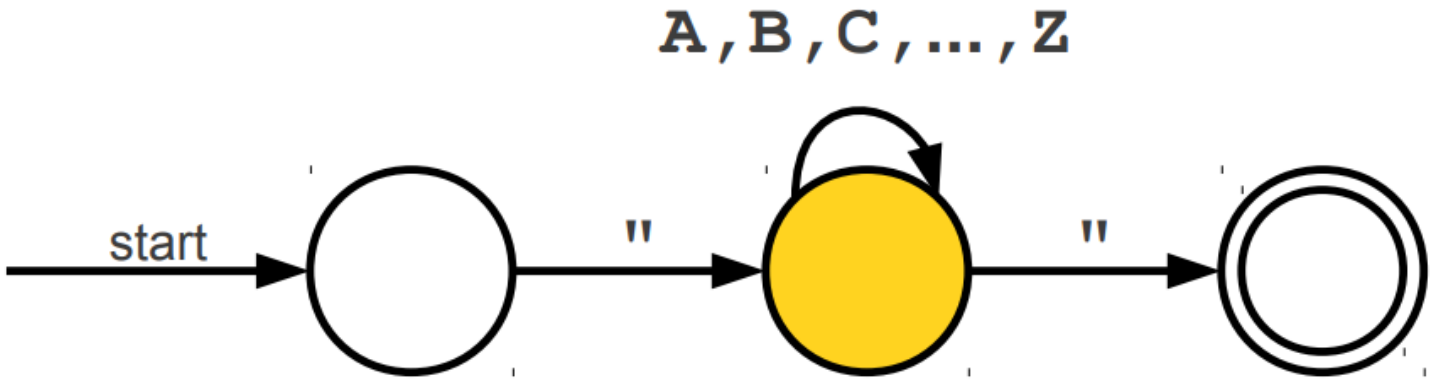
A Simple Automaton



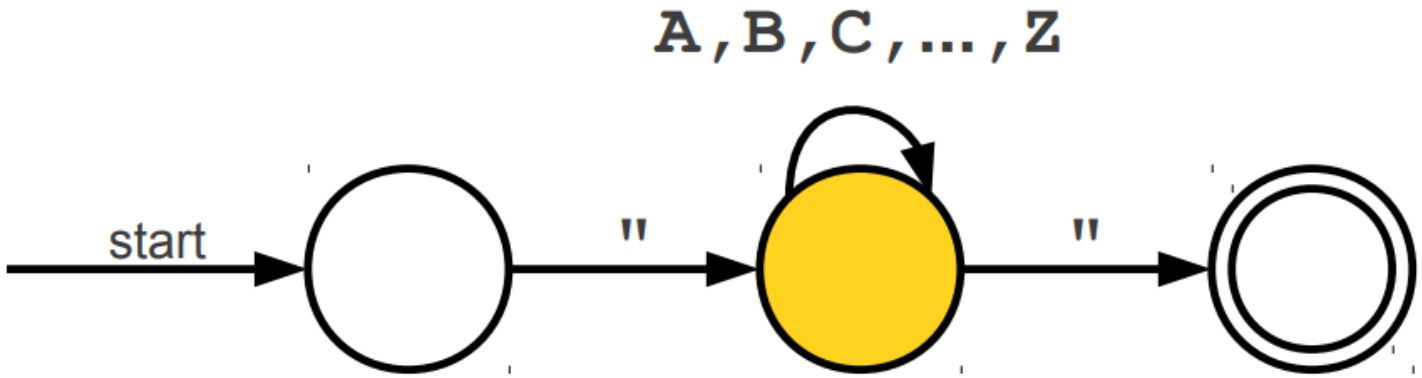
A Simple Automaton



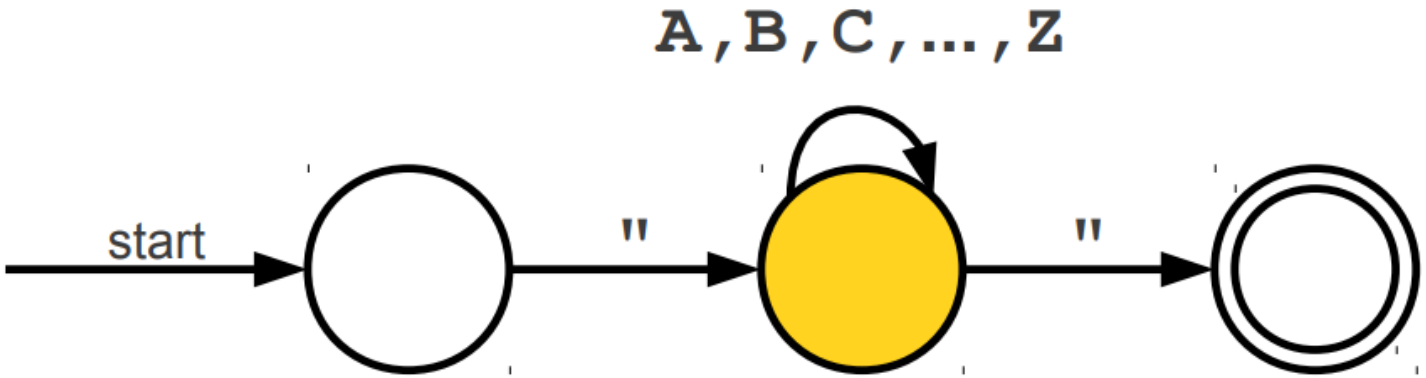
A Simple Automaton

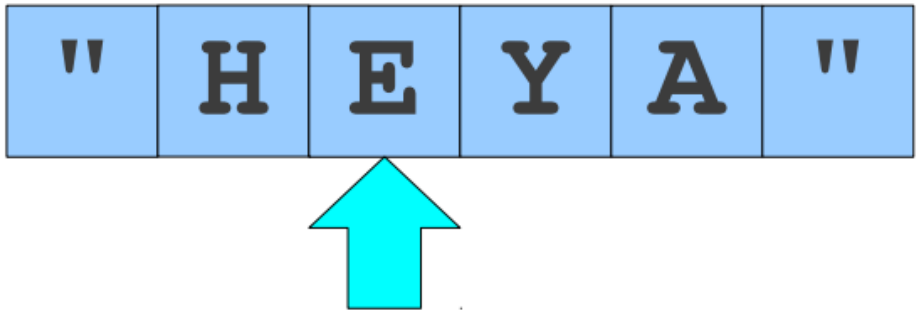


A Simple Automaton

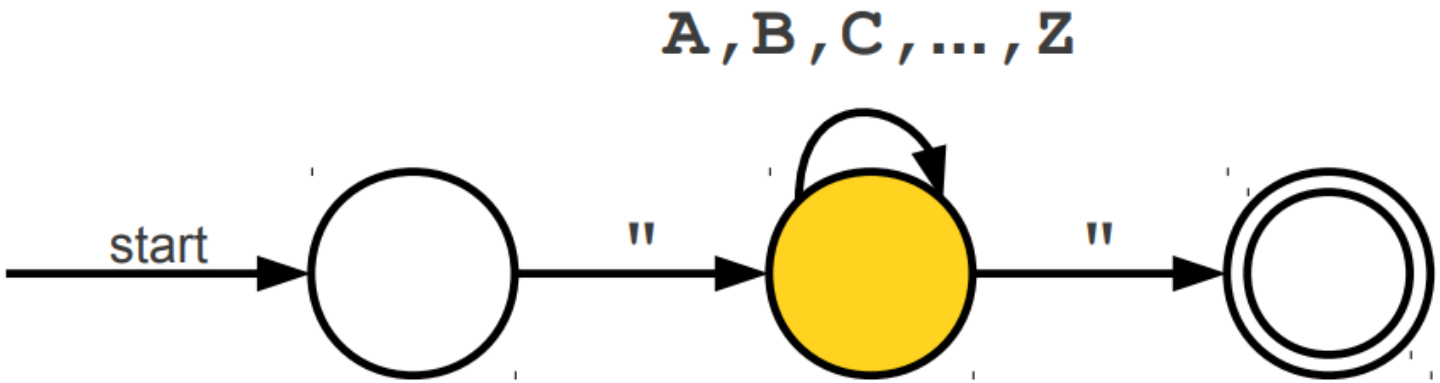


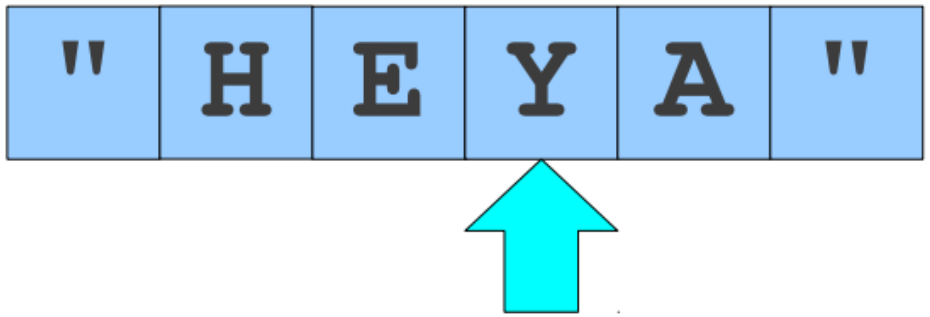
A Simple Automaton



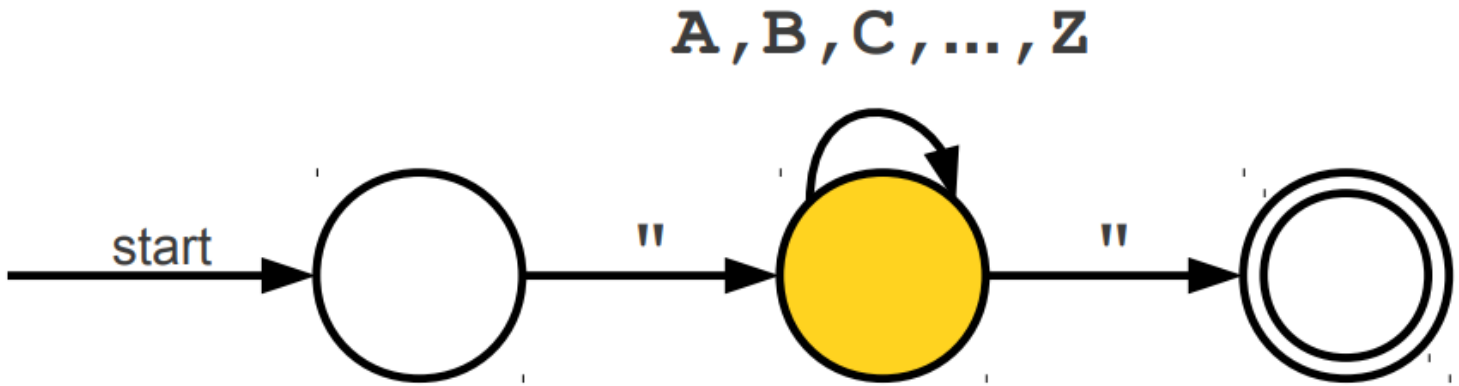


A Simple Automaton

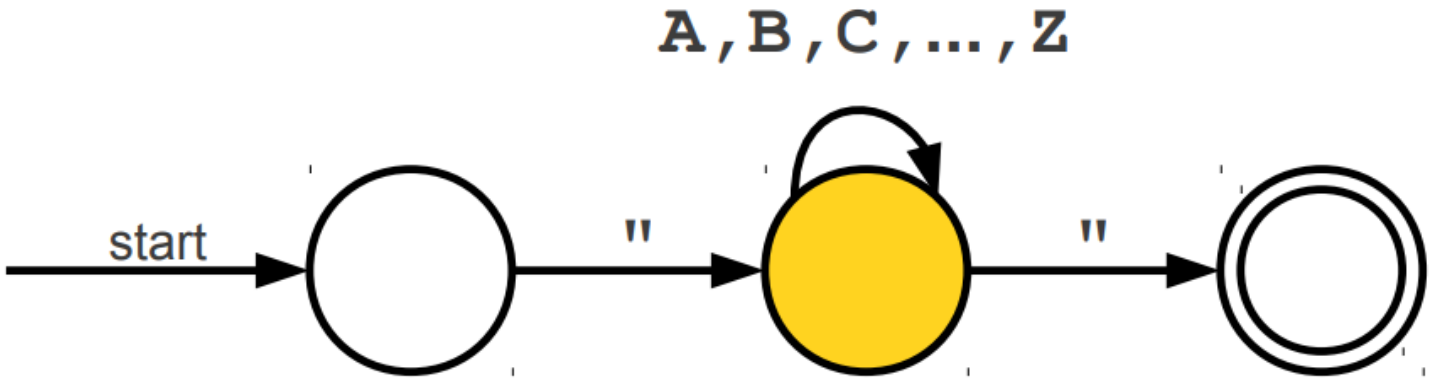




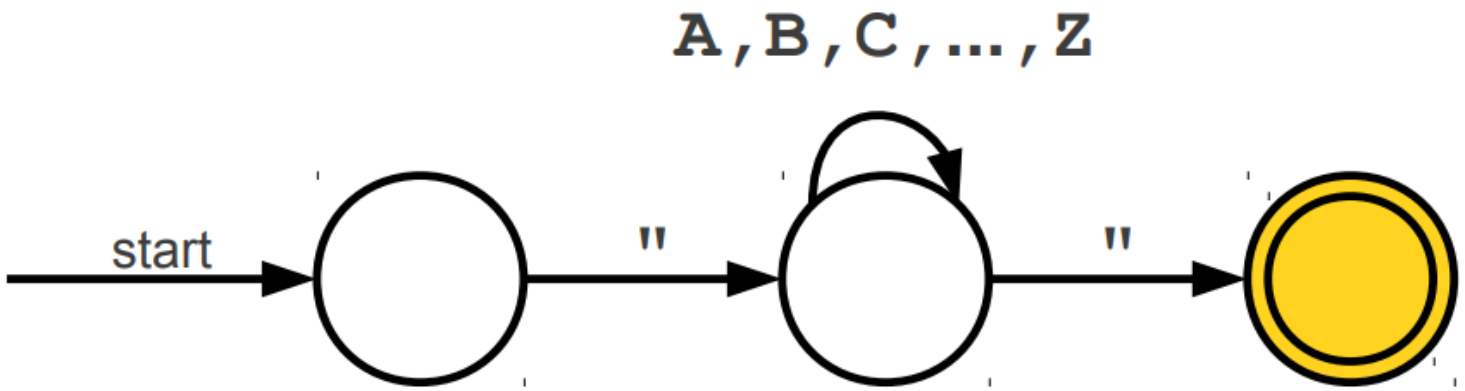
A Simple Automaton



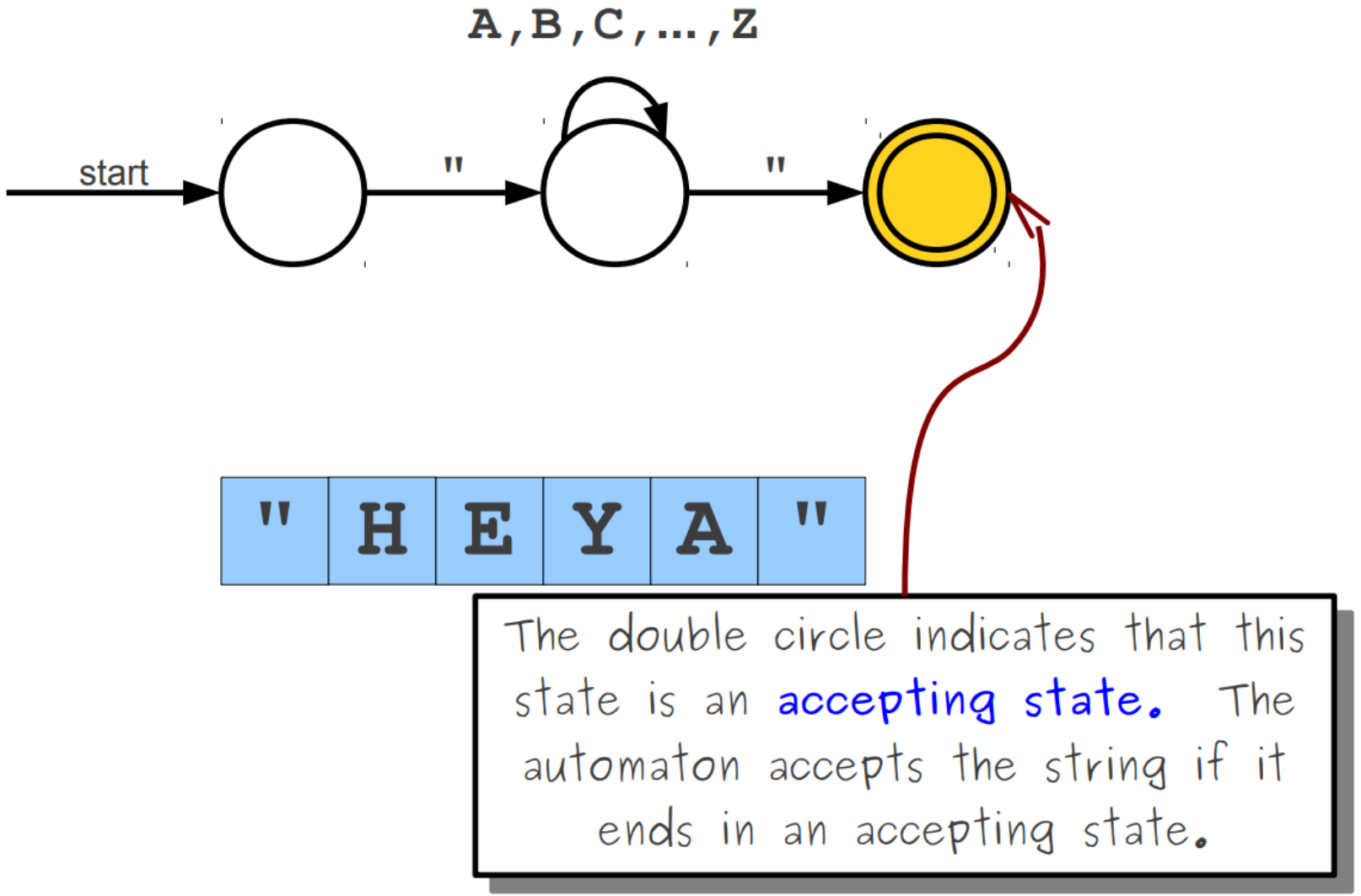
A Simple Automaton



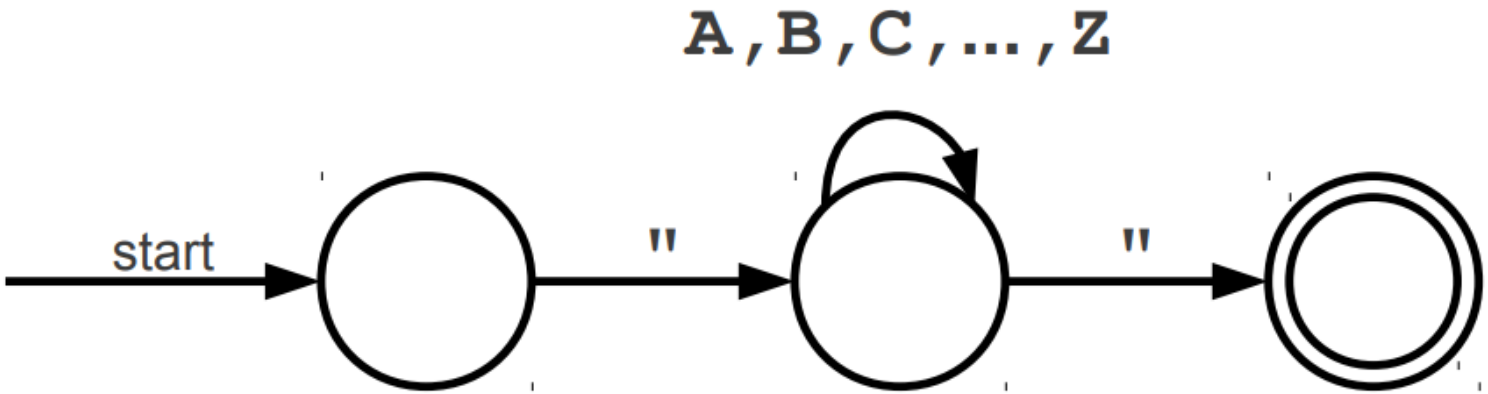
A Simple Automaton

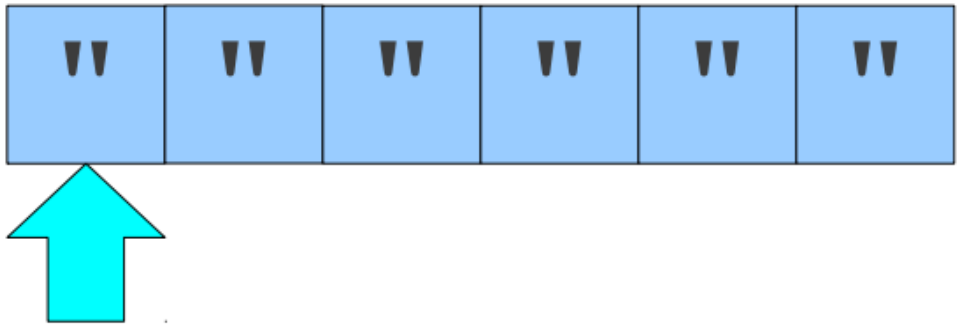


A Simple Automaton

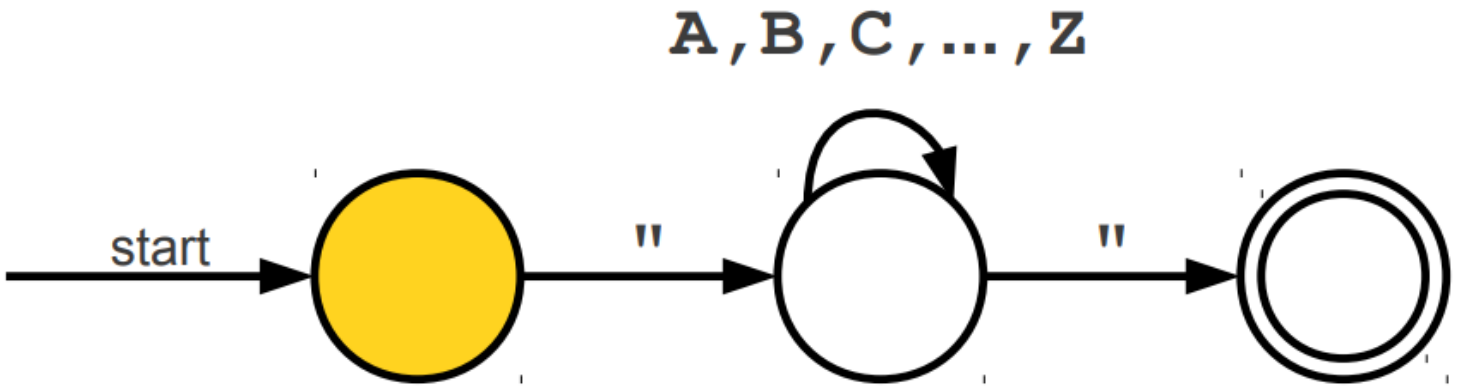


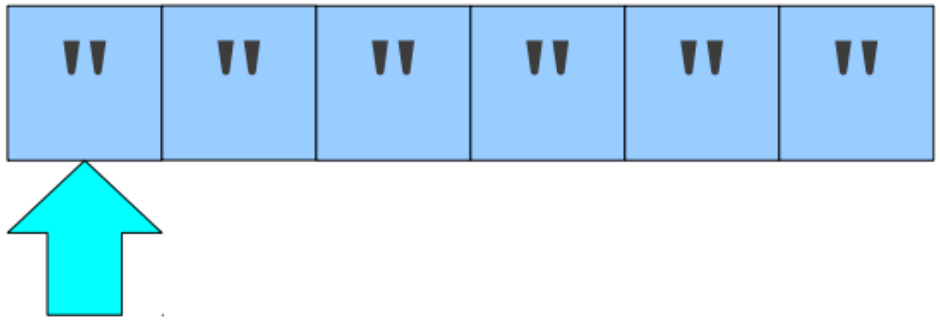
A Simple Automaton



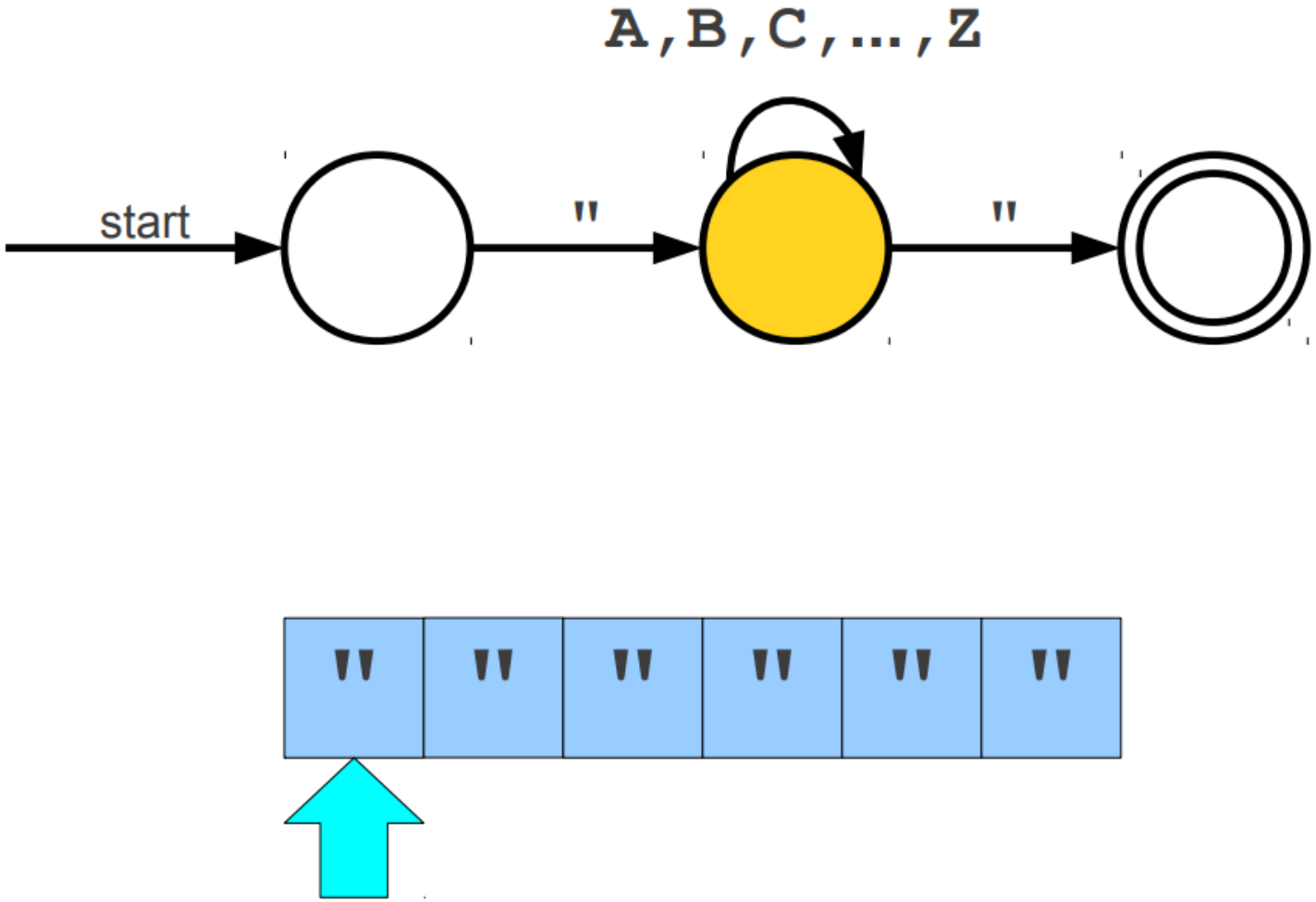


A Simple Automaton

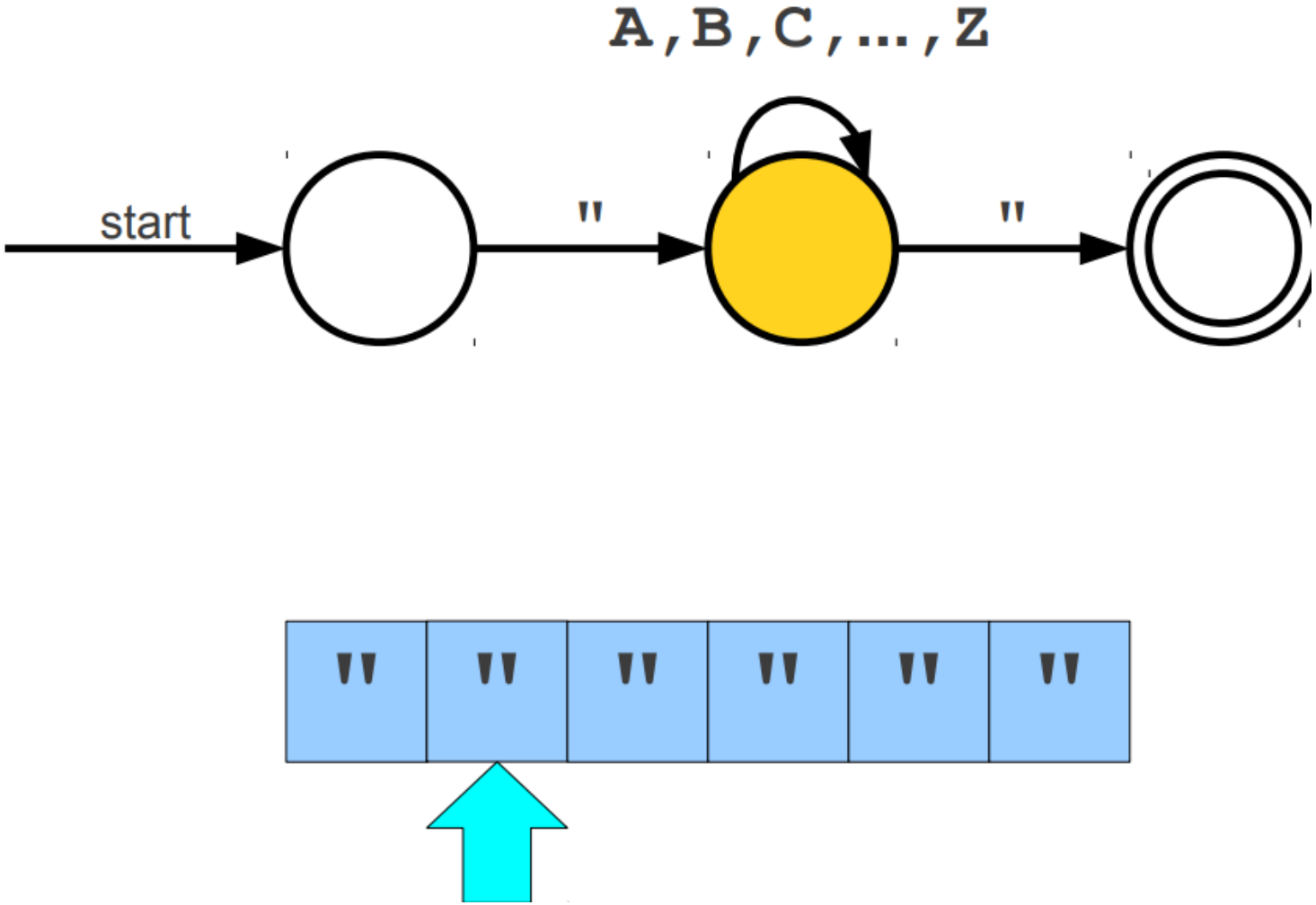




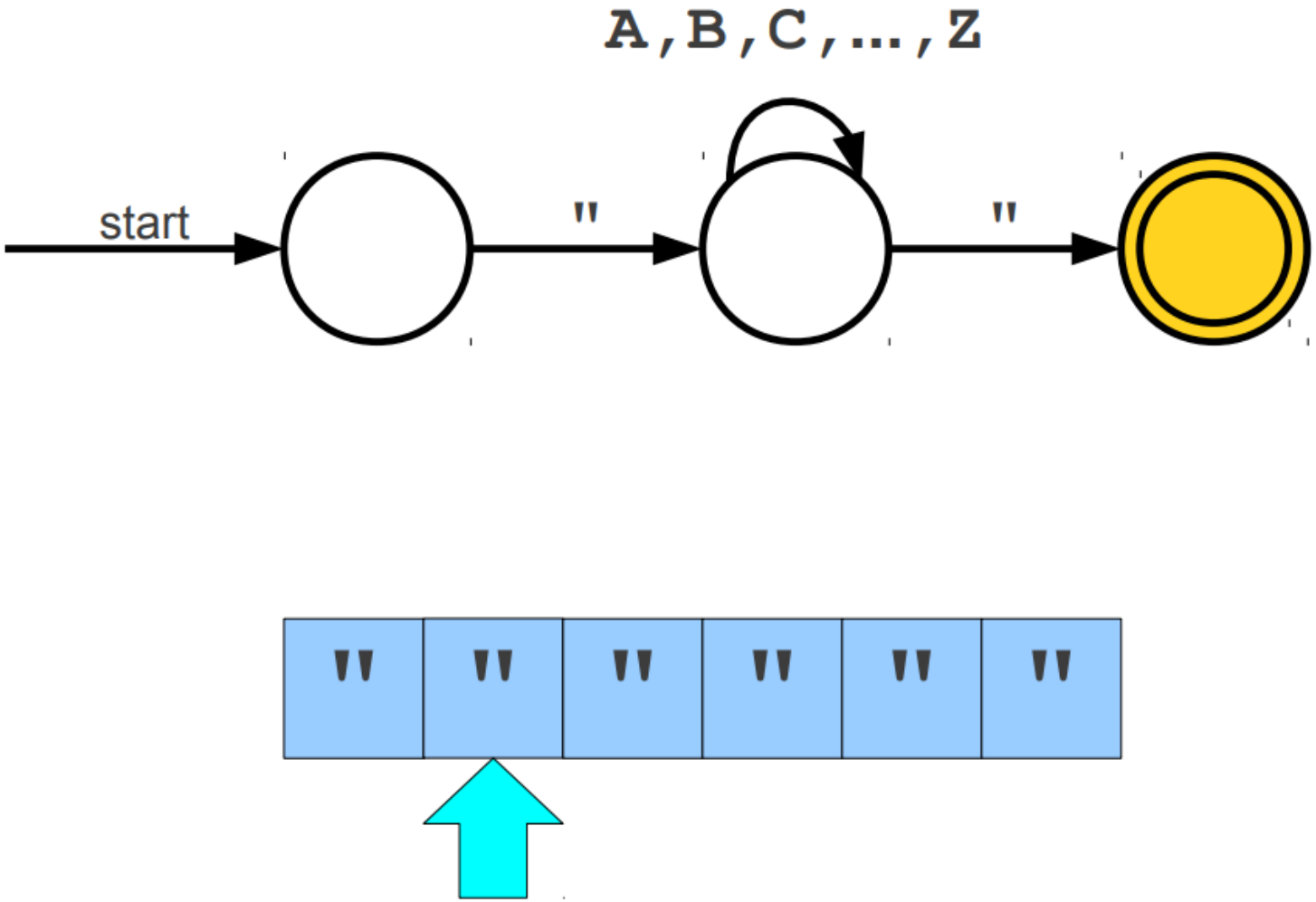
A Simple Automaton



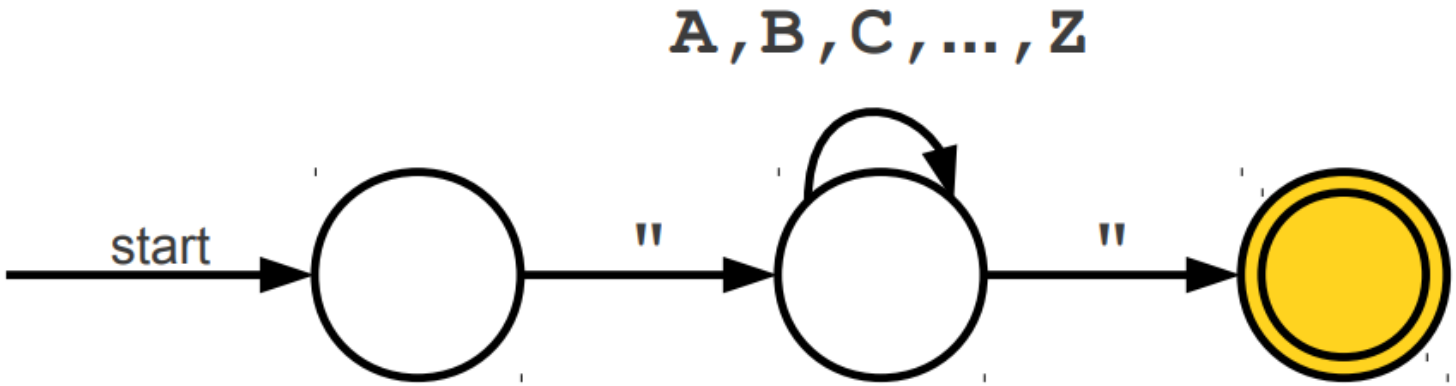
A Simple Automaton

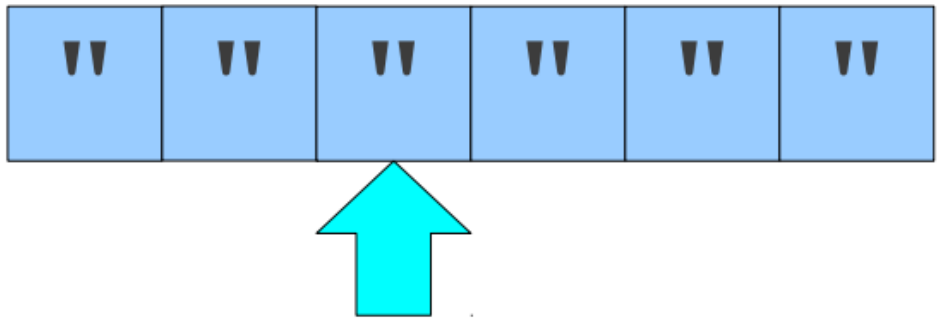


A Simple Automaton

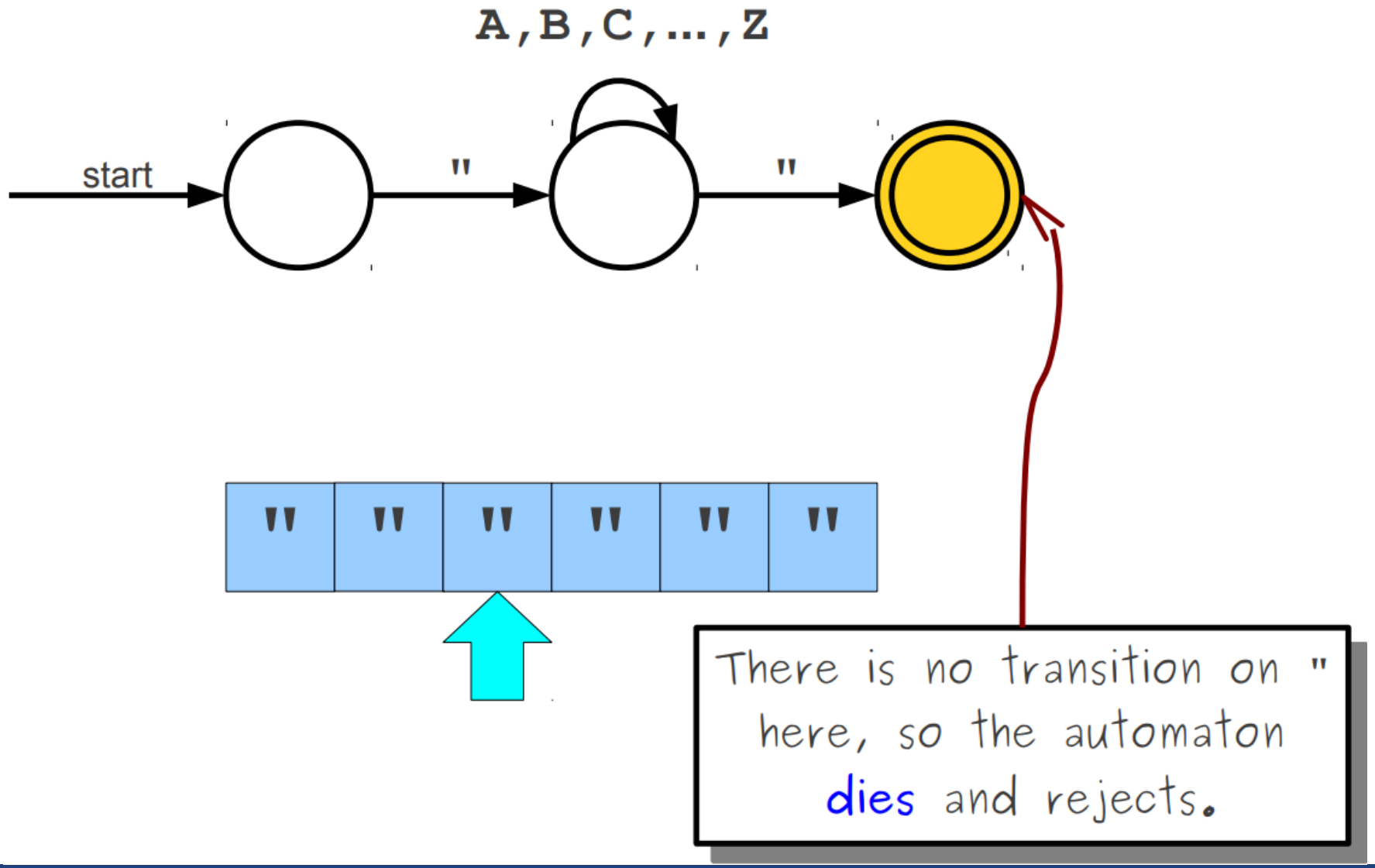


A Simple Automaton

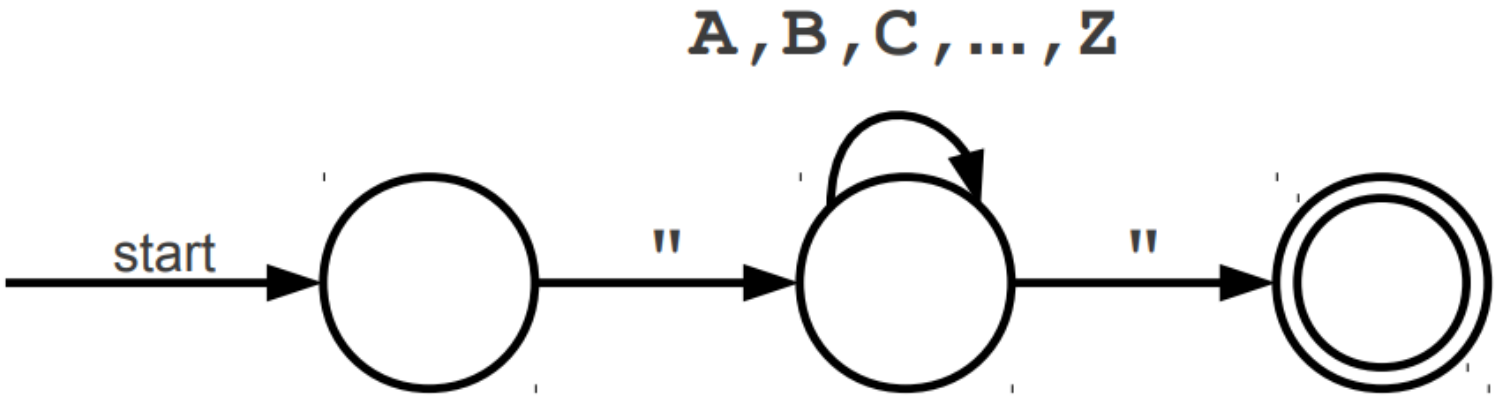


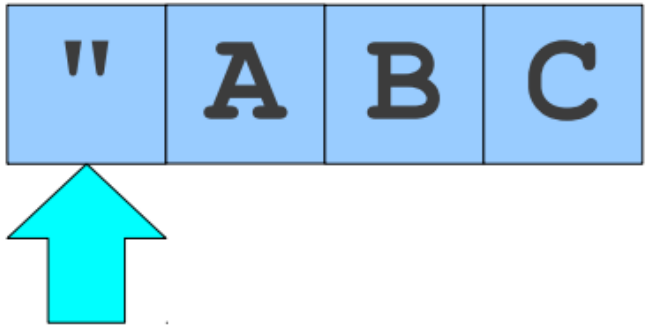


A Simple Automaton

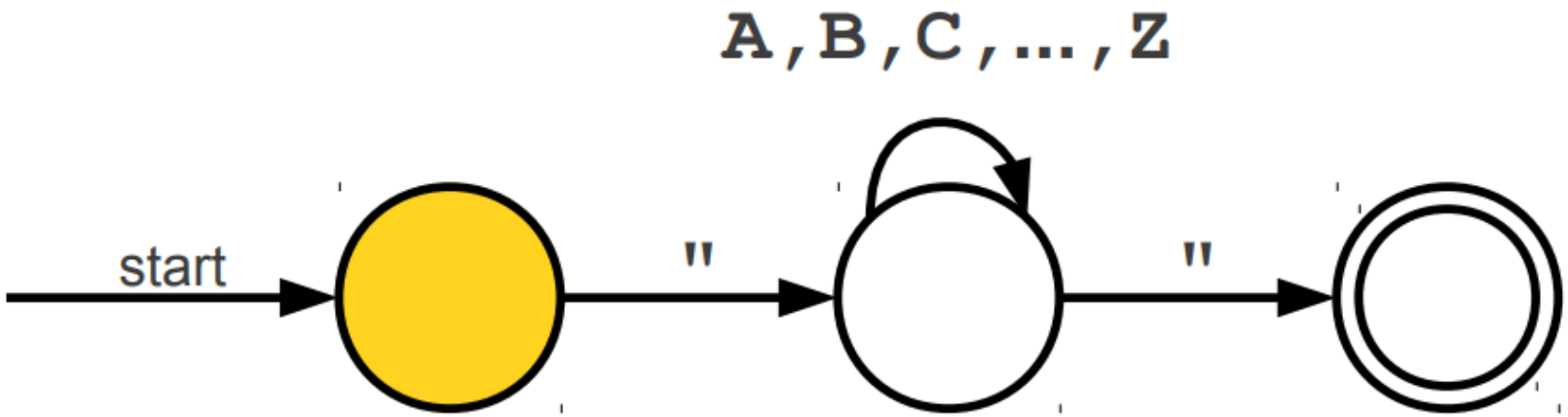


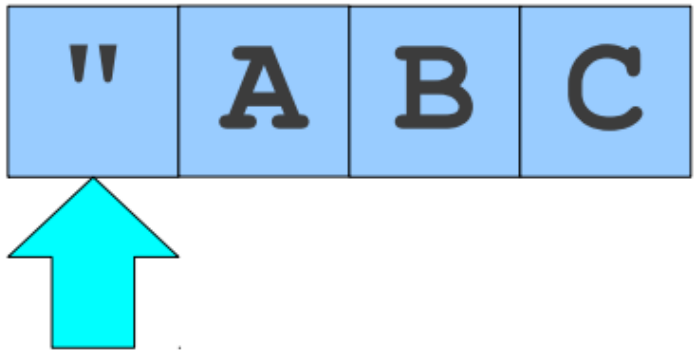
A Simple Automaton



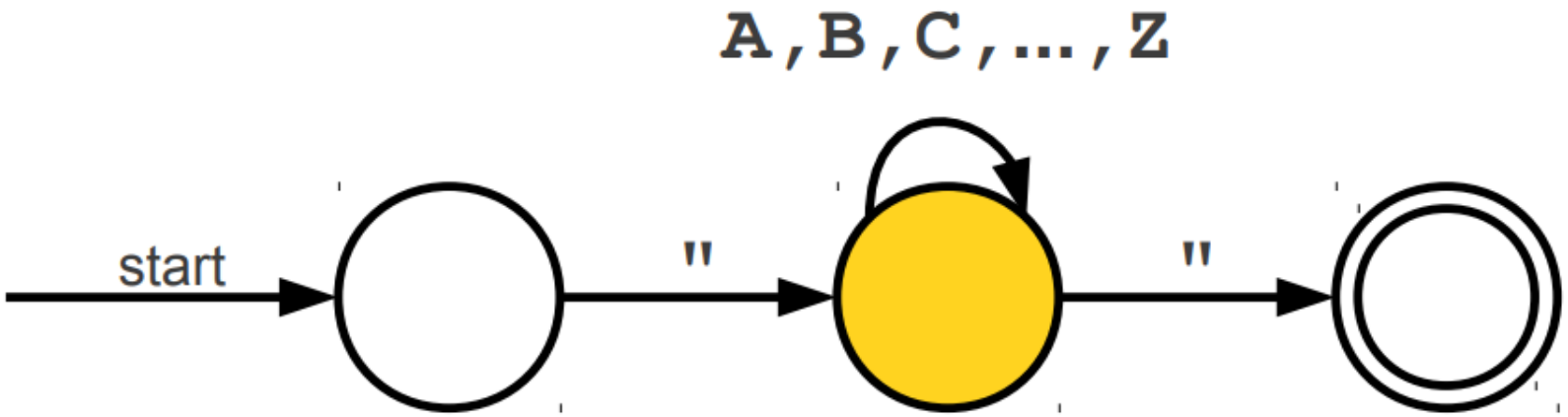


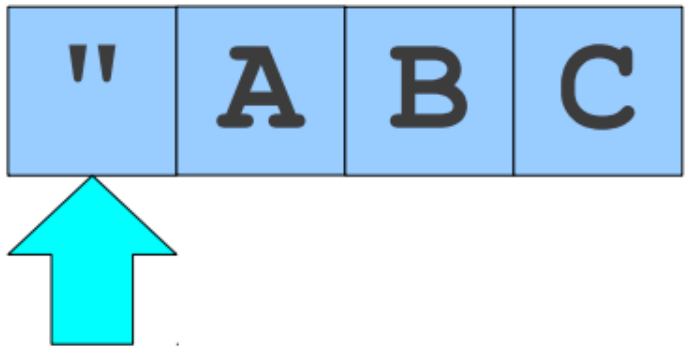
A Simple Automaton



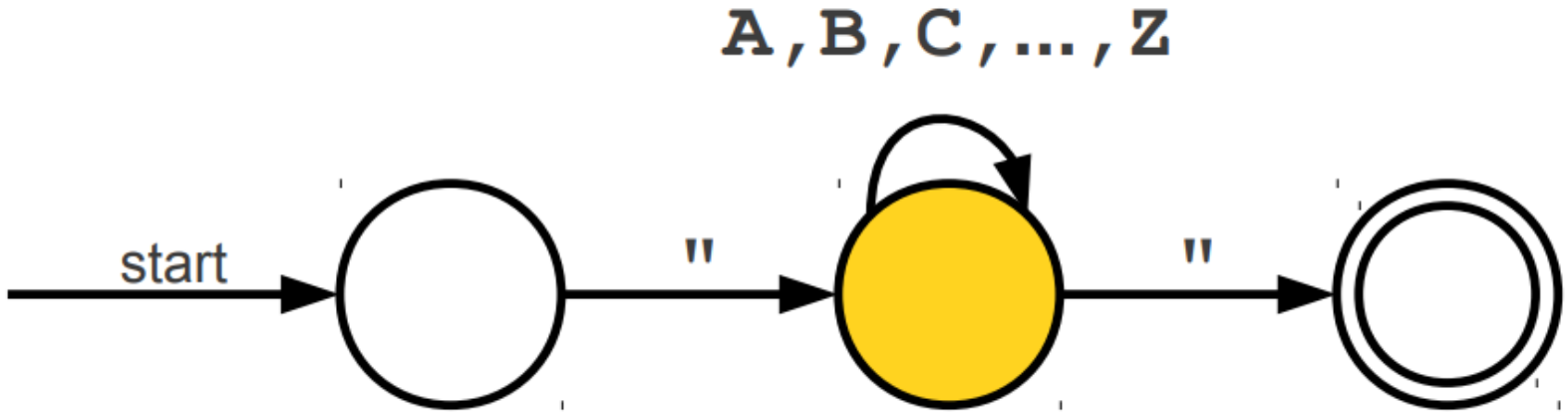


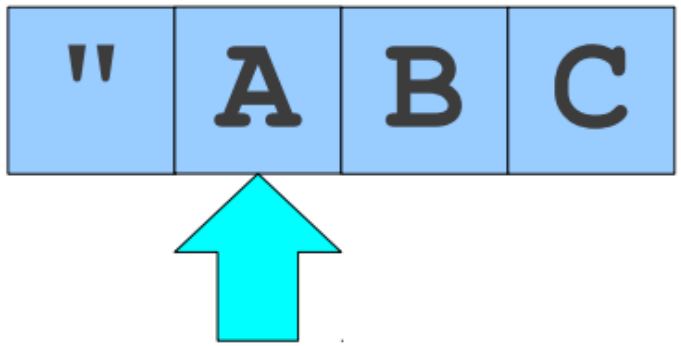
A Simple Automaton



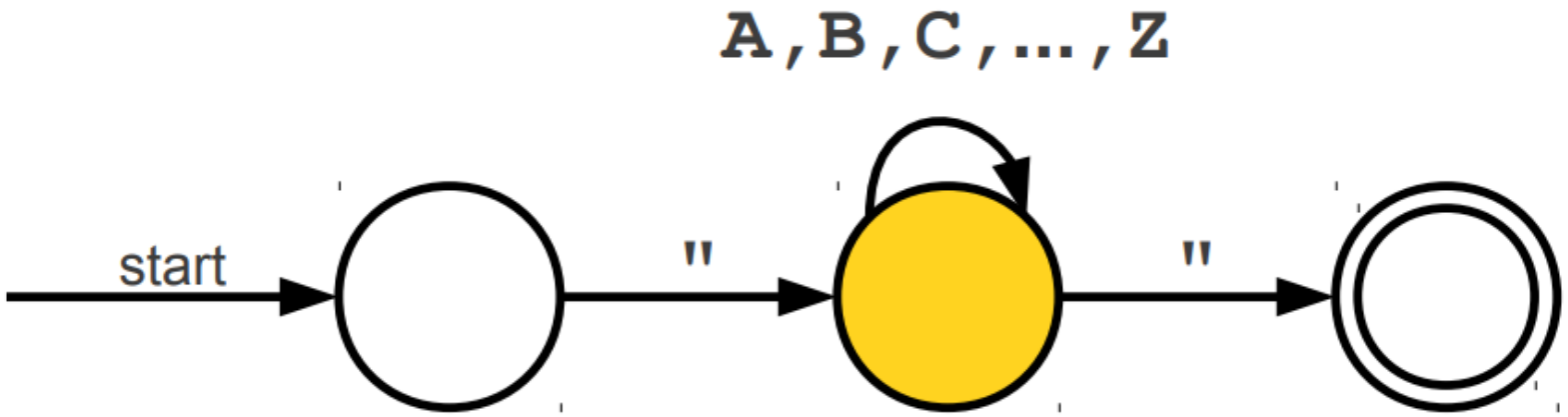


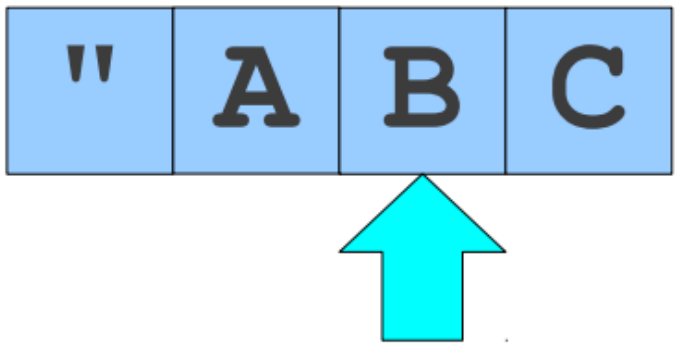
A Simple Automaton



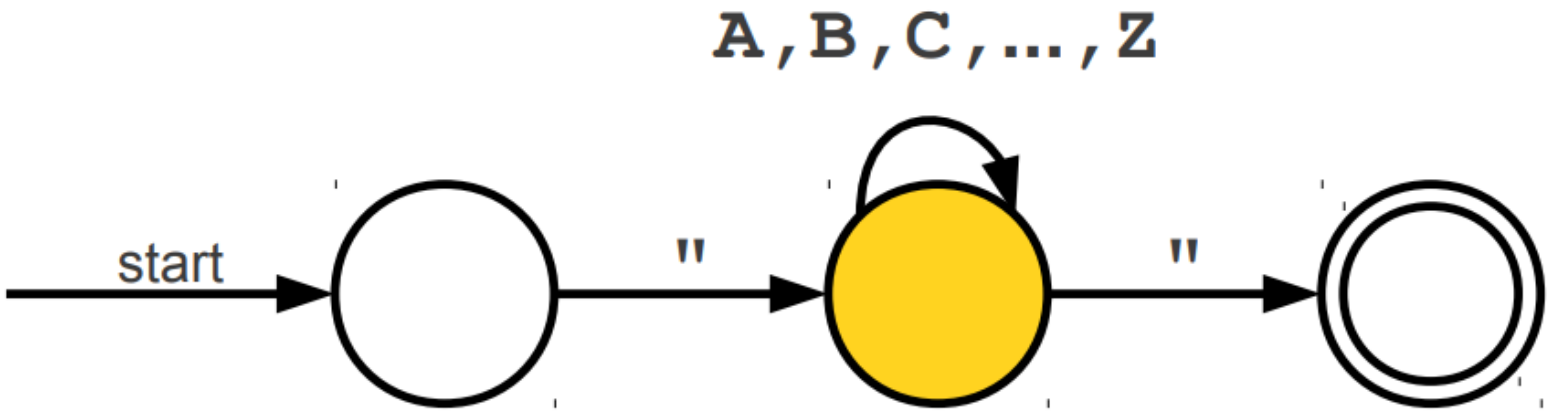


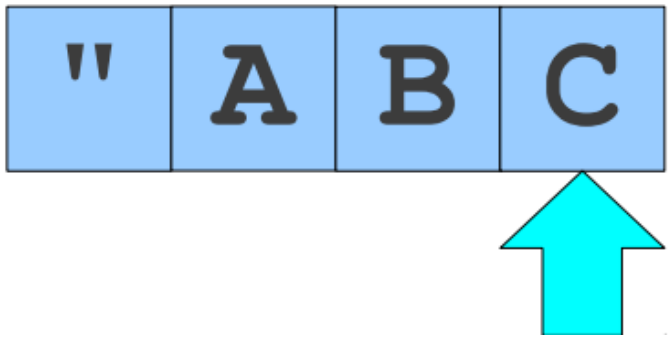
A Simple Automaton



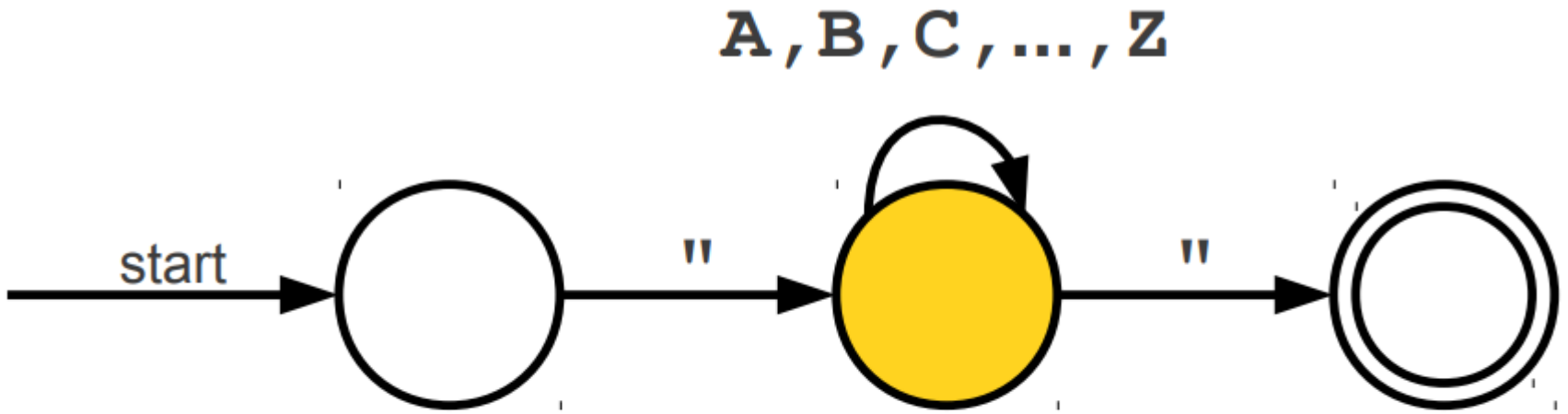


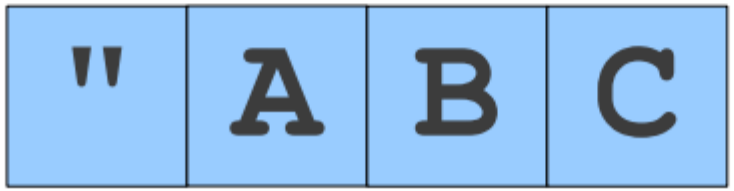
A Simple Automaton



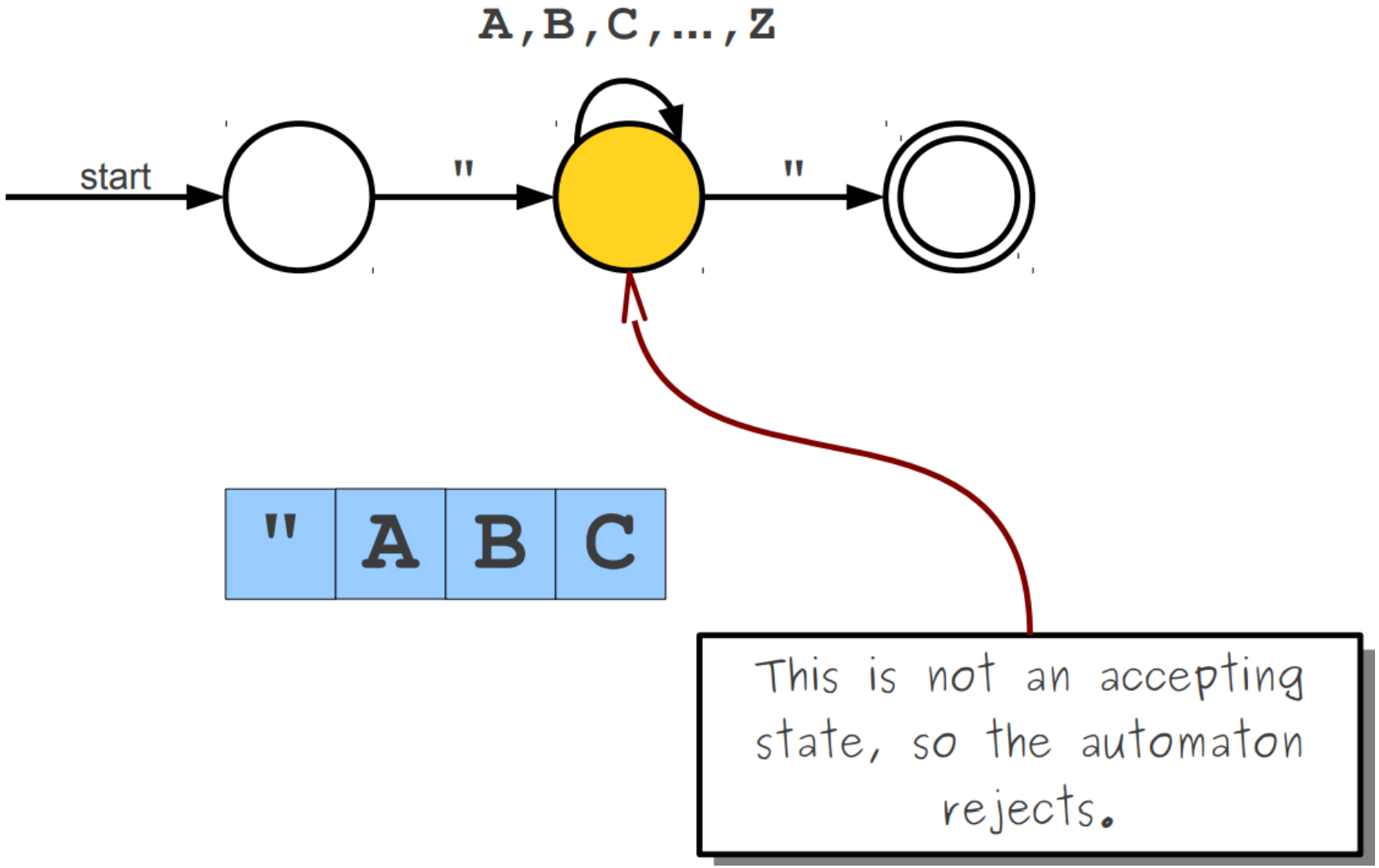


A Simple Automaton

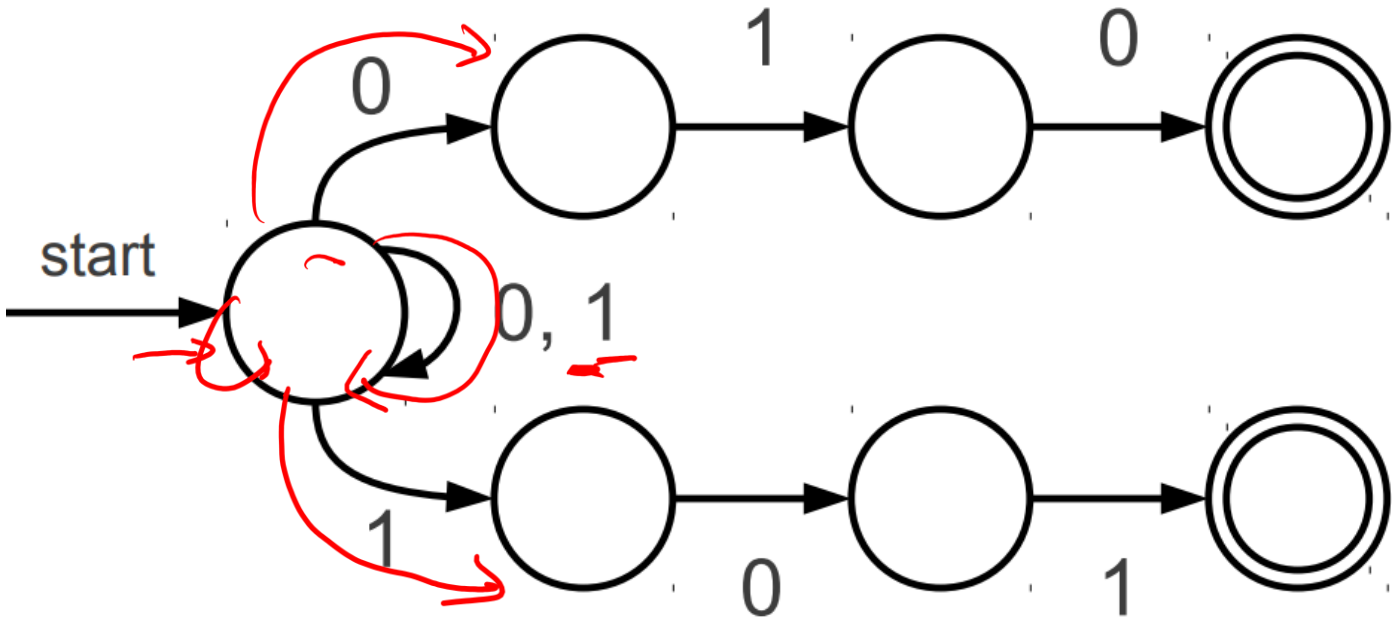




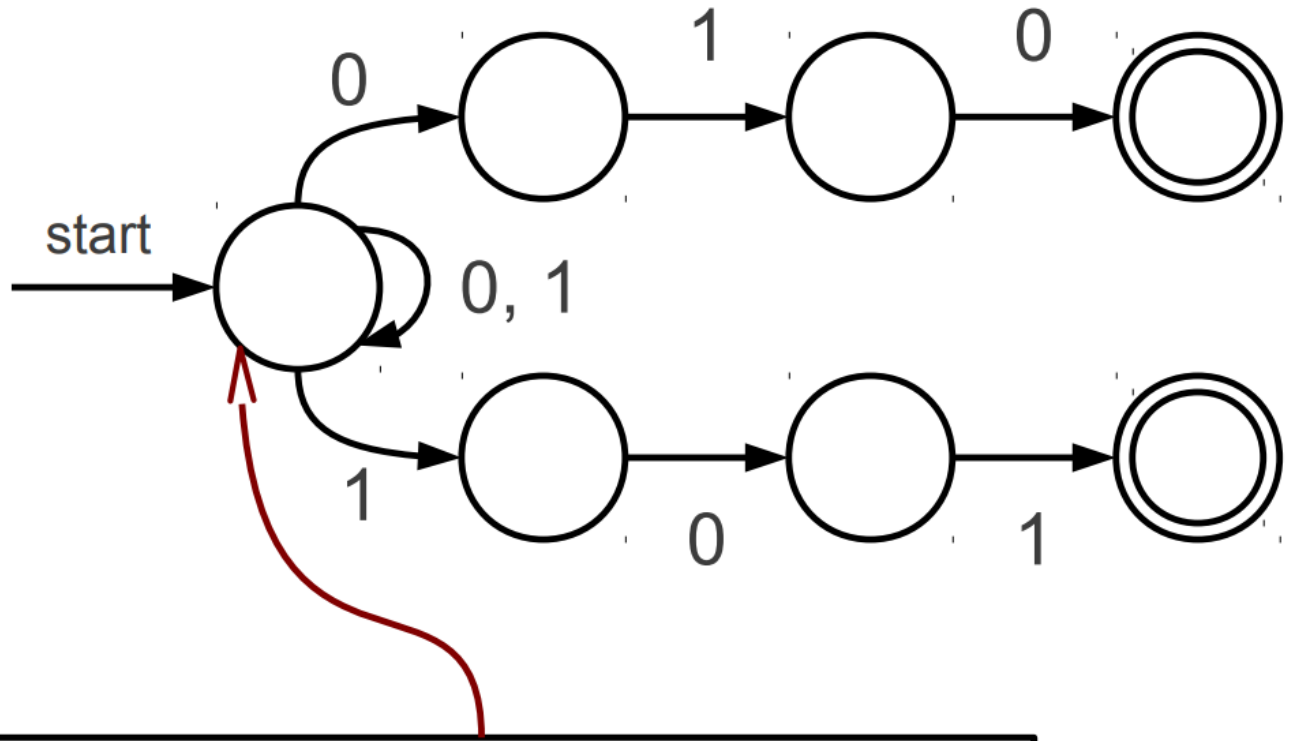
A Simple Automaton



A More Complex Automaton

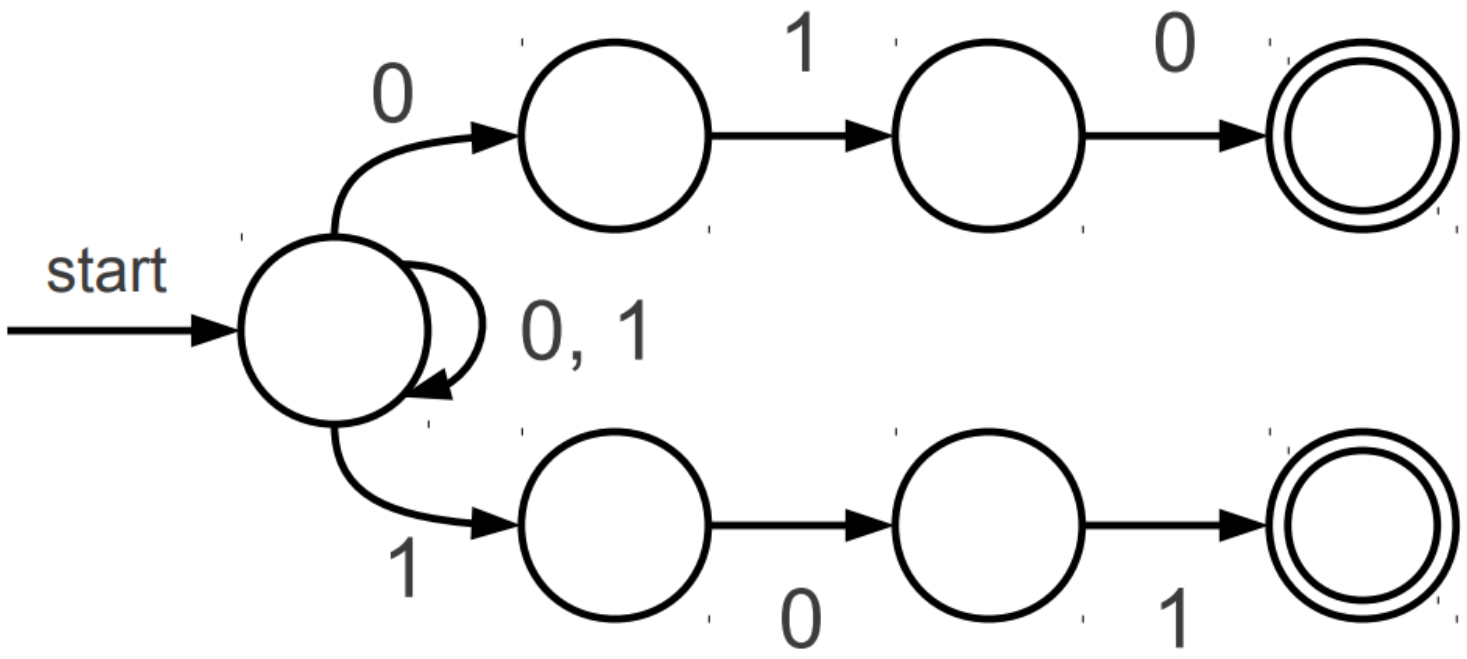


A More Complex Automaton



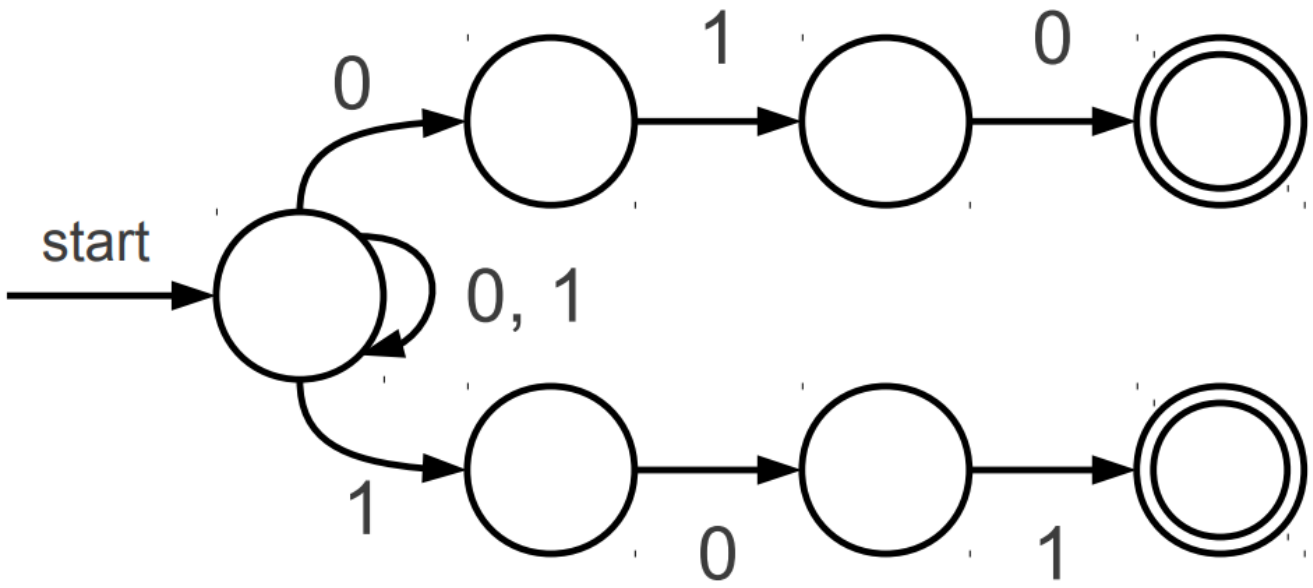
Notice that there are multiple transitions defined here on 0 and 1. If we read a 0 or 1 here, we follow *both* transitions and enter multiple states.

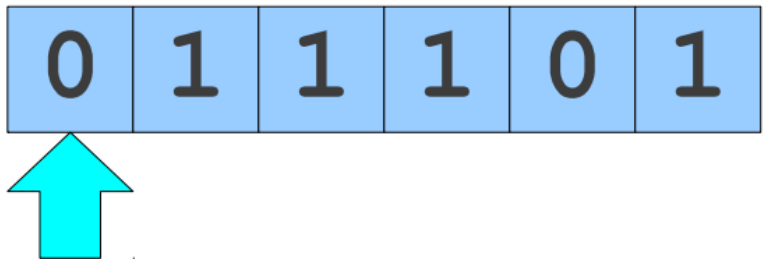
A More Complex Automaton



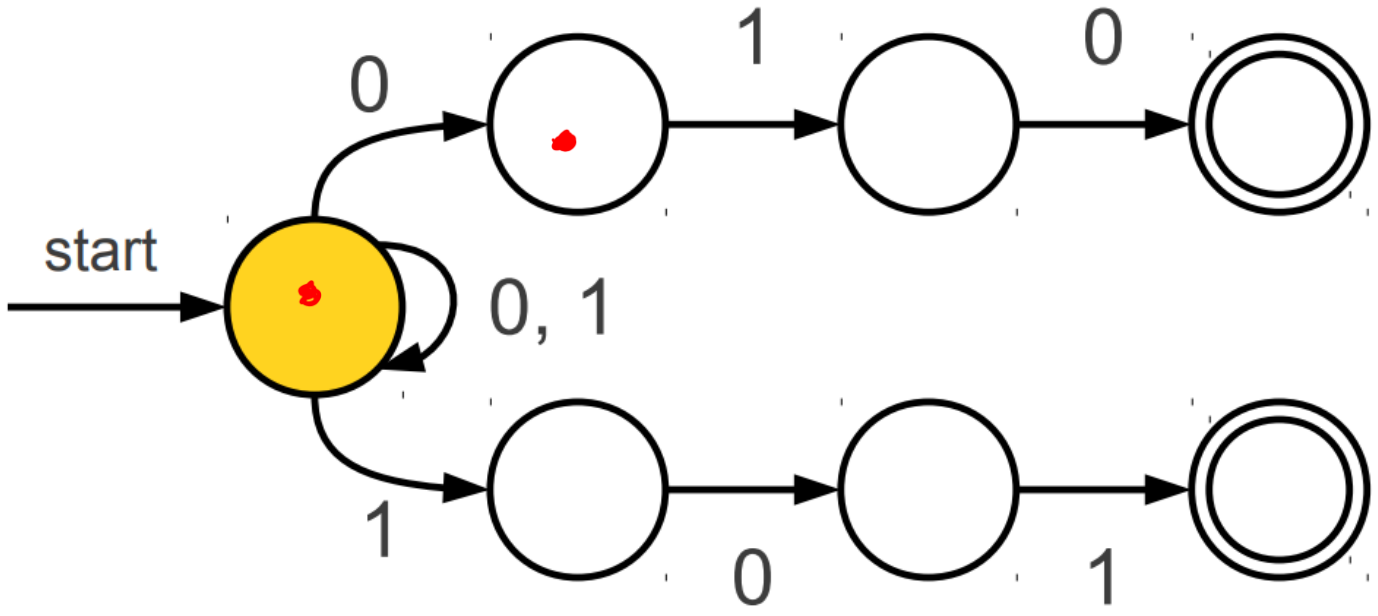
0	1	1	1	0	1

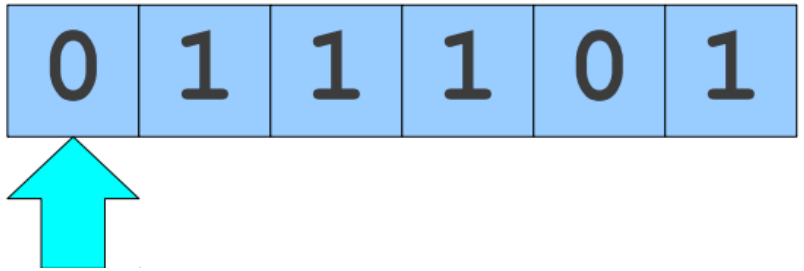
A More Complex Automaton



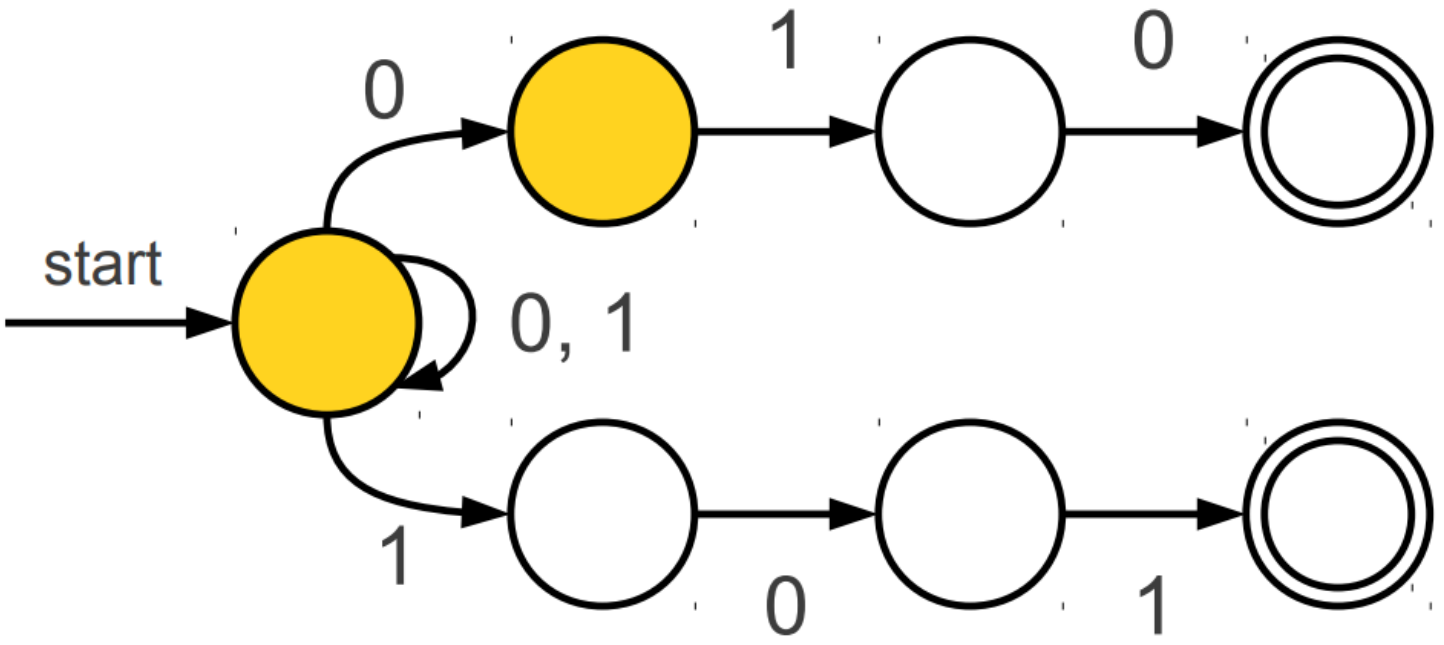


A More Complex Automaton

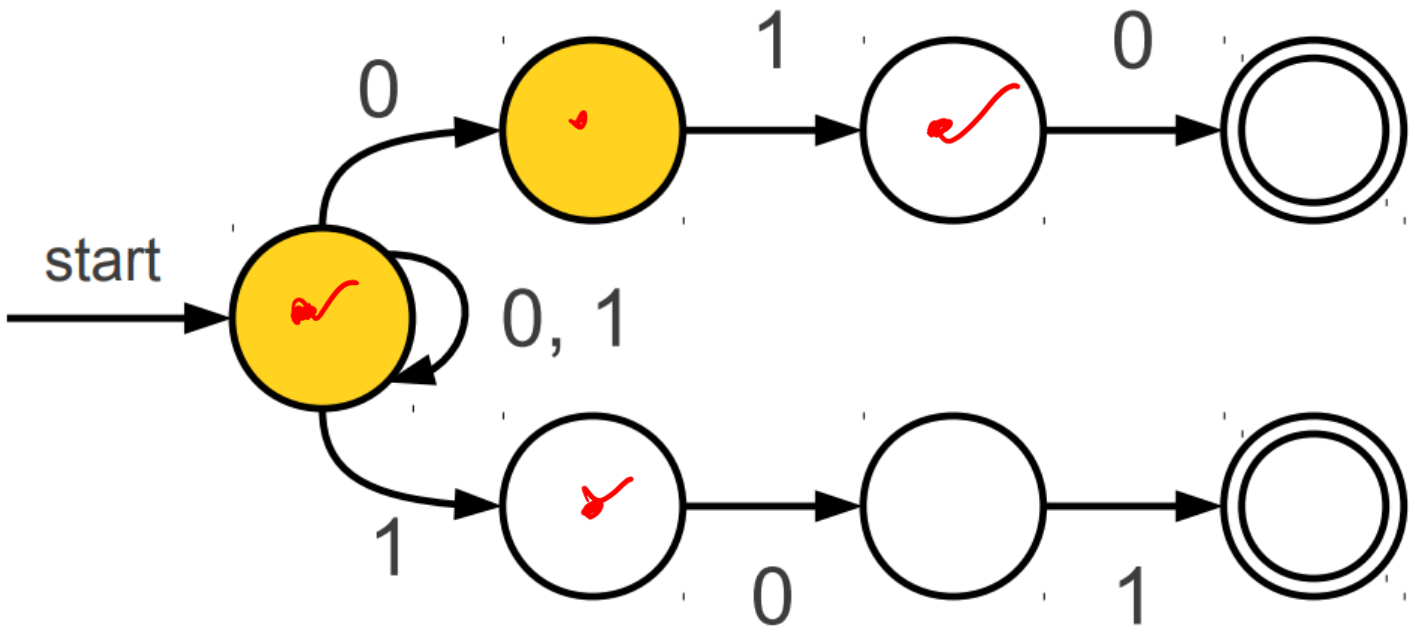


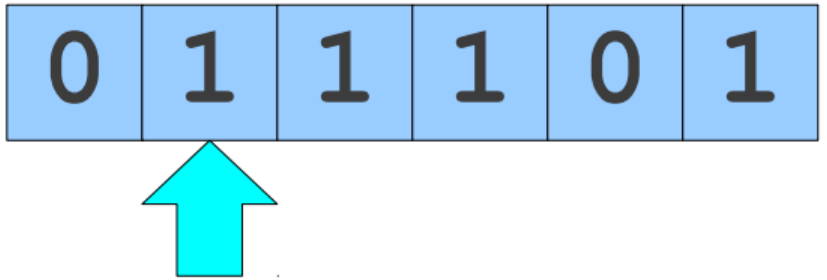


A More Complex Automaton

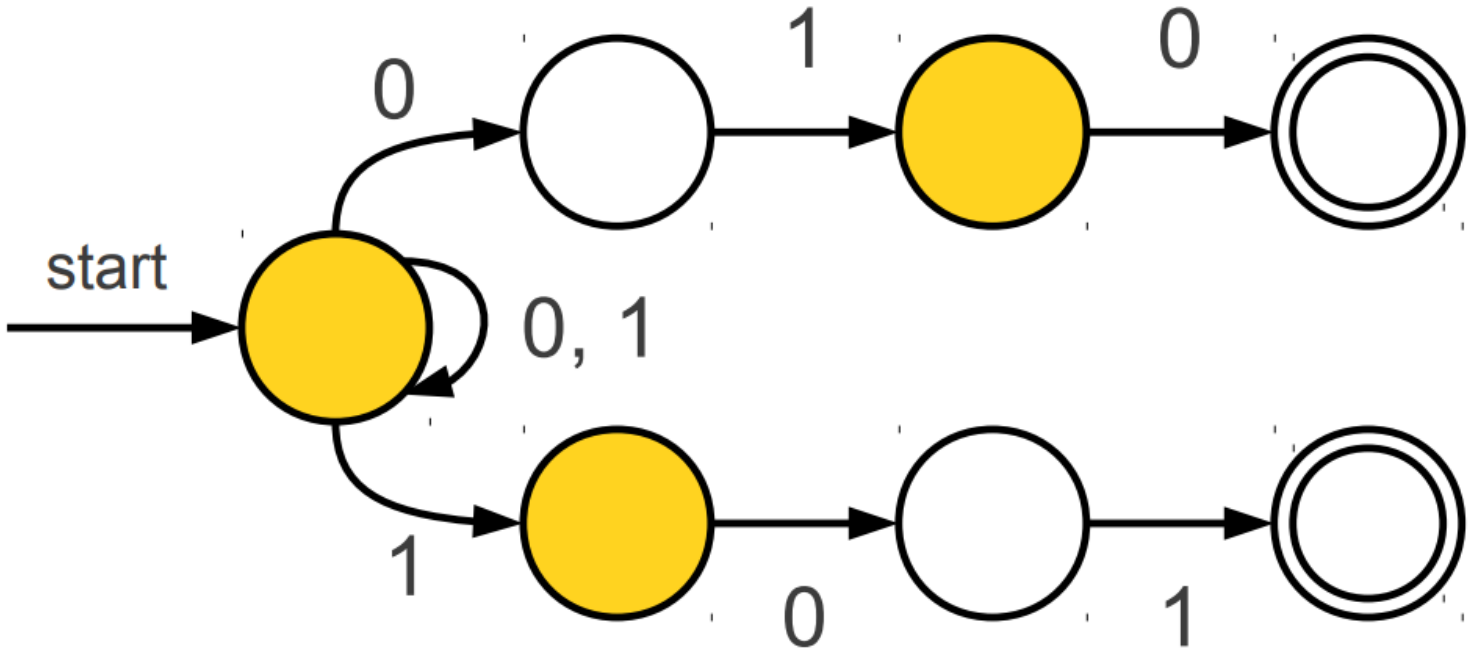


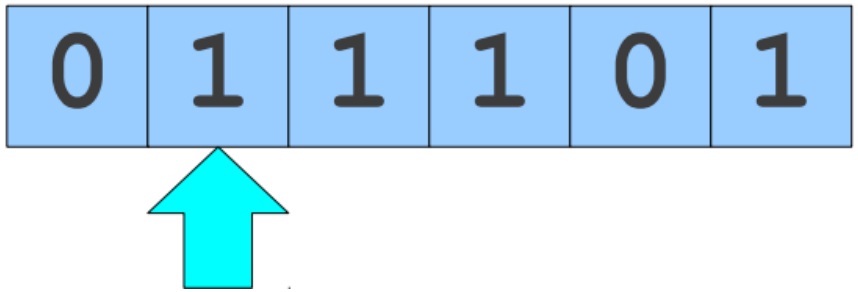
A More Complex Automaton



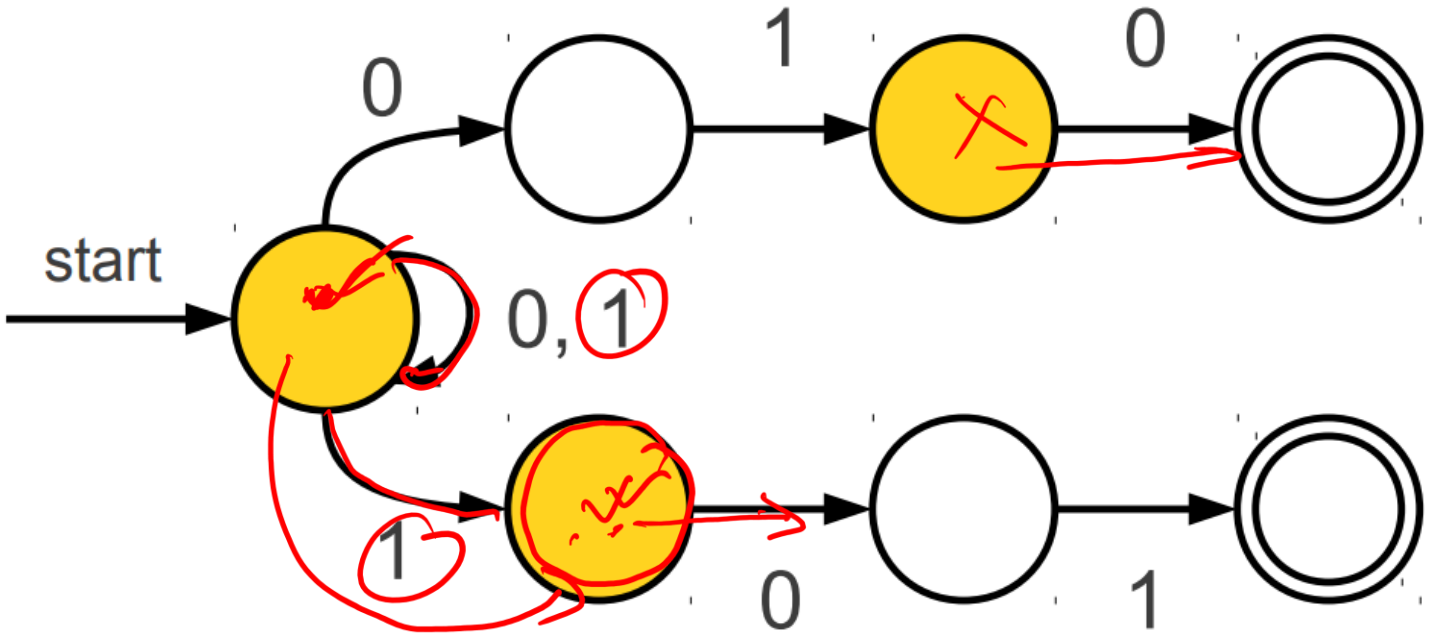


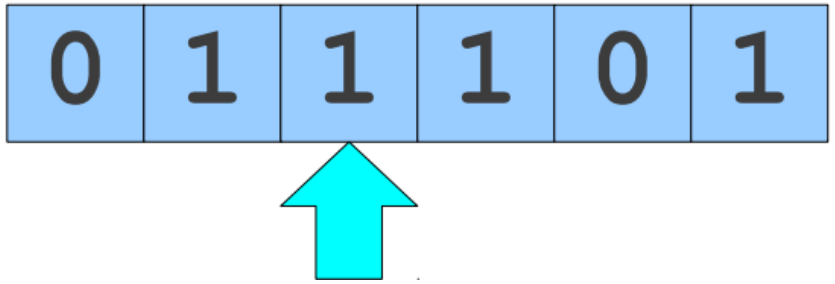
A More Complex Automaton



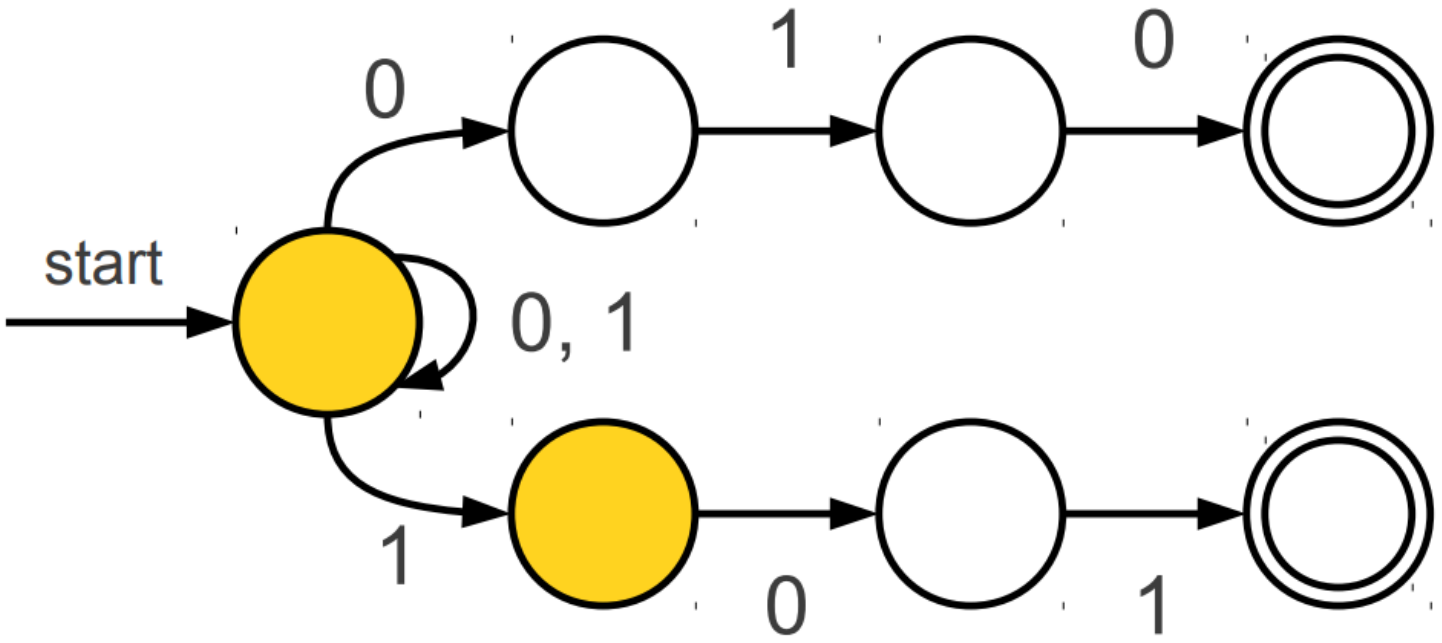


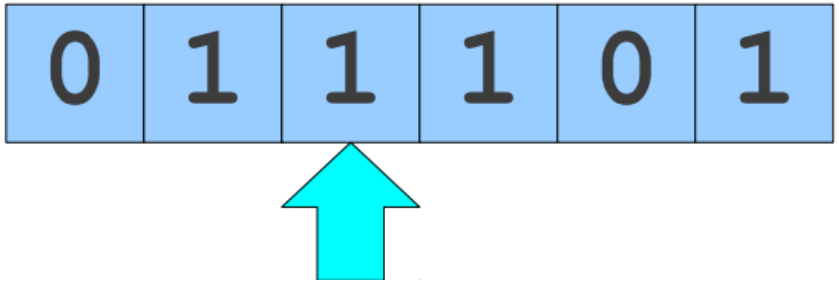
A More Complex Automaton



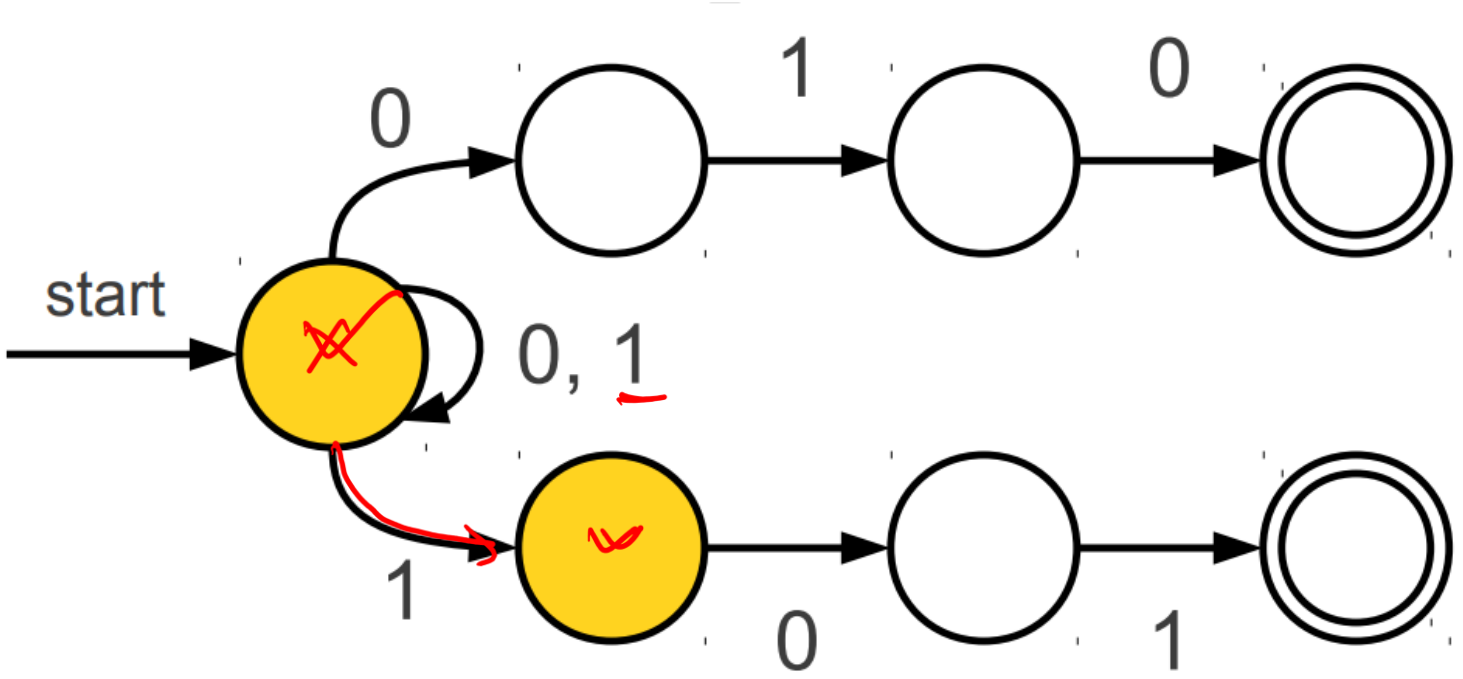


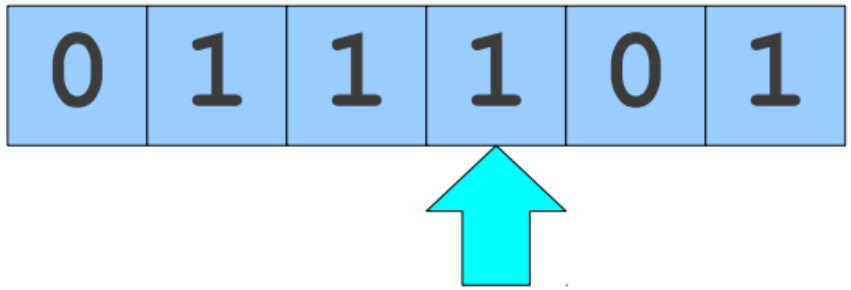
A More Complex Automaton



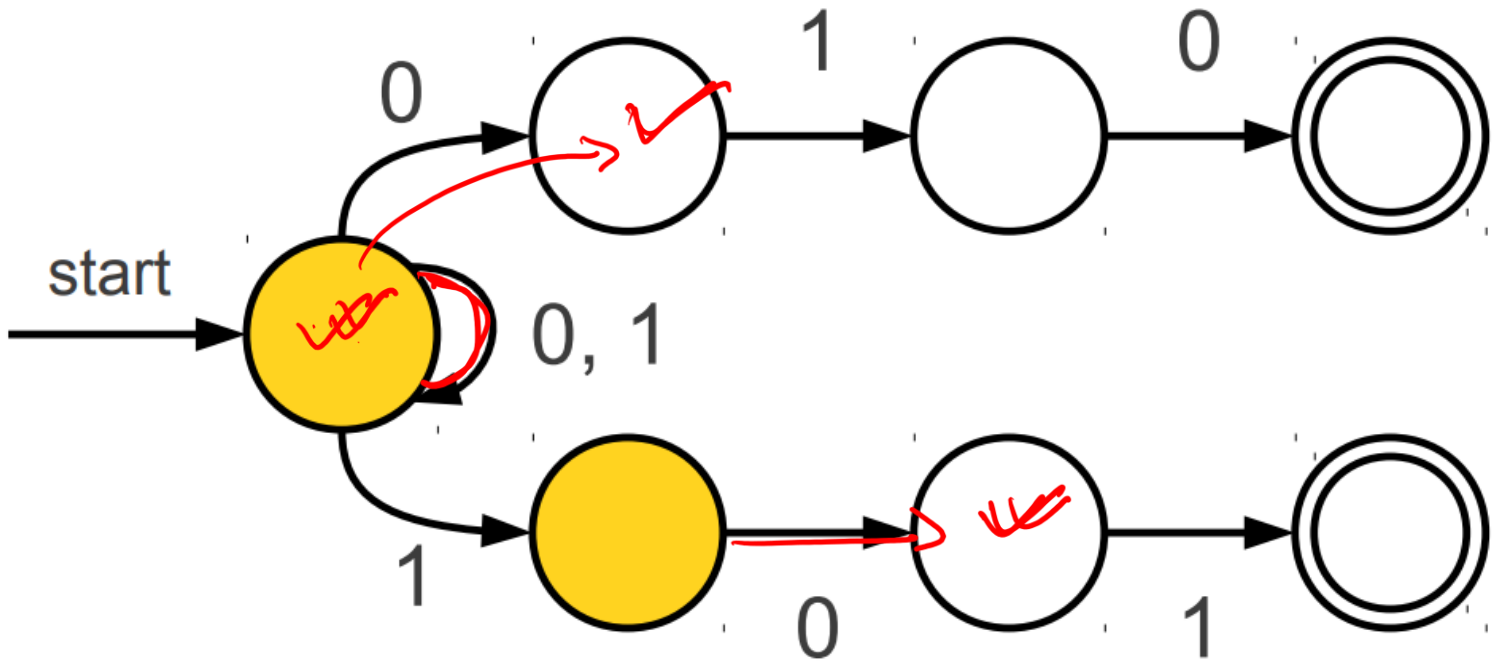


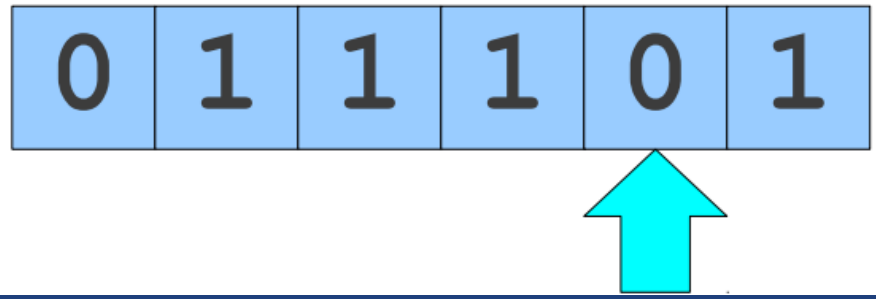
A More Complex Automaton



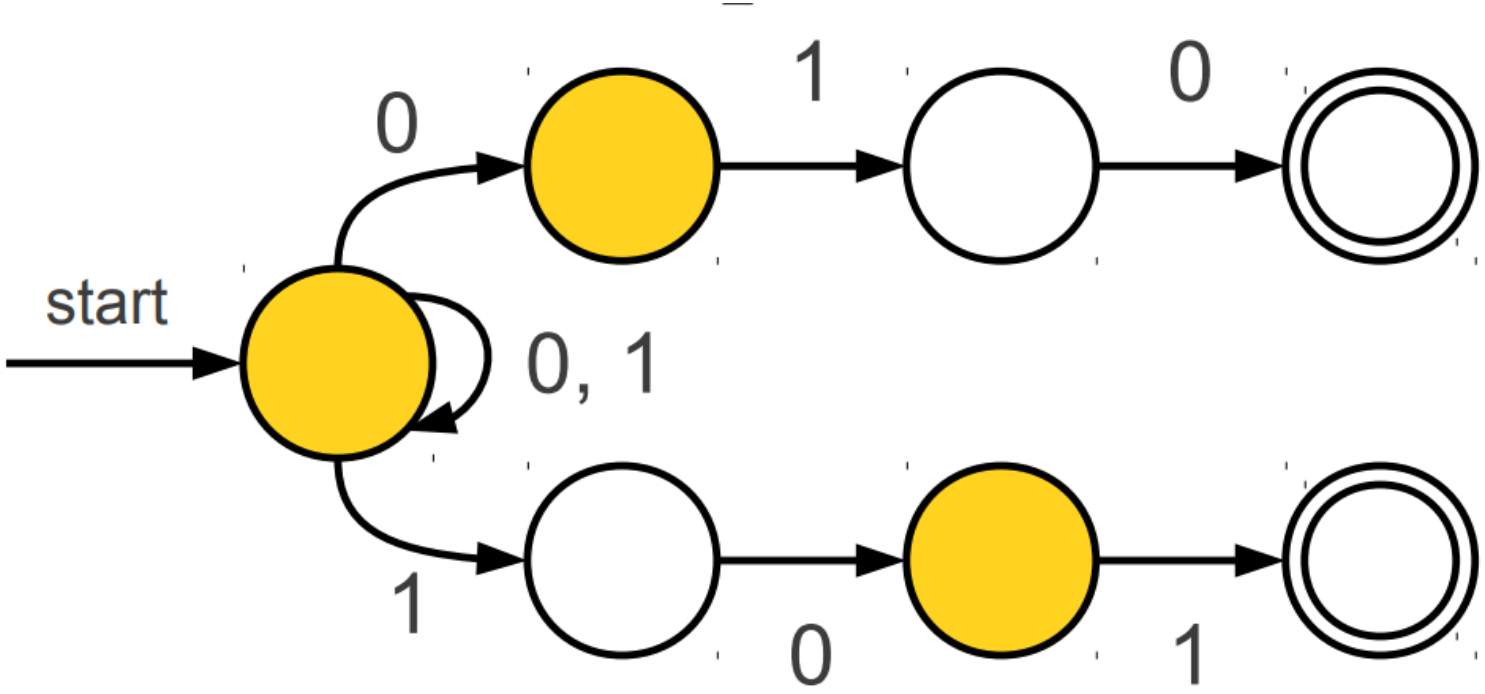


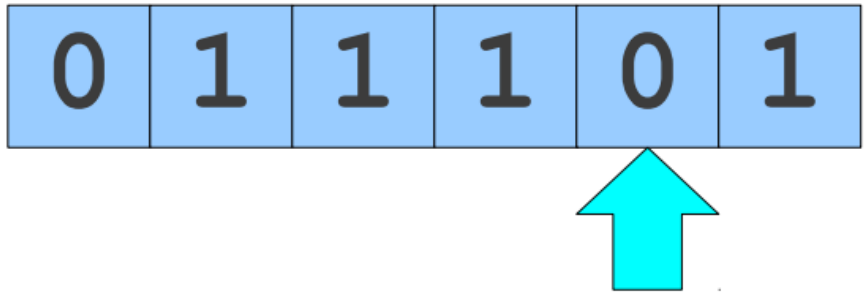
A More Complex Automaton



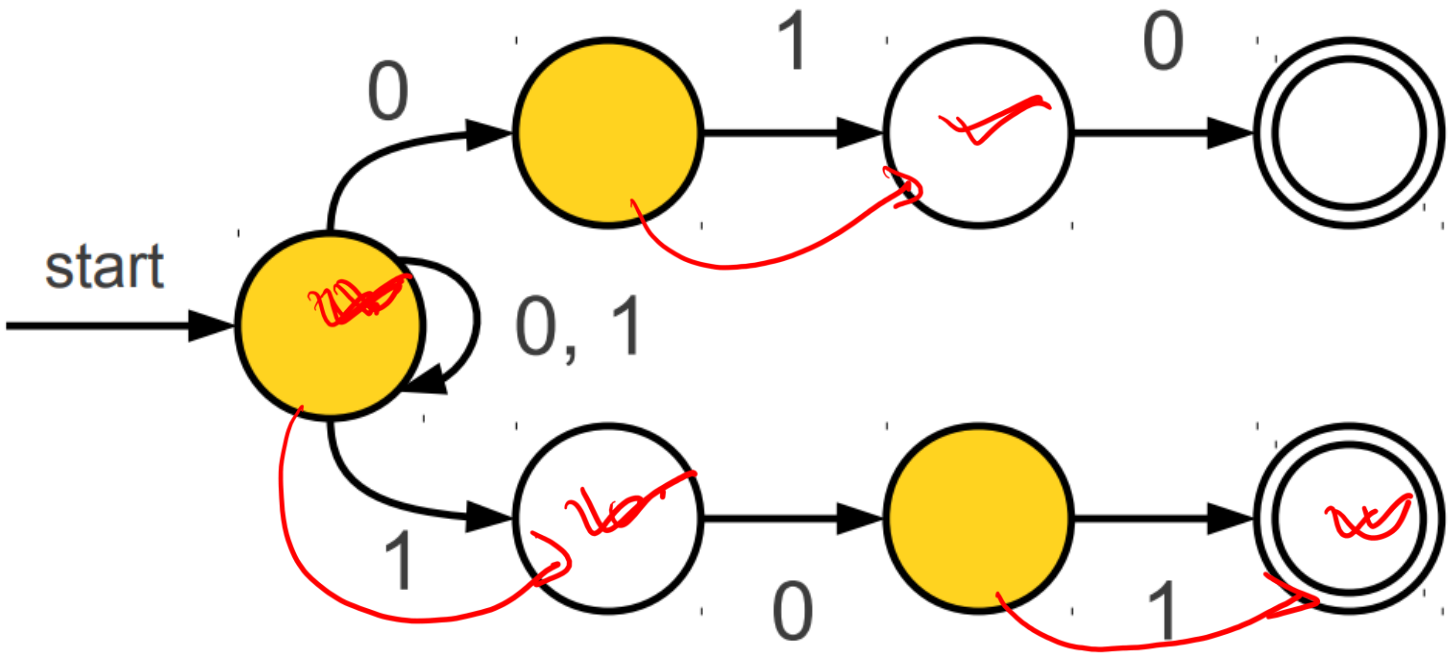


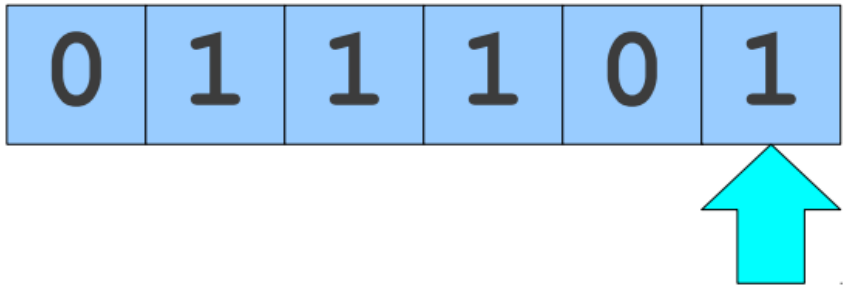
A More Complex Automaton



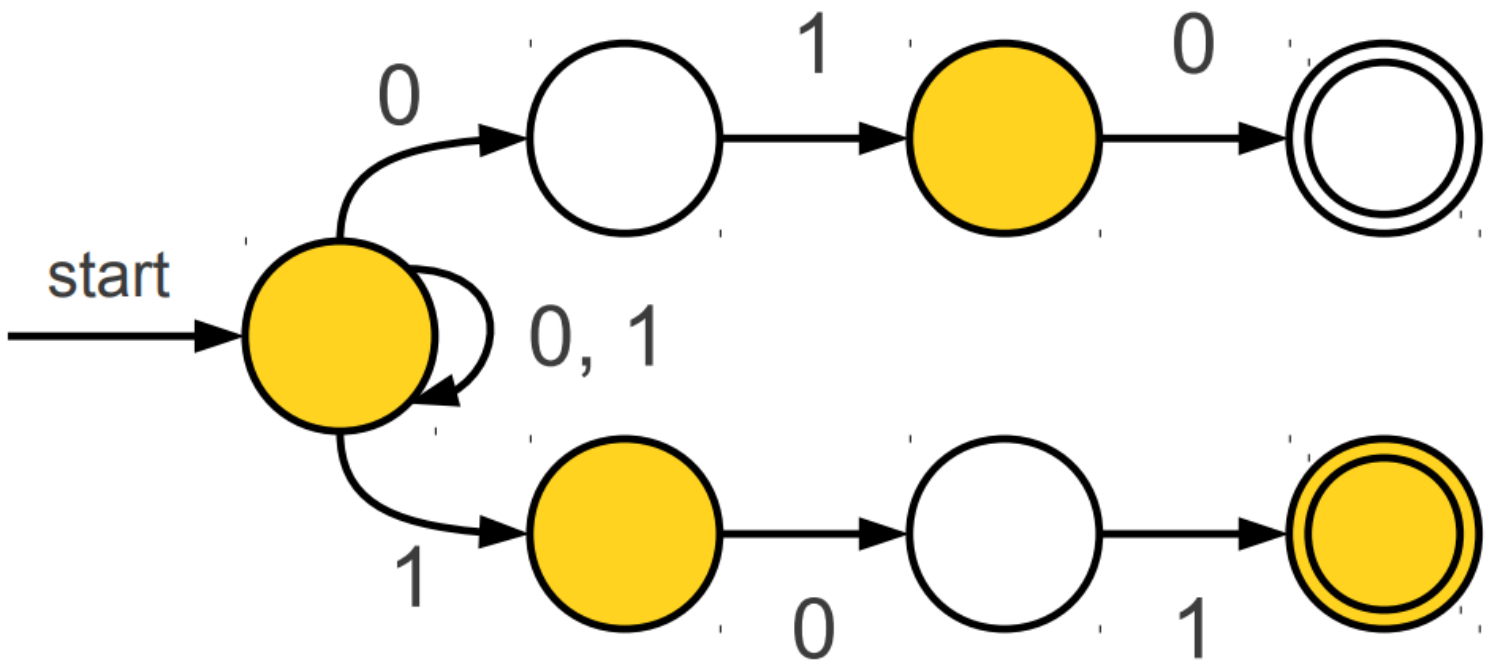


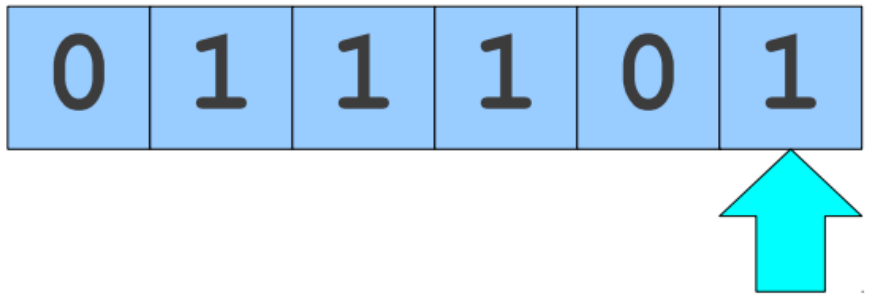
A More Complex Automaton



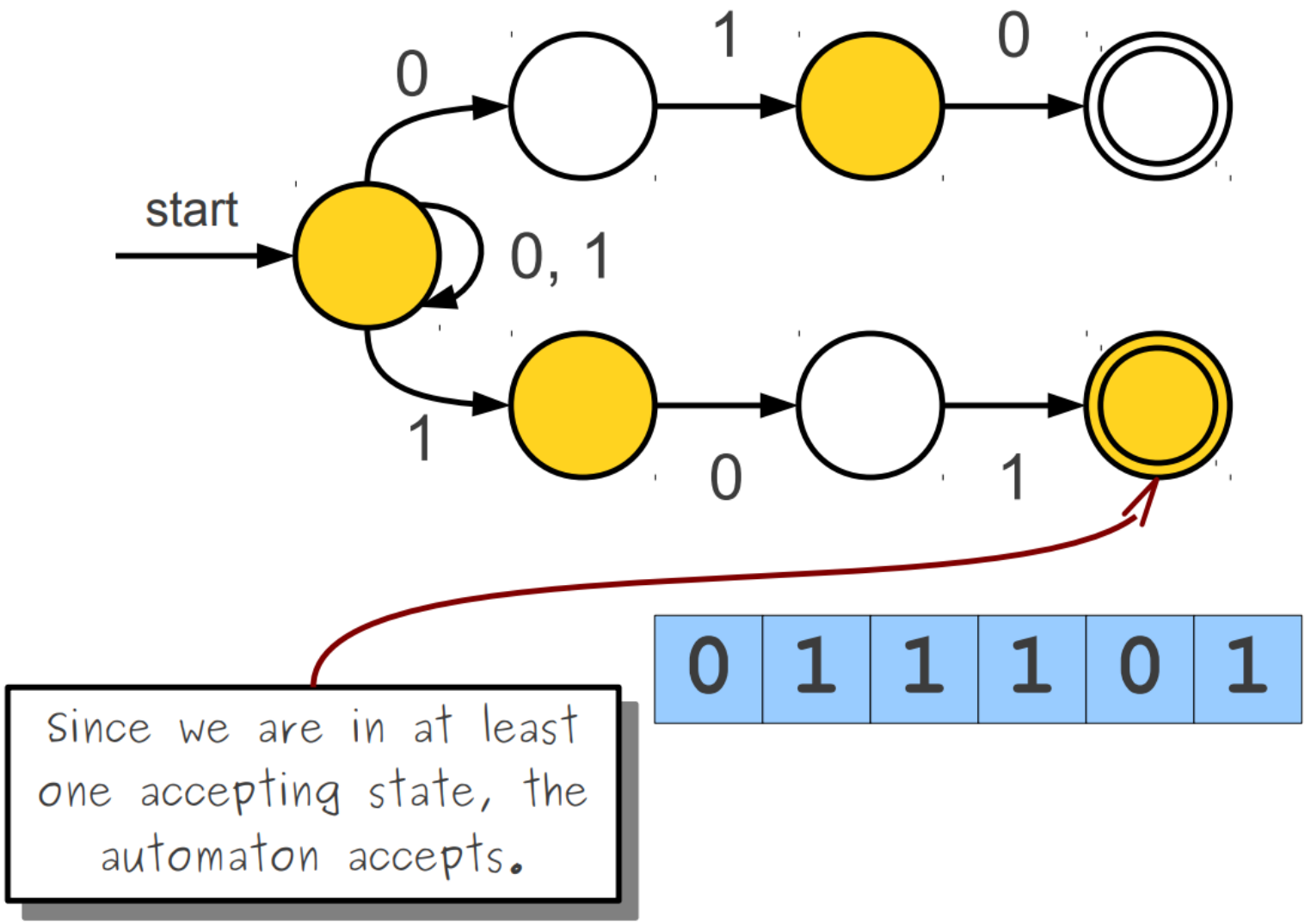


A More Complex Automaton



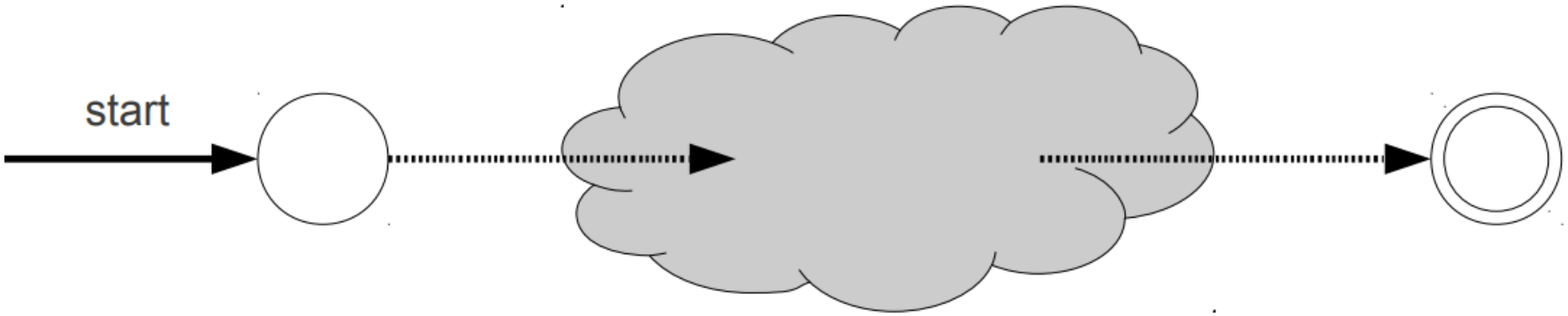


A More Complex Automaton

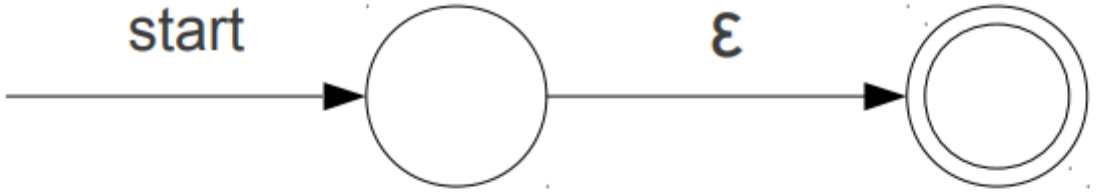


From RE to NFAs

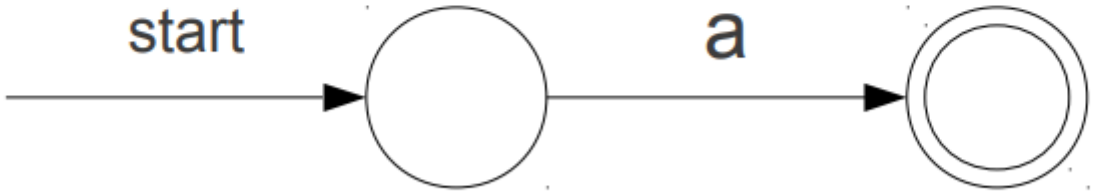
- There is a (beautiful!) procedure from converting a regular expression to an NFA



Base Cases

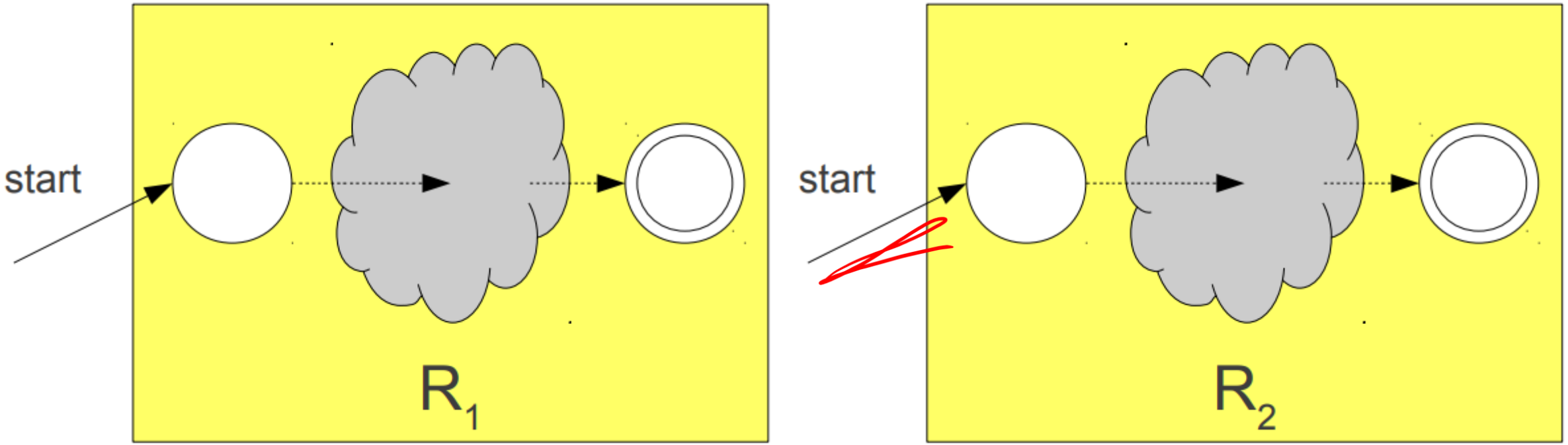


Automaton for ε

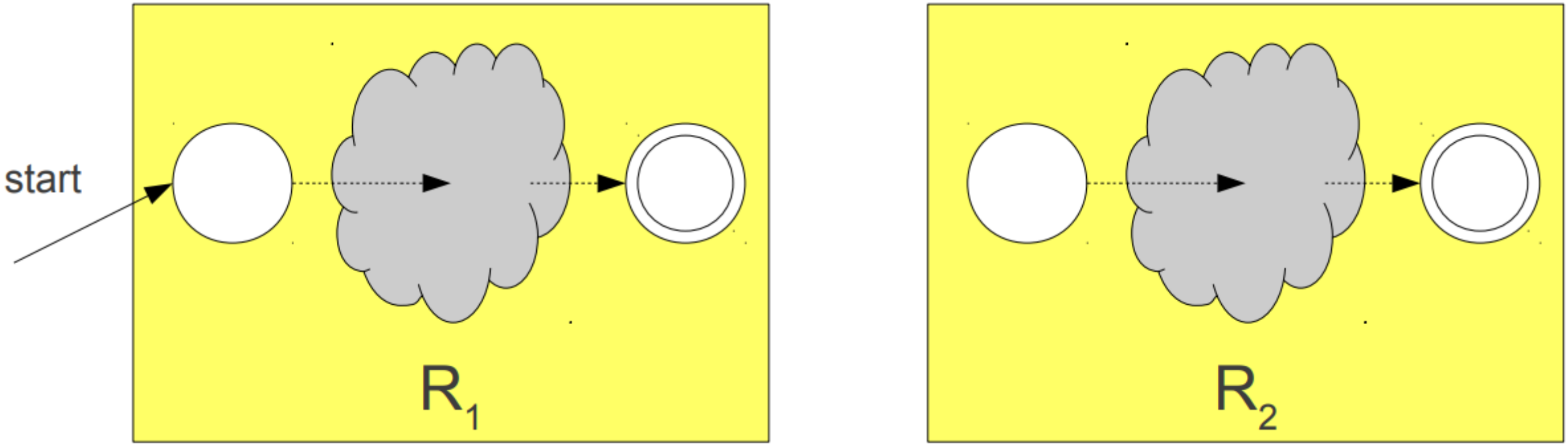


Automaton for single character `a`

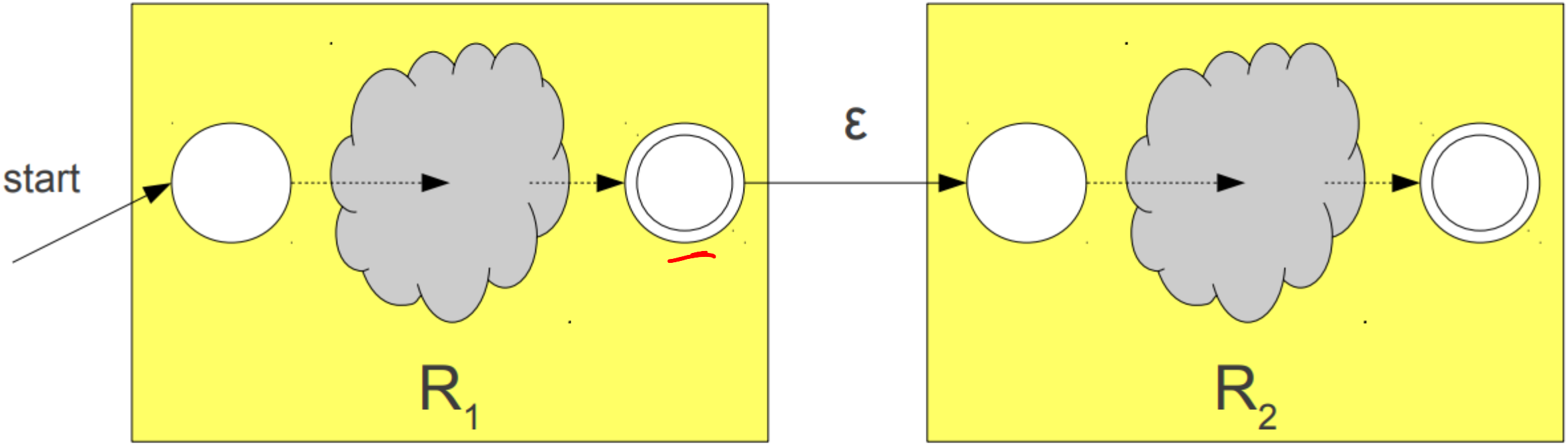
Construction for R_1 R_2



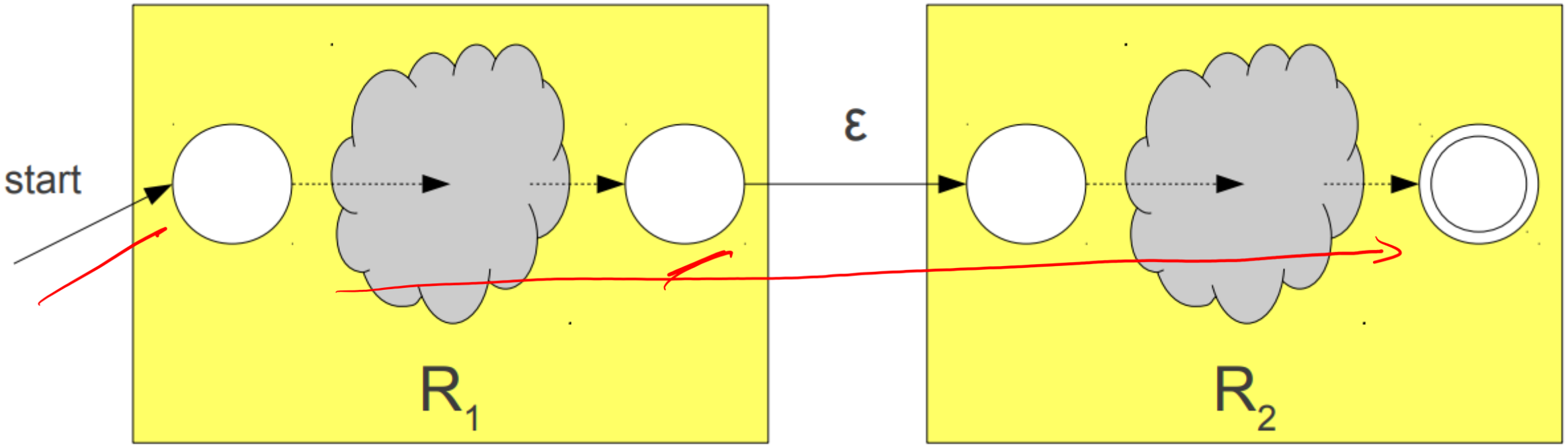
Construction for R_1 R_2



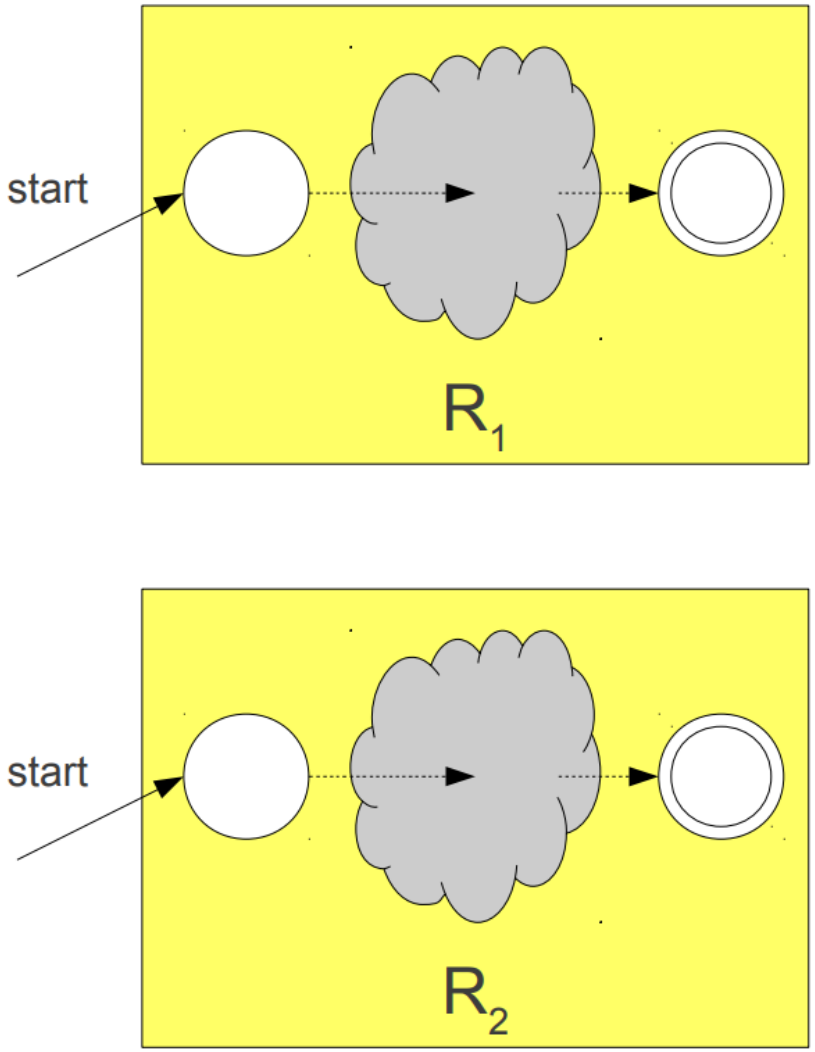
Construction for R_1 R_2



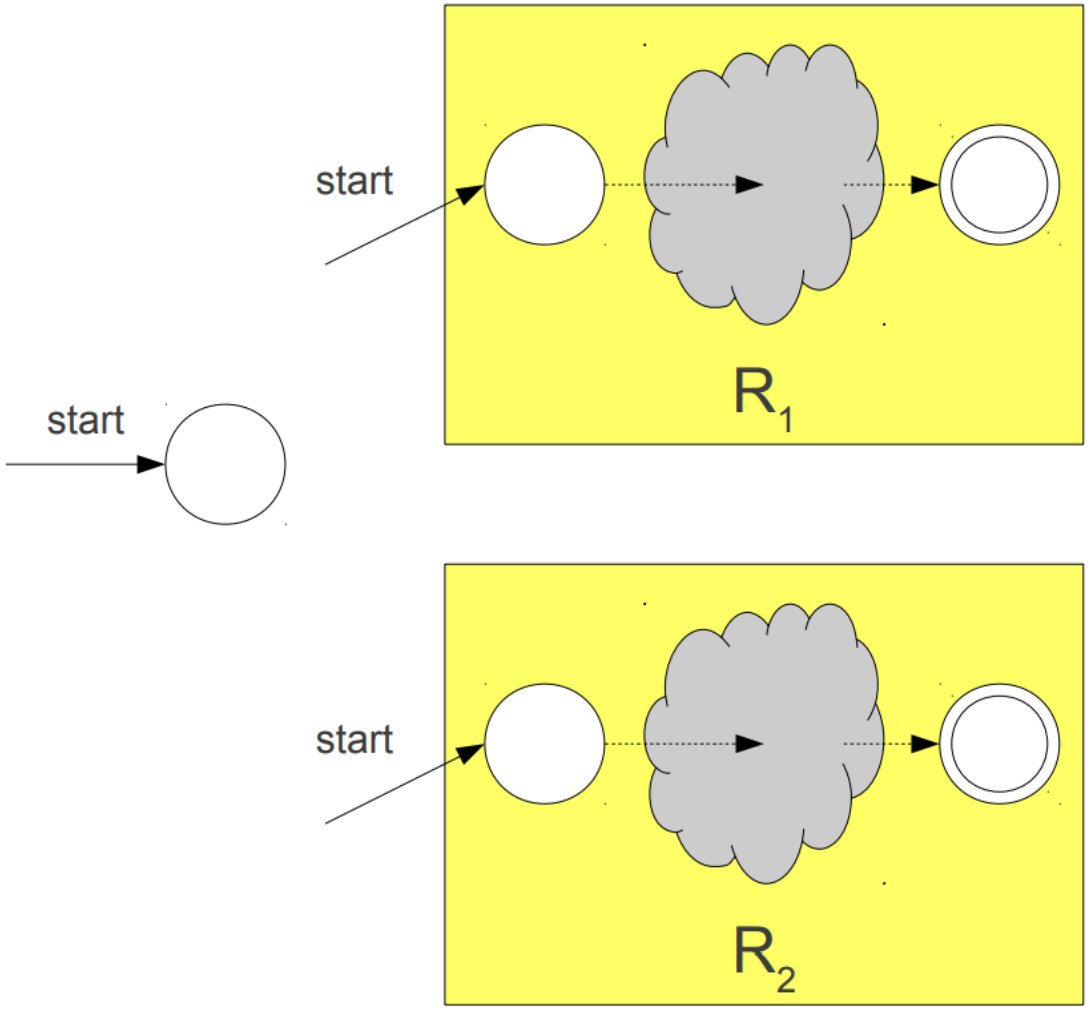
Construction for R_1 R_2



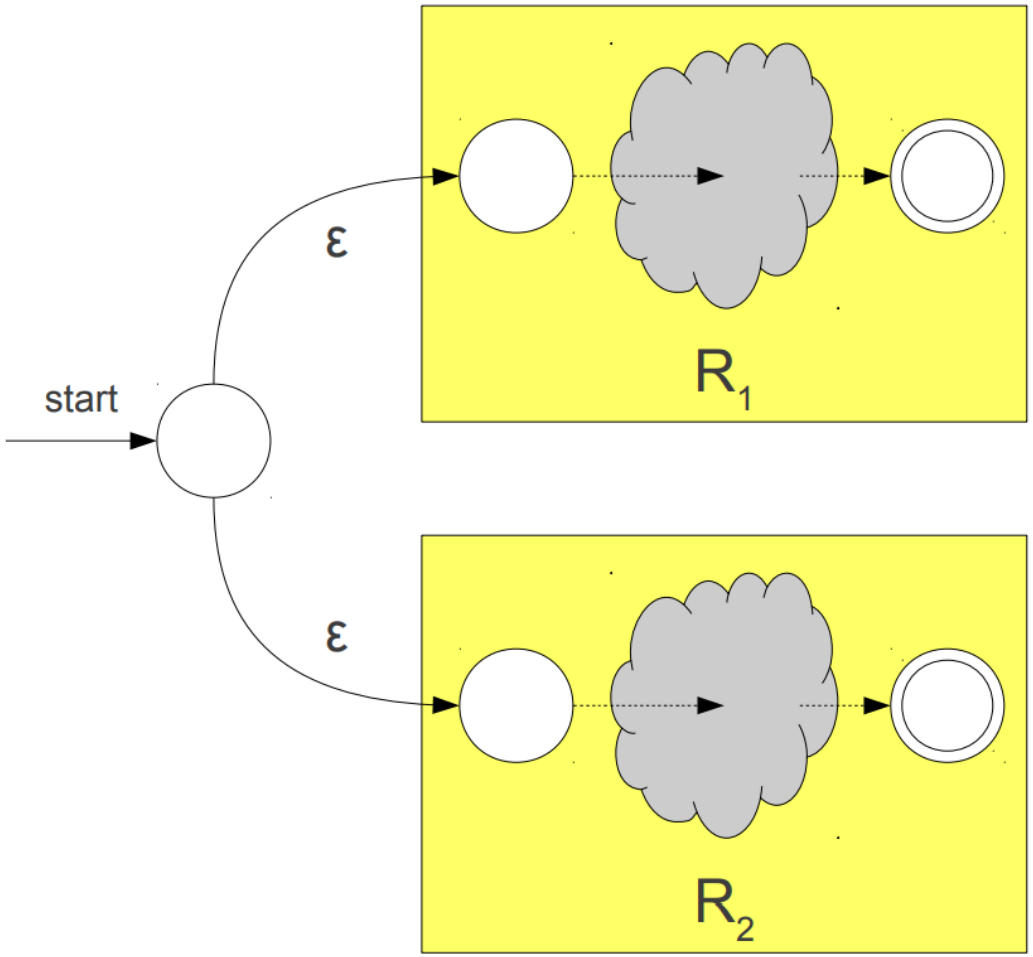
Construction for $R_1 \mid R_2$



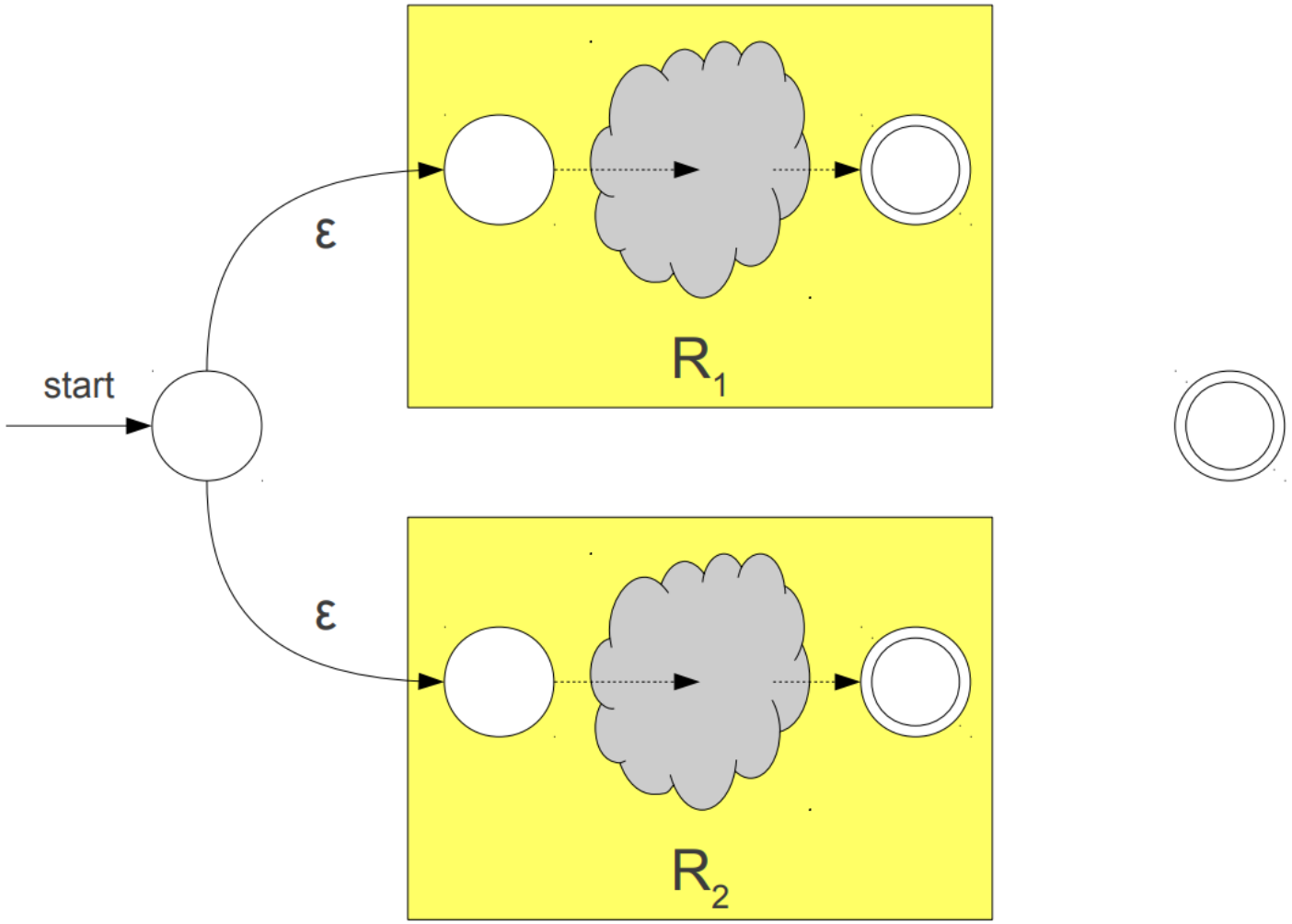
Construction for $R_1 \mid R_2$



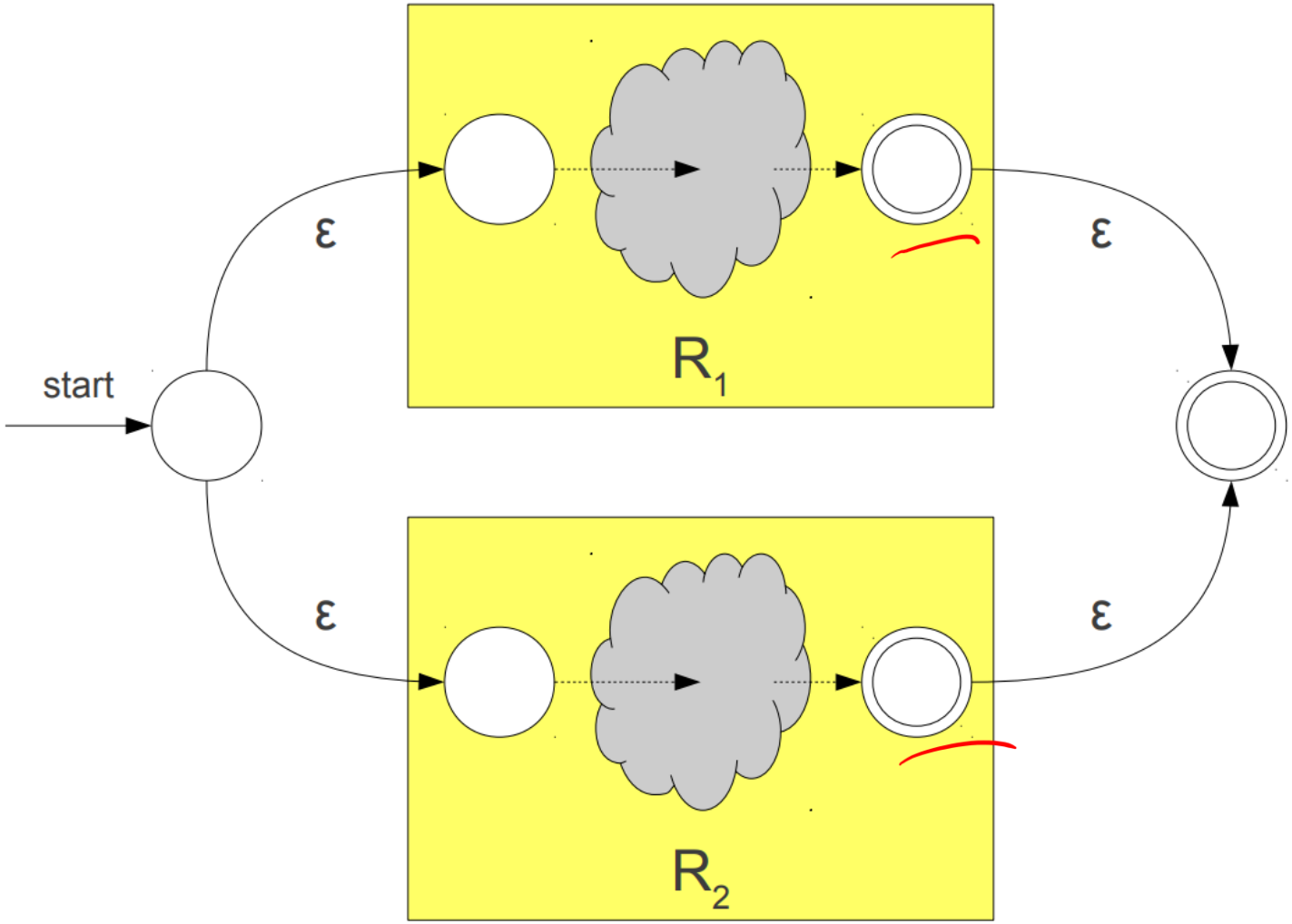
Construction for $R_1 \mid R_2$



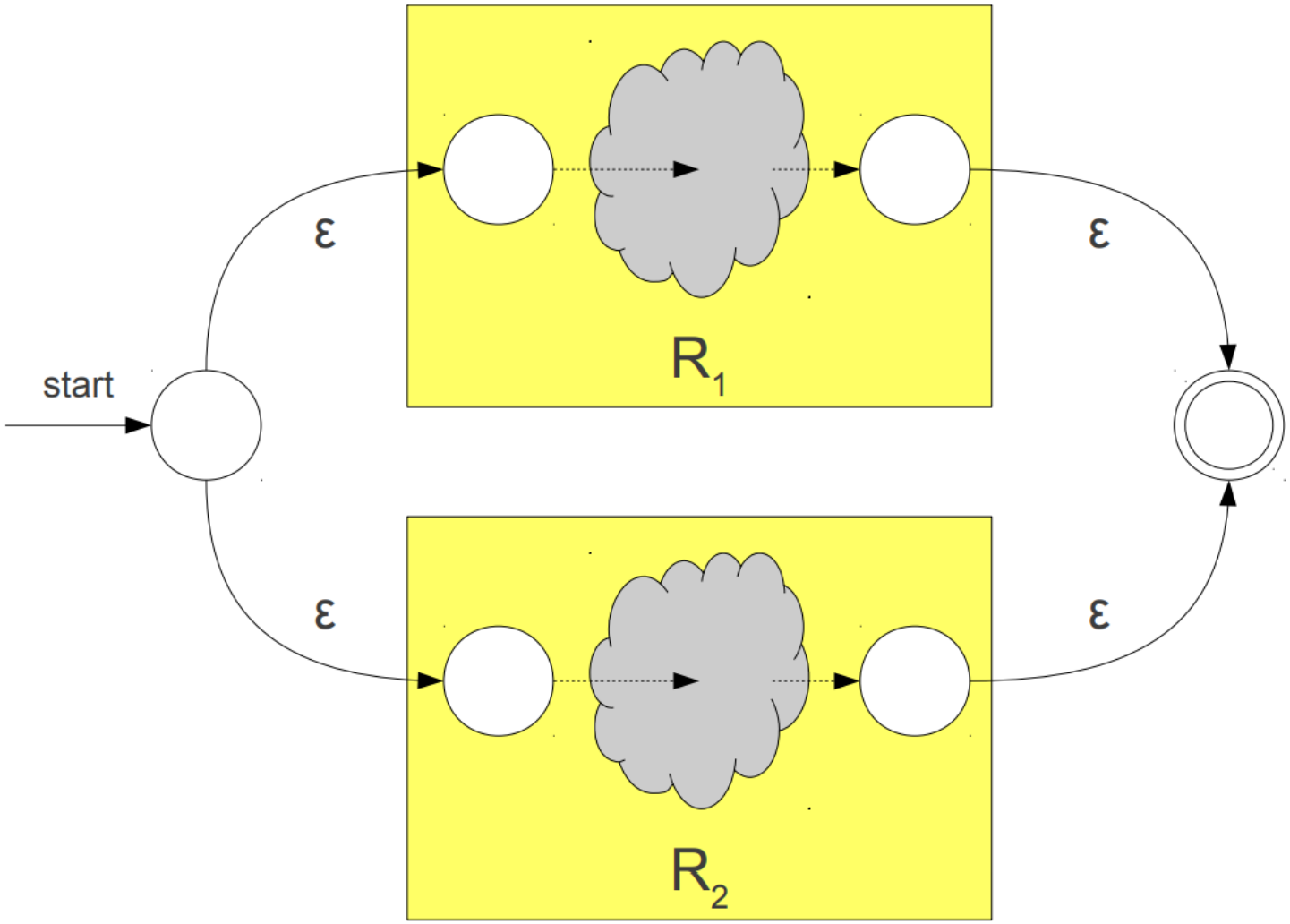
Construction for $R_1 \mid R_2$



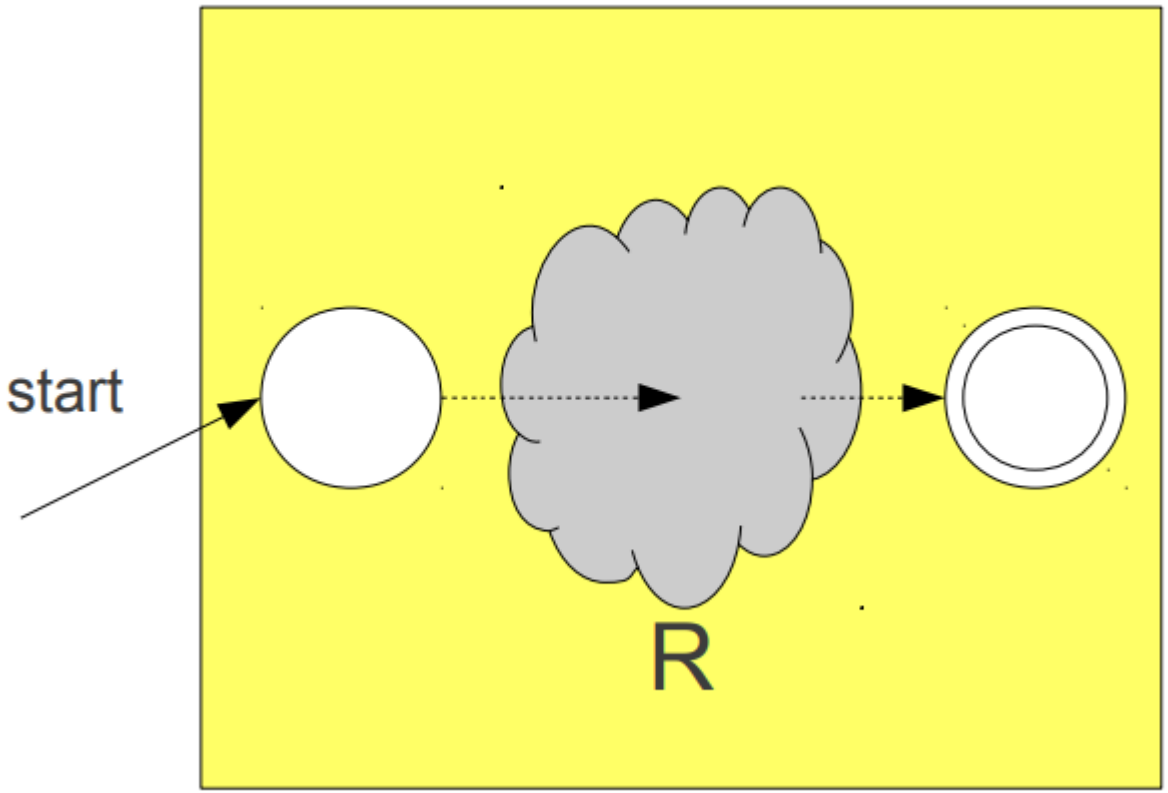
Construction for $R_1 \mid R_2$



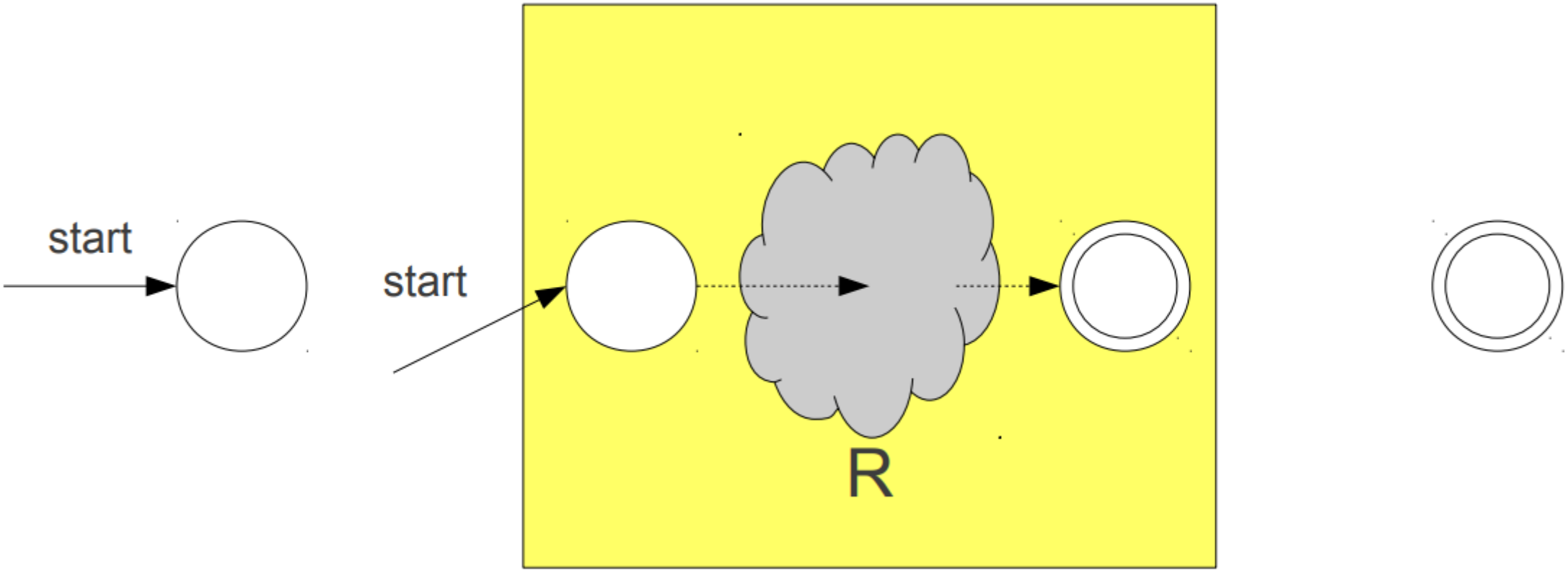
Construction for $R_1 \mid R_2$



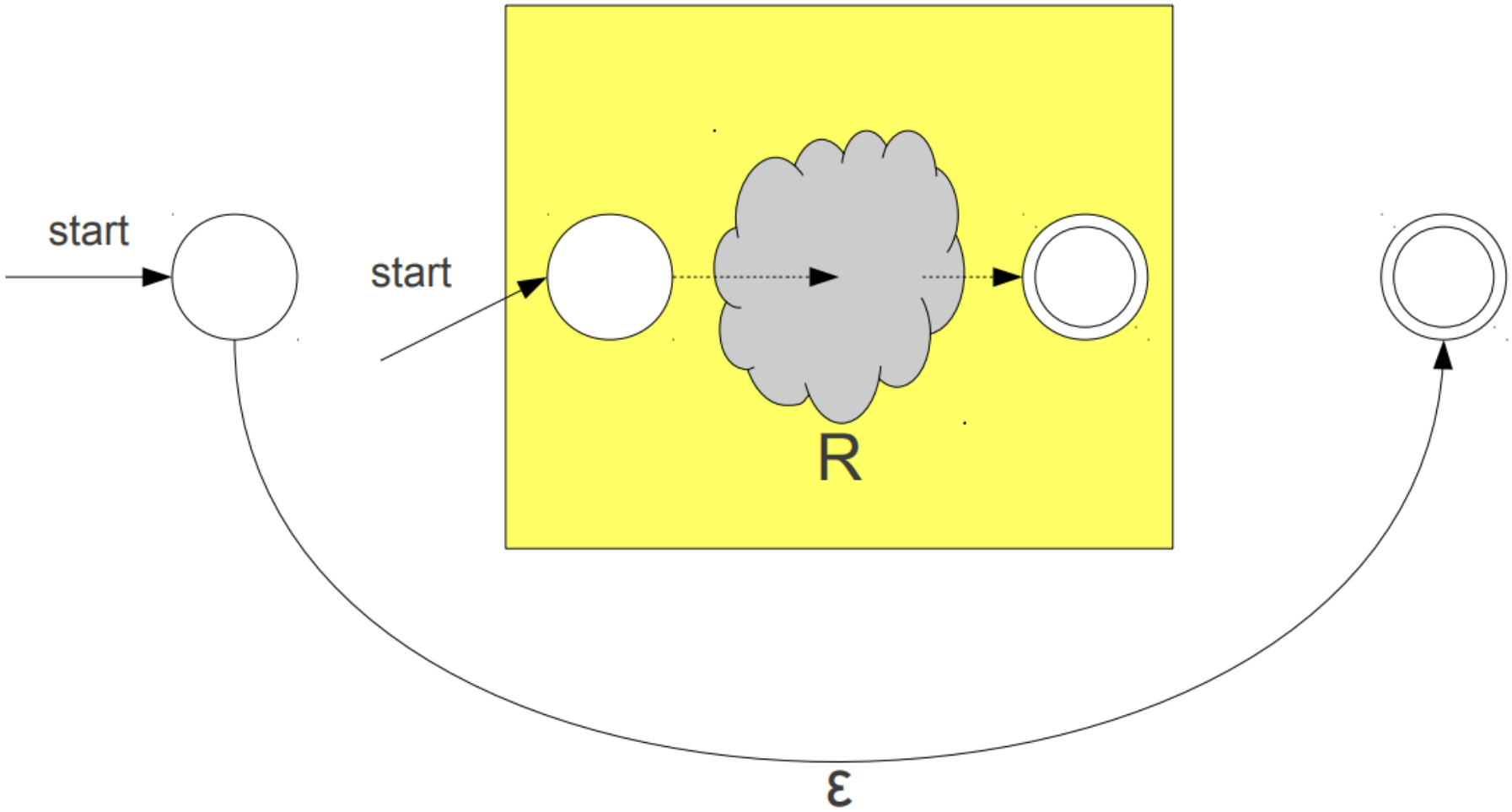
Construction for R*



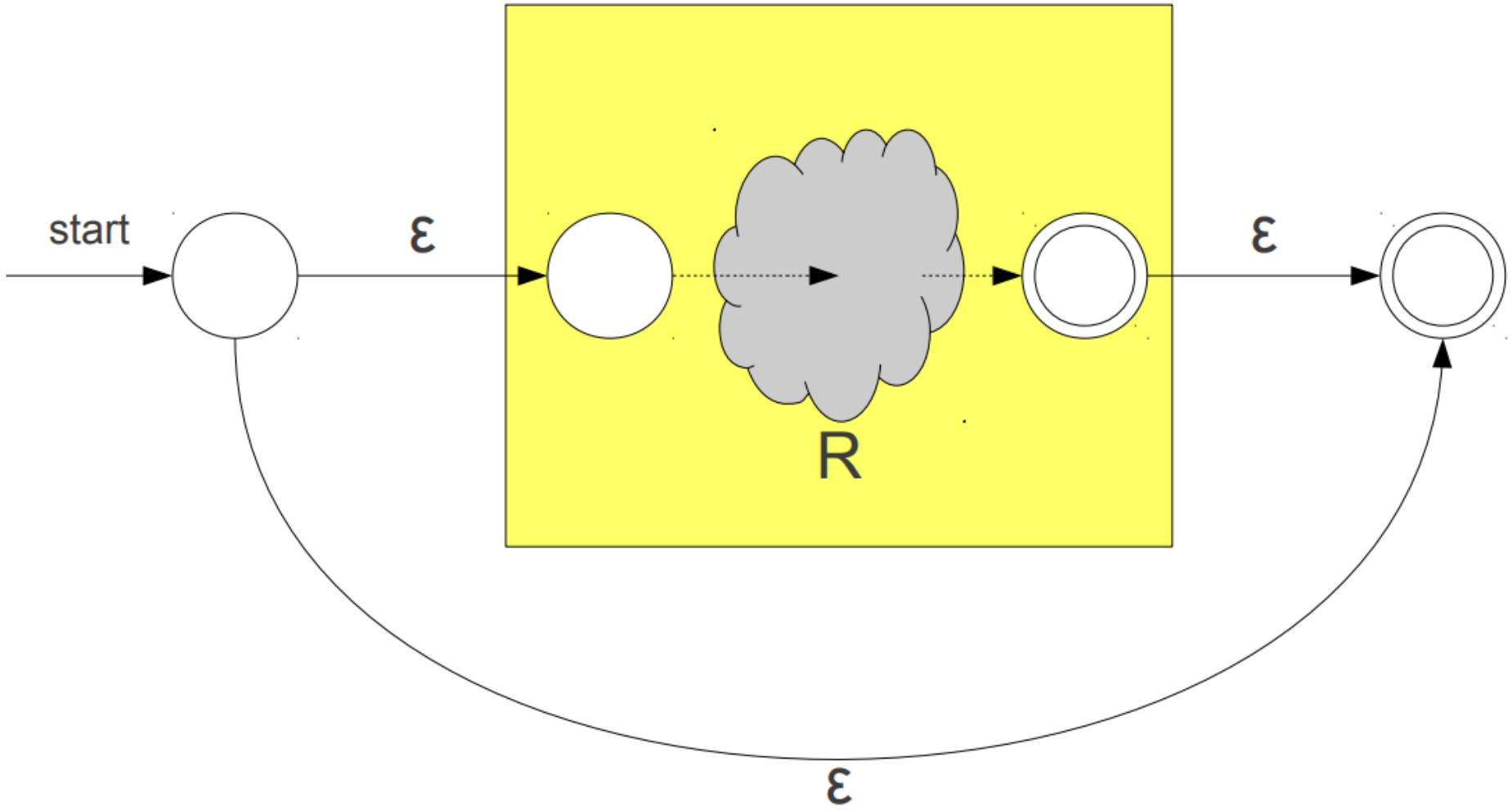
Construction for R*



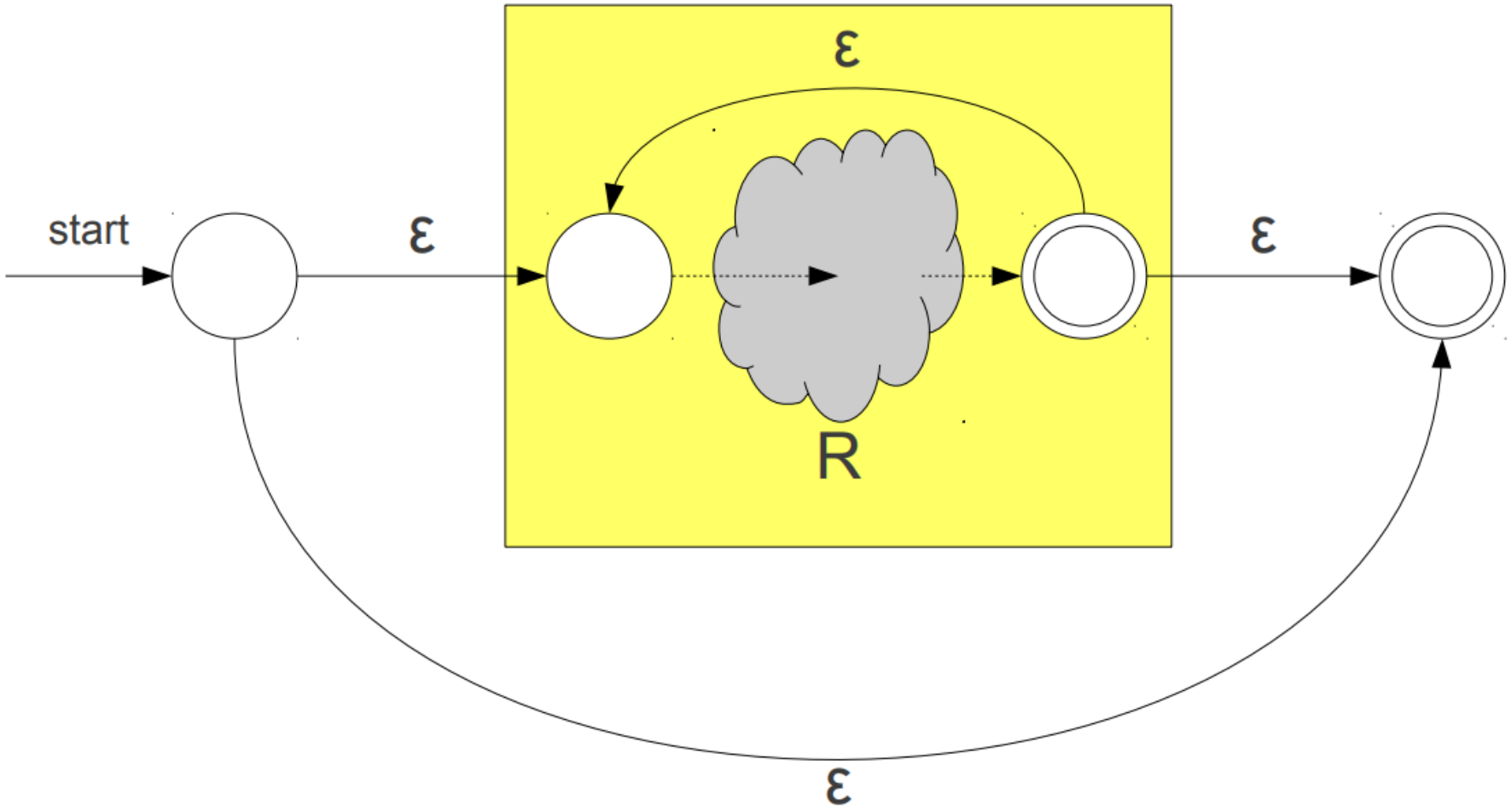
Construction for R*



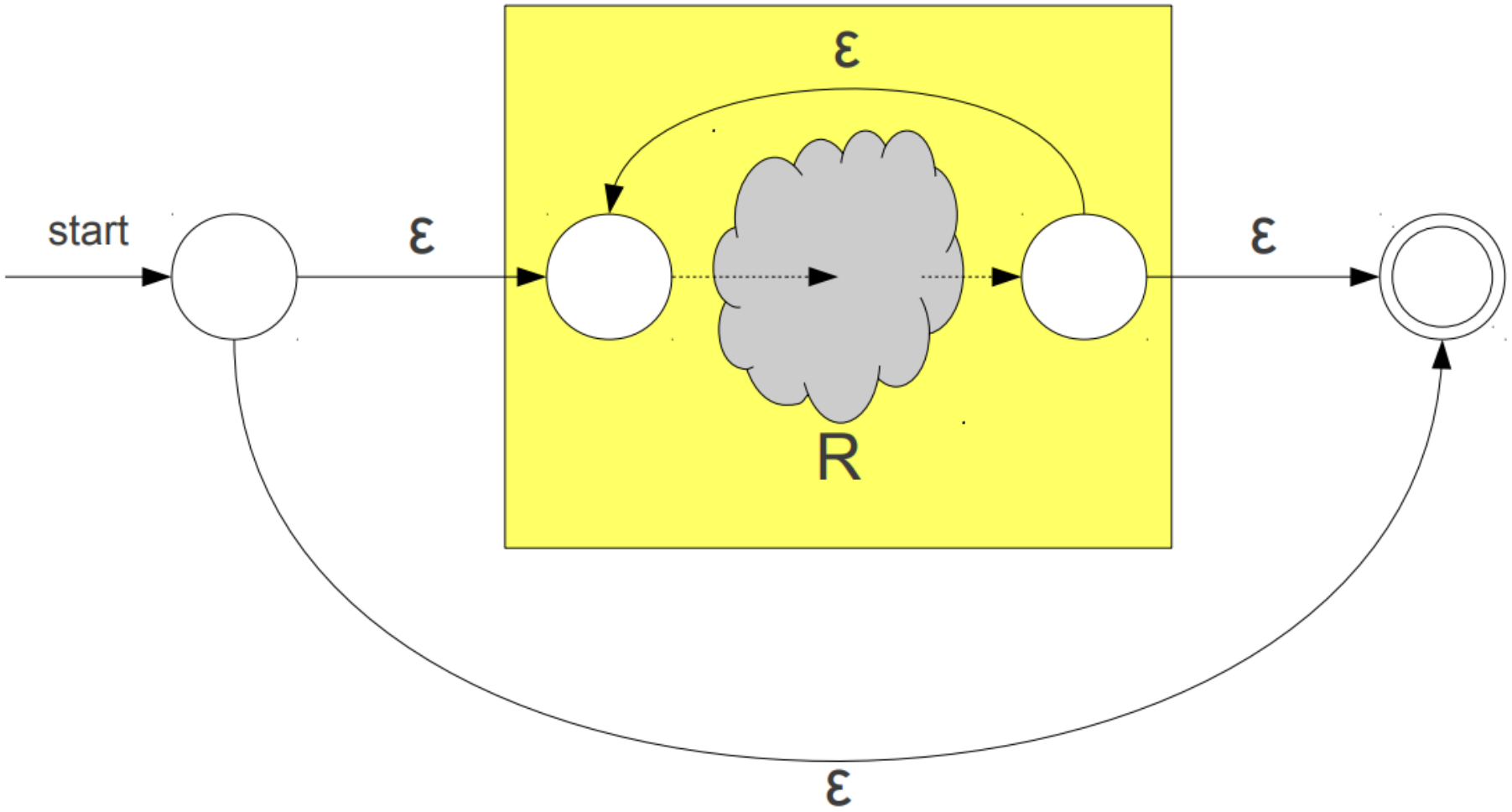
Construction for R*



Construction for R*



Construction for R*

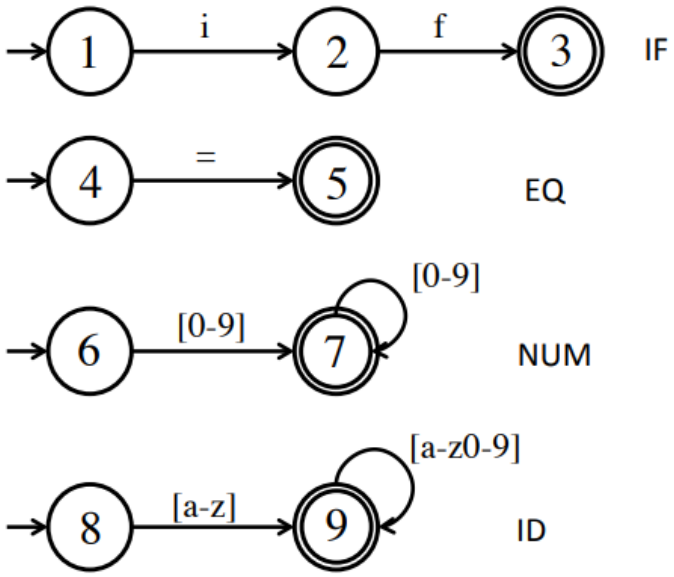


Lexical Analysis

PennState

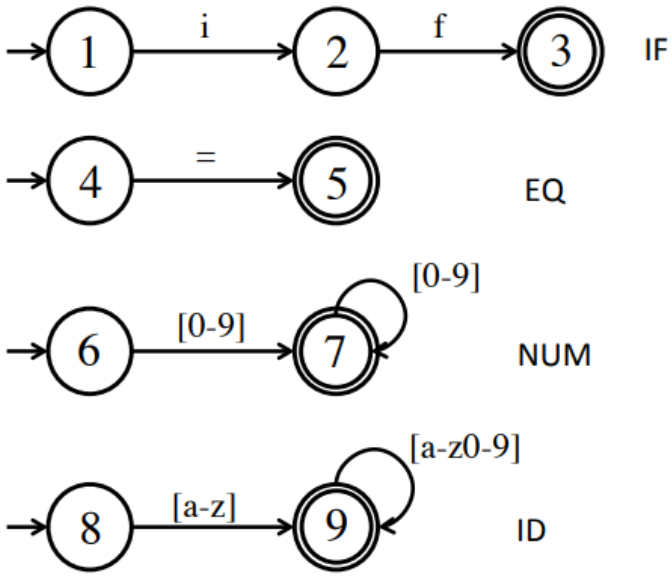
- What we have so far:
  - Regular expressions for each token
  - NFAs for each token that can recognize the corresponding lexemes
  - A way to simulate an NFA
- How to combine these to cut apart the input text and recognize tokens?
- Two ways:
  - Simulate all NFAs in turn (or in parallel) from the current position and output the token of the first one to get to an accepting state
  - Merge all NFAs into a single one with labels of the tokens on the accepting states

Illustration



- Four tokens: IF=if, ID=[a-z][a-z0-9]*, EQ='=', NUM=[0-9]+
- Lexical analysis of x = 60 yields:
  - <ID, x>, <EQ>, <NUM, 60>

Illustration: ambiguities



- Lexical analysis of $ifu26 = 60$ yields:
- Many splits are possible
  - <IF>, <ID, u26>, <EQ>, <NUM, 60>
  - <ID, ifu26>, <EQ>, <NUM, 60>
  - <ID, ifu>, <NUM, 26>, <EQ>, <NUM, 6>, <NUM, 0>

Conflict Resolutions

- Principle of the longest matching prefix: we choose the longest of the input that matches any token
- Following this principle, $\mathrm{ifu26} = 60$ will be split into
  - <ID, ifu26>, <EQ>, <NUM, 60>
- How to implement?
  - Run all NFAs in parallel, keeping track of the last accepting state reached by any of the NFAs
  - When all automata get stuck, report the last match and restart the search at that point
- Requires to retain the characters read since the last match to re-insert them on the input
  - In our example, '=' would be read and then re-inserted in the buffer

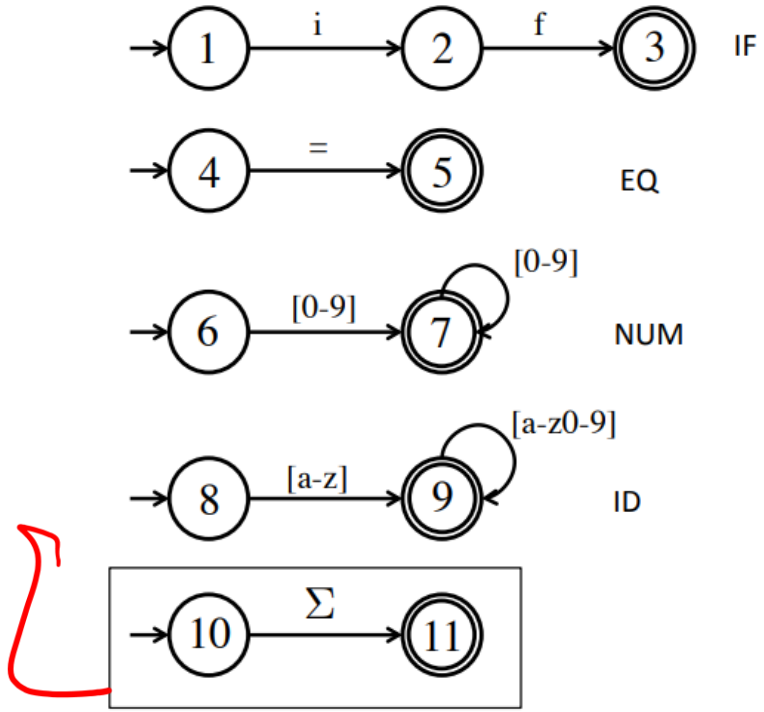
Other Source of Ambiguity

PennState

- A lexeme can be accepted by two NFAs
  - Example: keywords are often also identifiers (if in the example)
- Two Solutions:
  - Report an error (such conflict is not allowed in the language)
  - Let the user decide on a priority order on the tokens (eg., keywords have priority over identifiers)

What if nothing matches

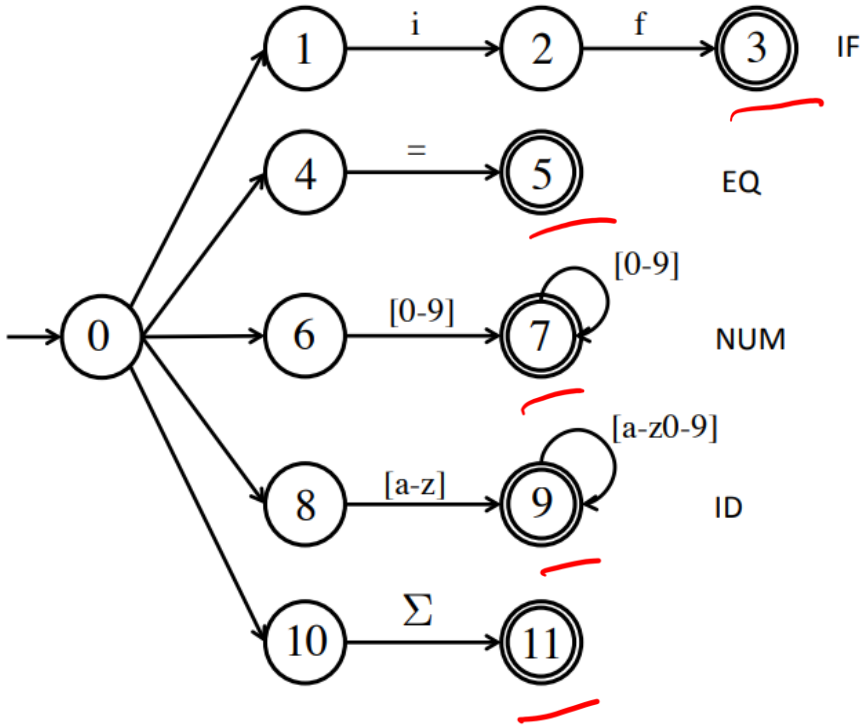
- What if we can not reach any accepting states given the current input?
- Add a “catch-all” rule that matches any character and reports an error



#260

Merging all automata into a single NFA

- In practice, all NFAs are merged and simulated as a single NFA
- Accepting states are labeled with the token name



Top Hat

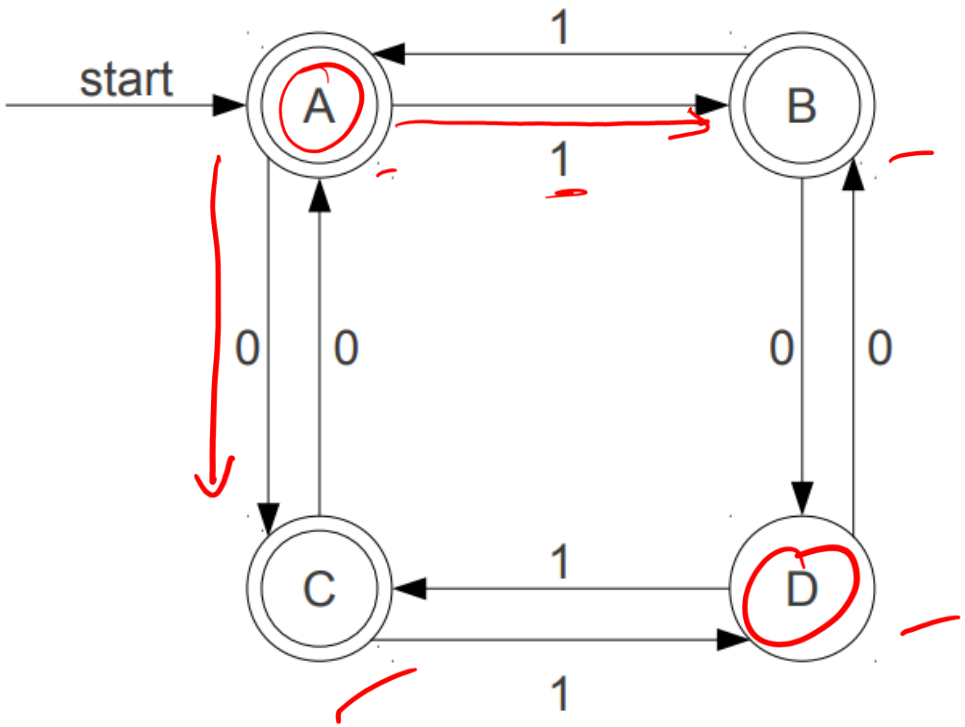
PennState

DFAs

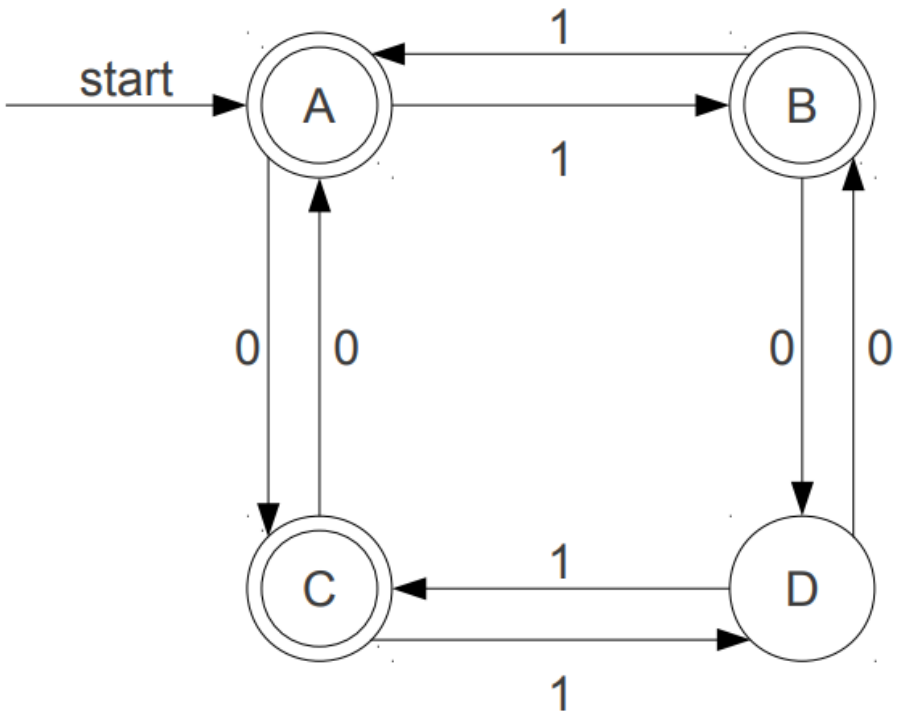
PennState

- The automata we've seen so far have all been NFAs
- It is possible to reduce complexity of matching to by transforming the NFA into an equivalent deterministic finite automata (DFA)
- DFA:
  - Transitions based on ε are not allowed
  - Each state has at most one outgoing transition defined for every letter

A Sample DFA



A Sample DFA



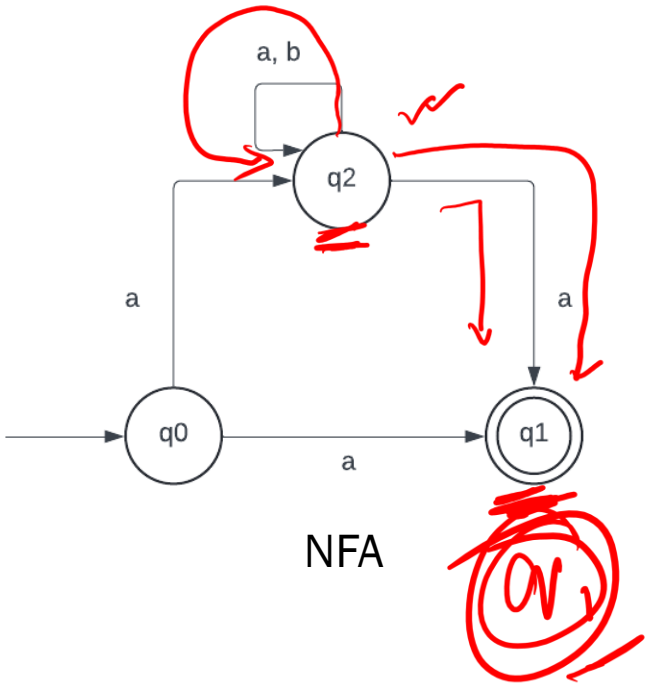
	0	1
A	C	B
B	D	A
C	A	D
D	B	C

Subset Construction

PennState

- NFAs can be in many states at once, while DFAs can only be in a single state at a time
- Key idea: Make the DFA simulate the NFA
- Have the states of the DFA correspond to the sets of states of the NFA
- Transitions between states of DFA correspond to transitions between sets of states in the NFA

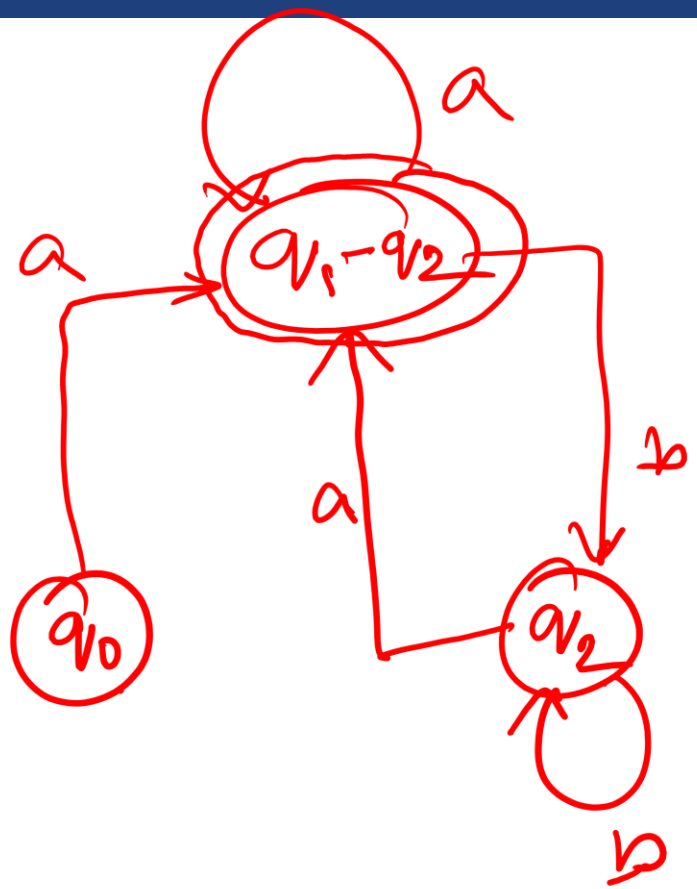
From NFA to DFA



NFA

NFA table		
state	a	b
q0	q_1, q_2	–
q1	–	–
q2	q_1, q_2	q_2

DFA table		
state	a	b
q_0	$q_1 - q_2$	–
$q_1 - q_2$	$q_1 - q_2$	q_2
q_2	$q_1 - q_2$	q_2



Reading and Exercises

PennState

Reading

- Chapter: 2.2 (Michael Scott Book)

Exercises

- Exercises: 2.2, 2.4, and 2.5 (Michael Scott Book)

References

Lecture Materials of CS 143, Stanford University