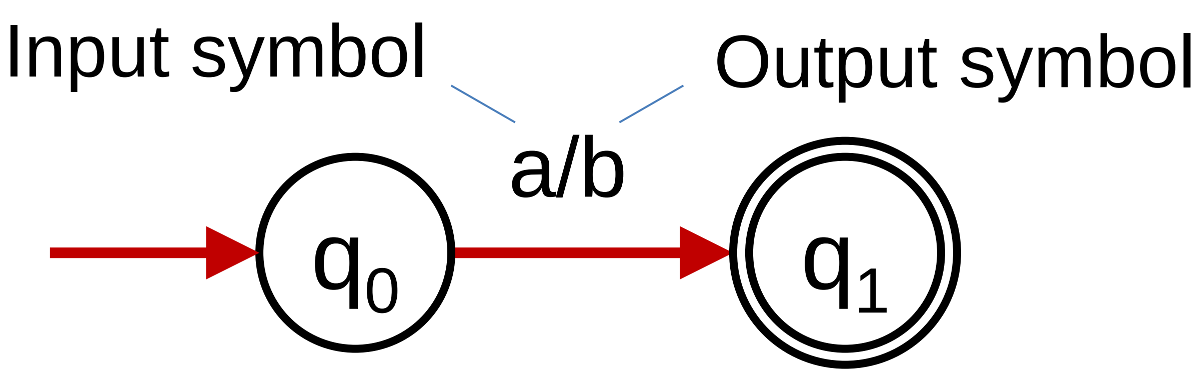
Mathematical Concepts (G6012)

Computing Machines III

Thomas Nowotny
Chichester I, Room CI-105
Office hours: Tuesdays 15:00-16:45
T.Nowotny@sussex.ac.uk

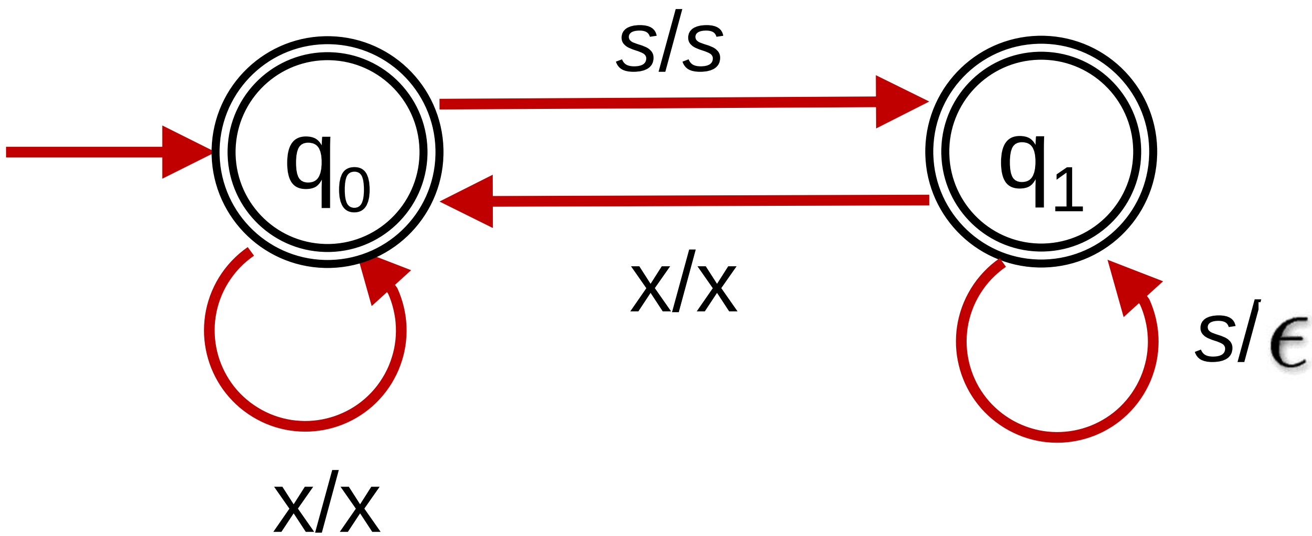
Last time

- We saw that deterministic FSA and non-deterministic FSA are equivalent
- We introduced Finite State Transducers: Addition of an output tape



BB Another example

- Let's build a whitespace remover (remove any multiple space characters)



x used here for anything that is not *s*.

Today: Pushdown Automata

- What FSA cannot do is processing nested structures:
  - In programming languages: if-then-else, procedure calls, …
  - In natural languages: phrases embedded in others
  - Generally: We need the power to process strings of brackets, e.g. ([{}()])

Processing nested structures

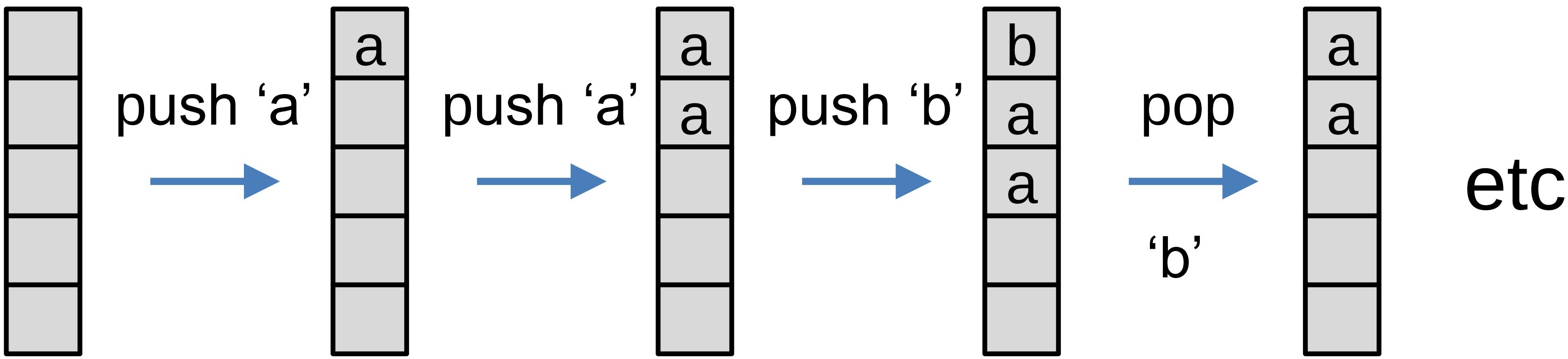
- Memory requirements:
  - Unlimited unless we limit the depth of nesting
  - Need to keep track of the order in which brackets must be closed
  - We have a linear structure (sequence) of symbols
  - Most recently recorded memories (opening brackets) must be accessed first

Pushdown

- A special kind of list
- Provides (in principle) unbound storage
- Last in, first out (LIFO)
- Add/remove items only from one end (“top”)
- Push – add an element to the top of PD
- Pop – remove and element from the top of PD
- No other editing or browsing allowed

Example

- It’s like a stack of paper where you stack stuff on the top and take it away from the top:



- Alternative notation:

$$(\ ) \xrightarrow{\text{push ‘a’}} (a) \xrightarrow{\text{push ‘a’}} (aa) \xrightarrow{\text{push ‘b’}} (baa) \xrightarrow[\text{‘b’}]{\text{pop}} (aa)$$

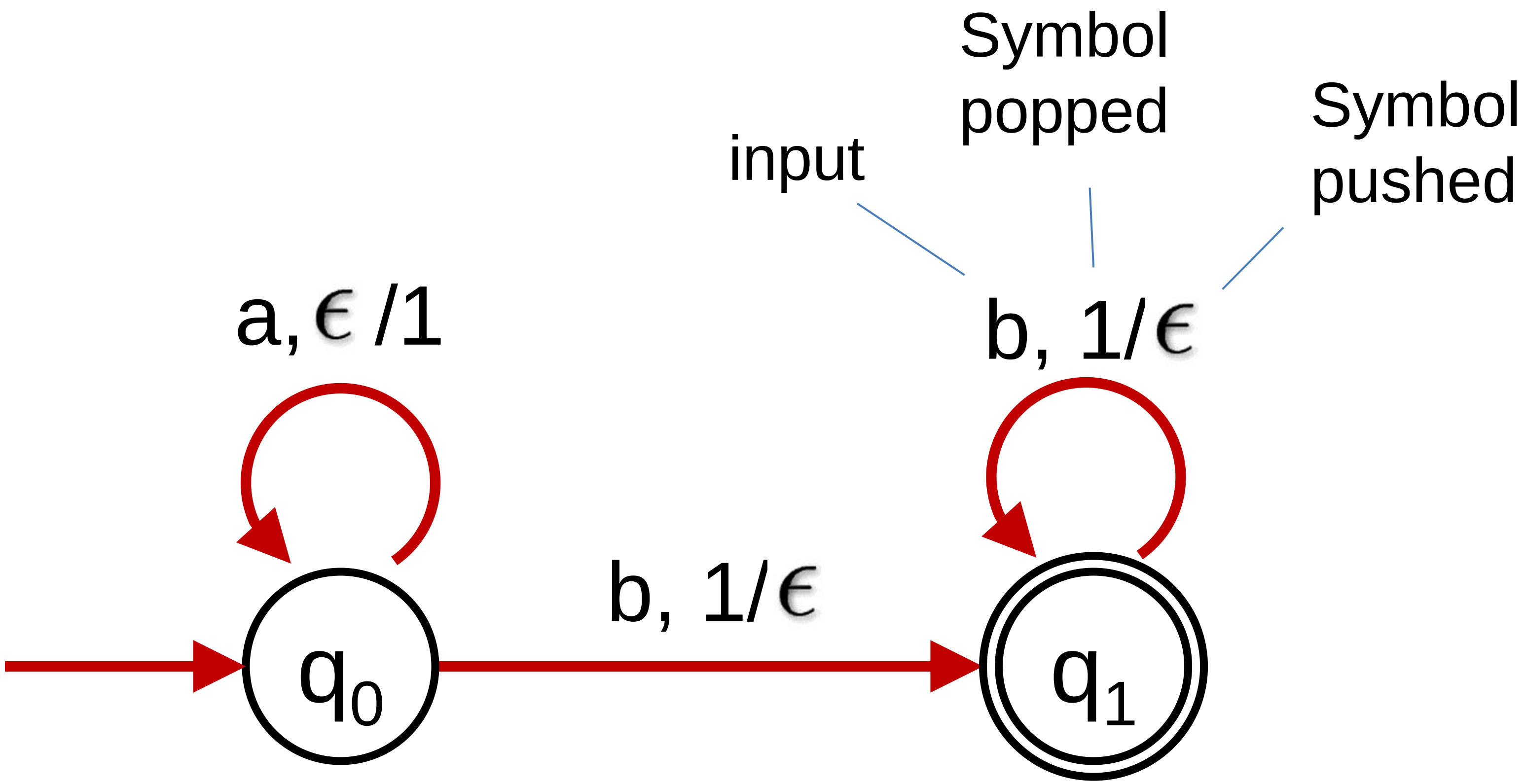
Pushdown Automata

- Result of adding a pushdown storage to an FSA
- This gives a more powerful language recogniser (see below)
- Provides enough power for programming language analysis (syntax analysis)
- May even be enough for natural language analysis

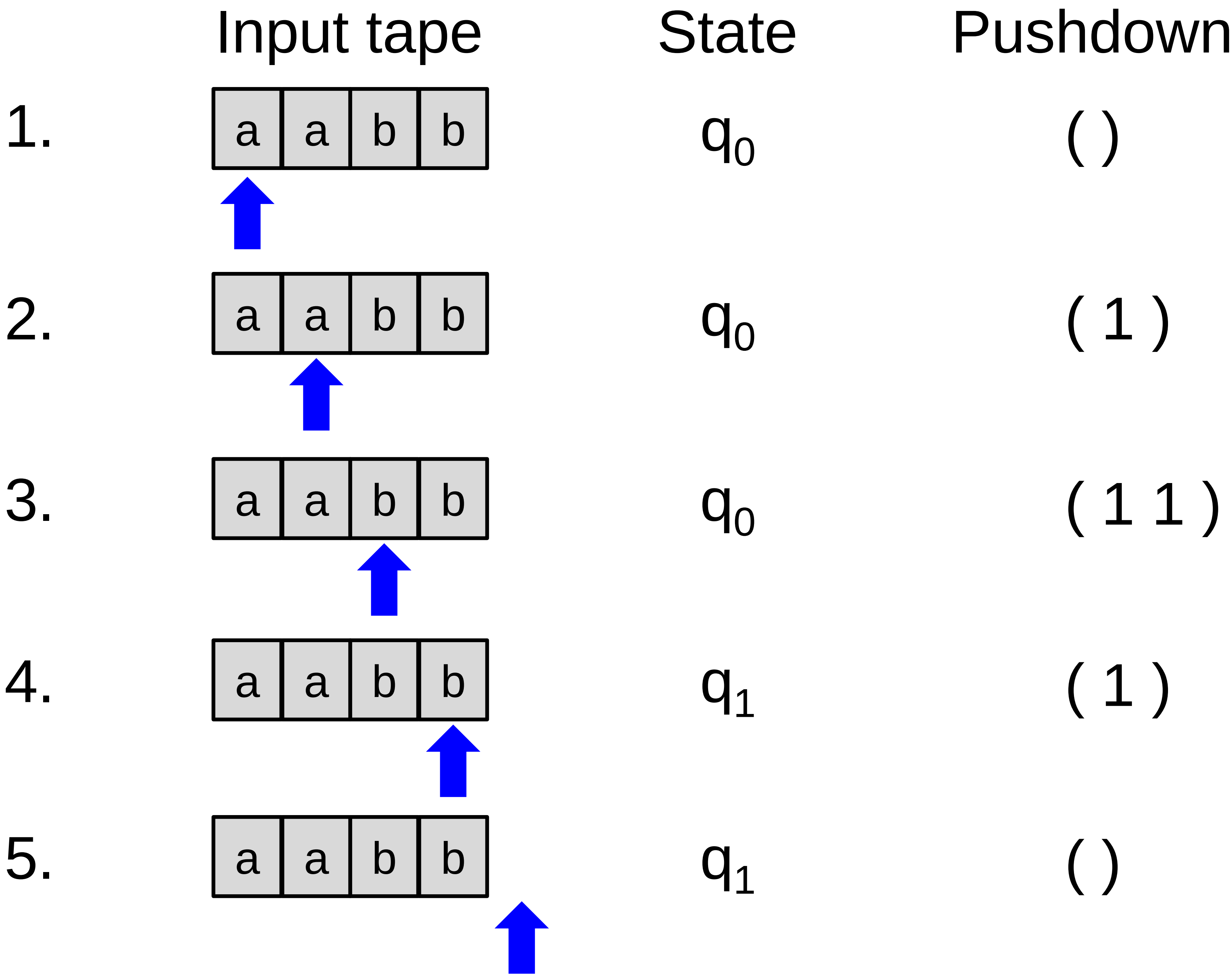
Pushdown Automaton: How does it work?

- Need to specify 3 things now:
  - Input symbol to be scanned
  - Symbol to be popped from pushdown
  - Symbol to be pushed onto pushdown
- Any of these can be the empty string ϵ

Example



What language does this accept?


Input tape
State
Pushdown
1.
a a b b
q0
( )
2.
a a b b
q0
(1)
3.
a a b b
q0
(1 1)
4.
a a b b
q1
(1)
5.
a a b b
q1
( )

- Accepts a string of a’s followed by the same number of b’s:

$$\{a^n b^n \mid n \geq 0\}$$

- This cannot be expressed with a regular expression!

Accepting computation

- Must be in a final state
- The input must have been consumed
- The pushdown must be empty

Error states

- No rule applies for input symbol (stuck)
- Cannot pop correct symbol (error)
(includes trying to pop a symbol other than ϵ from empty pushdown)

Alternative Notation for Computation

$(q_0, aabb, \epsilon) \quad \mapsto (q_0, abb, 1)$

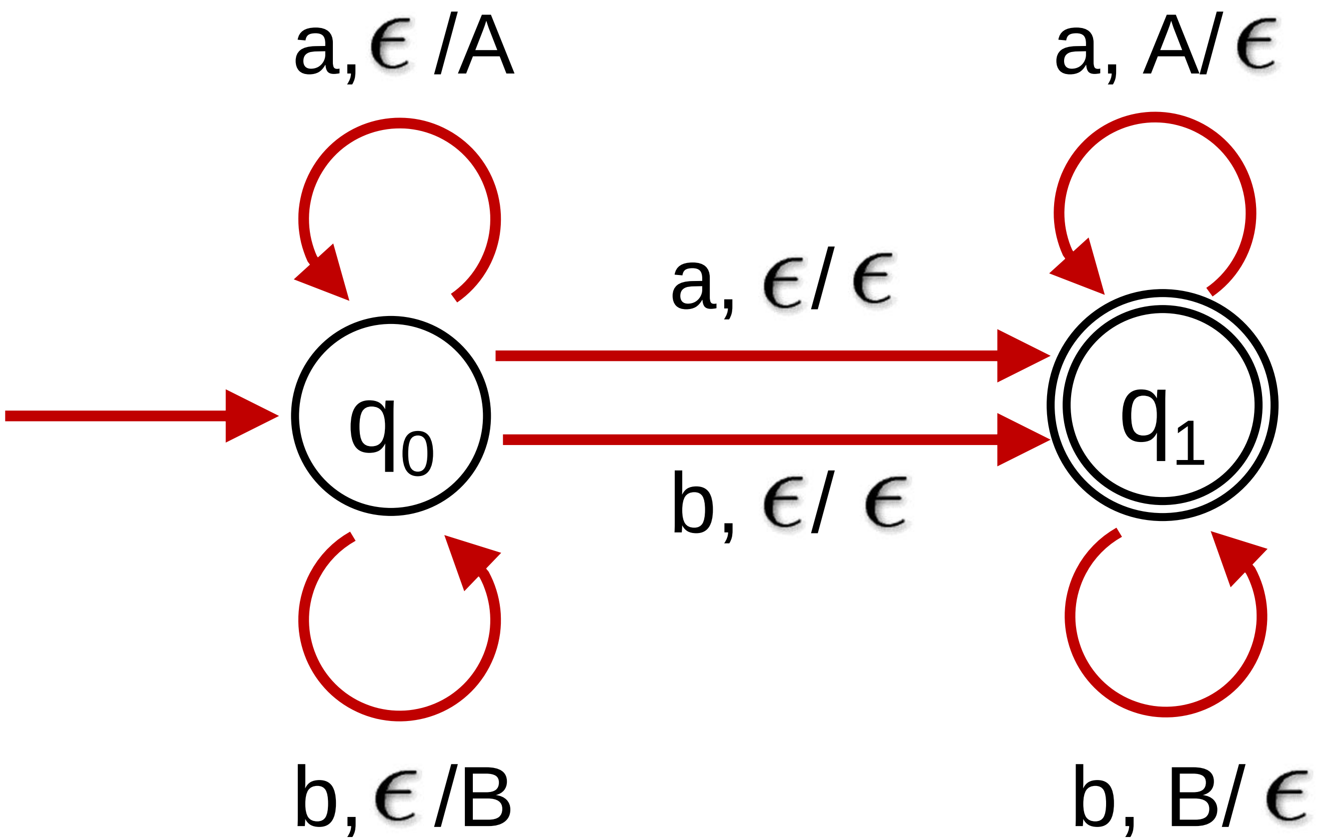
$\mapsto (q_0, bb, 11)$

$\mapsto (q_1, b, 1)$

$\mapsto (q_1, \epsilon, \epsilon)$

- Pushdown here represented as a string of pushdown symbols

Example



BB: A successful Computation

$(q_0, \text{babab}, \epsilon) \mapsto (q_0, \text{abab}, \text{B})$

$\mapsto (q_0, \text{bab}, \text{AB})$

$\mapsto (q_1, \text{ab}, \text{AB})$

$\mapsto (q_1, \text{b}, \text{B})$

$\mapsto (q_1, \epsilon, \epsilon)$

- This recognises the language:
  $\{w\{a,b\}w^R \mid w \in \{a,b\}^n, n \geq 0\}$

Non-deterministic – guessing midpoint

Wrongly guessed midpoint:

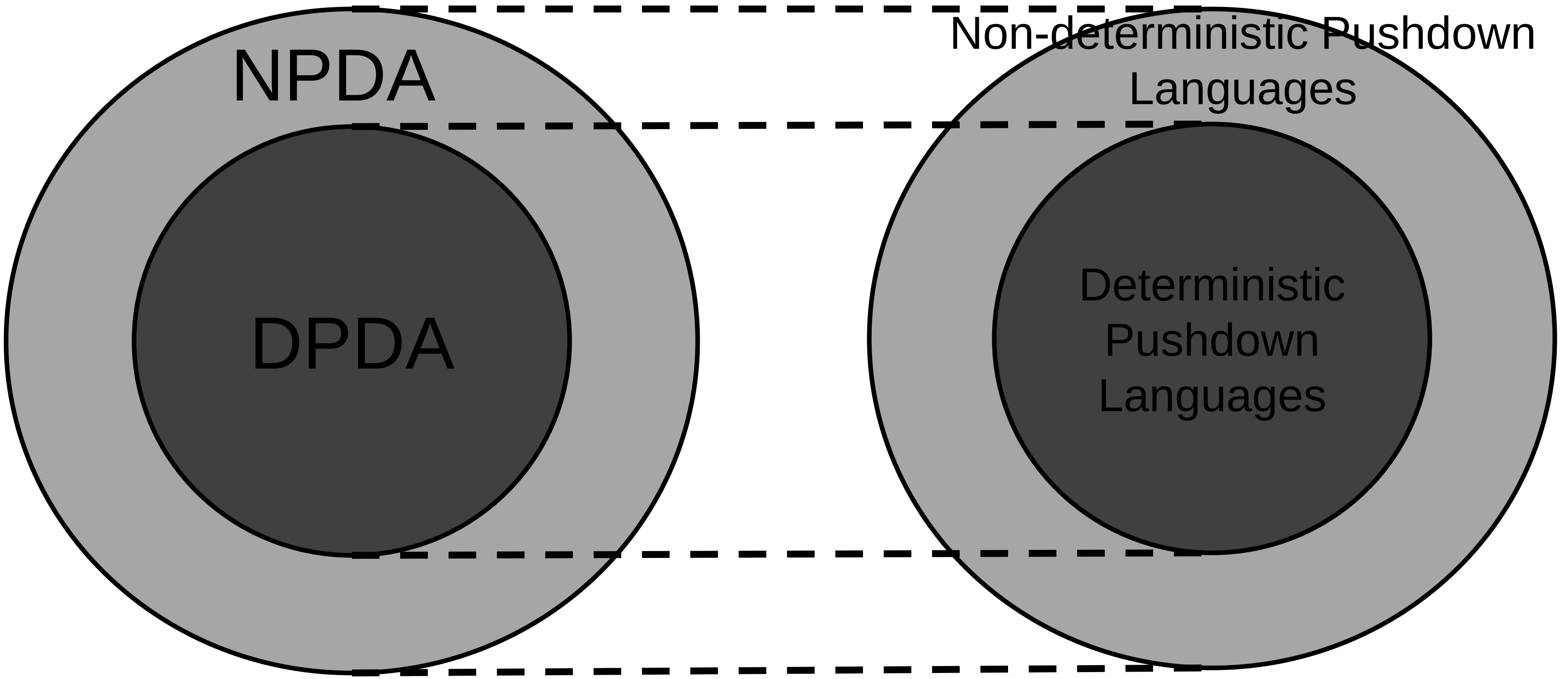
$$(q_0, babab, \epsilon) \mapsto (q_0, abab, B)$$
$$\mapsto (q_0, bab, AB)$$
$$\mapsto (q_0, ab, BAB)$$
$$\mapsto (q_1, b, BAB)$$
$$\mapsto (q_1, \epsilon, AB)$$

FAIL

As usual …

- Machine explores all possible choices
- Acceptance when one accepting computation path exists
- It turns out that here non-determinism does add power
- There are cases, where non-determinism cannot be removed (like guessing the midpoint)

Non-deterministic PDA (NPDA) and deterministic PDA (DPDA) accept the a different family of languages



Properties of Pushdown Automata

- Family of languages:
  - PDA accept the same family of languages as can be expressed by **Context Free Grammars**
  - In other words they accept exactly the **Context Free Languages**
  - Context Free Grammars are used to describe (define) programming language syntax
  - (Also equivalent to BNF and syntax charts)

Context-Free Grammar

- Is define by “productions” or “production rules”:

  1. $S \mapsto aSb$
  2. $S \mapsto ab$

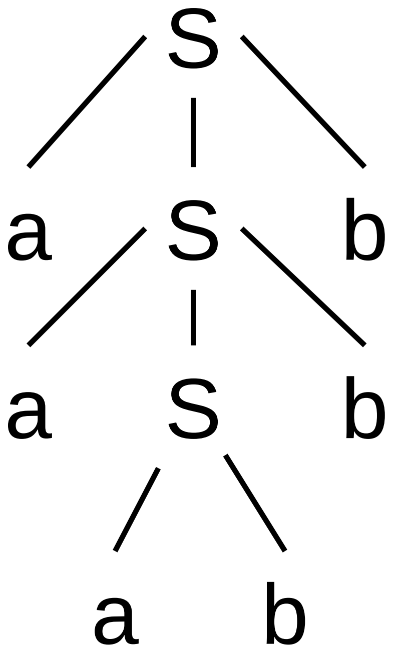
- Are applied repeatedly, e.g

  $$S \underset{1.}{\mapsto} aSb \underset{1.}{\mapsto} aaSbb \underset{2.}{\mapsto} aaabbb$$

- Generates a’s followed by same number of b’s

Derivation tree

- Generating a word can be visualised as a tree:



Other example

- Productions:

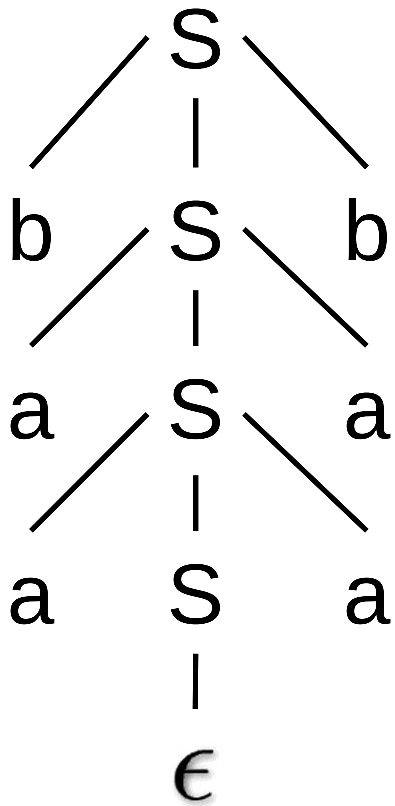
  S ↦ aSa

  S ↦ bSb

  S ↦ ϵ

- Generates the palindrome language $\{SS^R \mid S \in \{a,b\}^n, n \geq 0\}$ where the ^{R} denotes the reverse of the string

Derivation tree example



Limits of Power

- Can be achieved:
  - Language of palindroms
  - Counting two symbols
  - Programming languages (deterministically)
  - Natural Languages?
- What can't be done:
  - Copy language $\{SS \,|\, S \in \{a,b\}^n, n \geq 0\}$
  - Counting symbols beyond 2
  - (Crossing dependencies)

Performance consideration

- When syntax-checking programs,
  - PDA based checking can take $O(n^3)$ time
  - This can be very slow for large programs
  - However, if the PDA is deterministic, time is only $O(n)$

Stacks vs Pushdowns

- Most people would not make a distinction
- If a distinction is made
  - Pushdown strictly push-pop
  - Stack can be inspected read-only
- Stack automata are a more powerful but little known type of machine

TURING MACHINES

Turing Machines (TM)

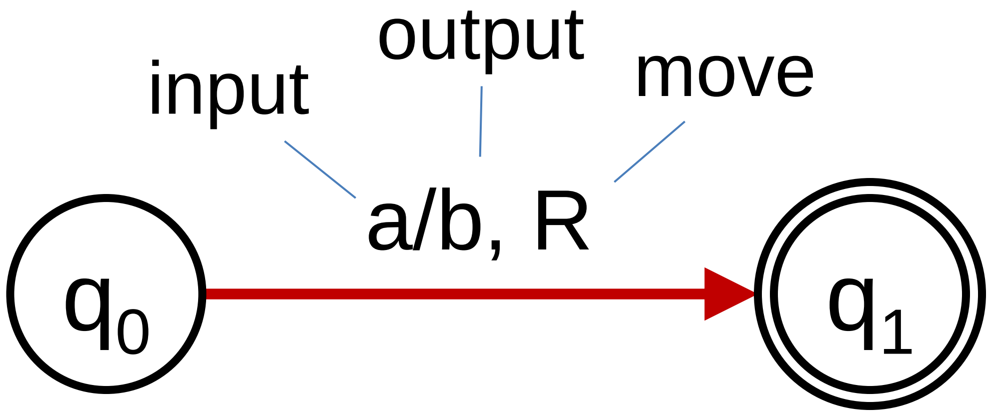
- Are a very simple extension to finite state machines
- The main change is to allow **editing the input tape**
- No limit on the size of the tape
- Tape 2-way infinite (like the integers)

…	a	a	B	b	a	a	*	B	…

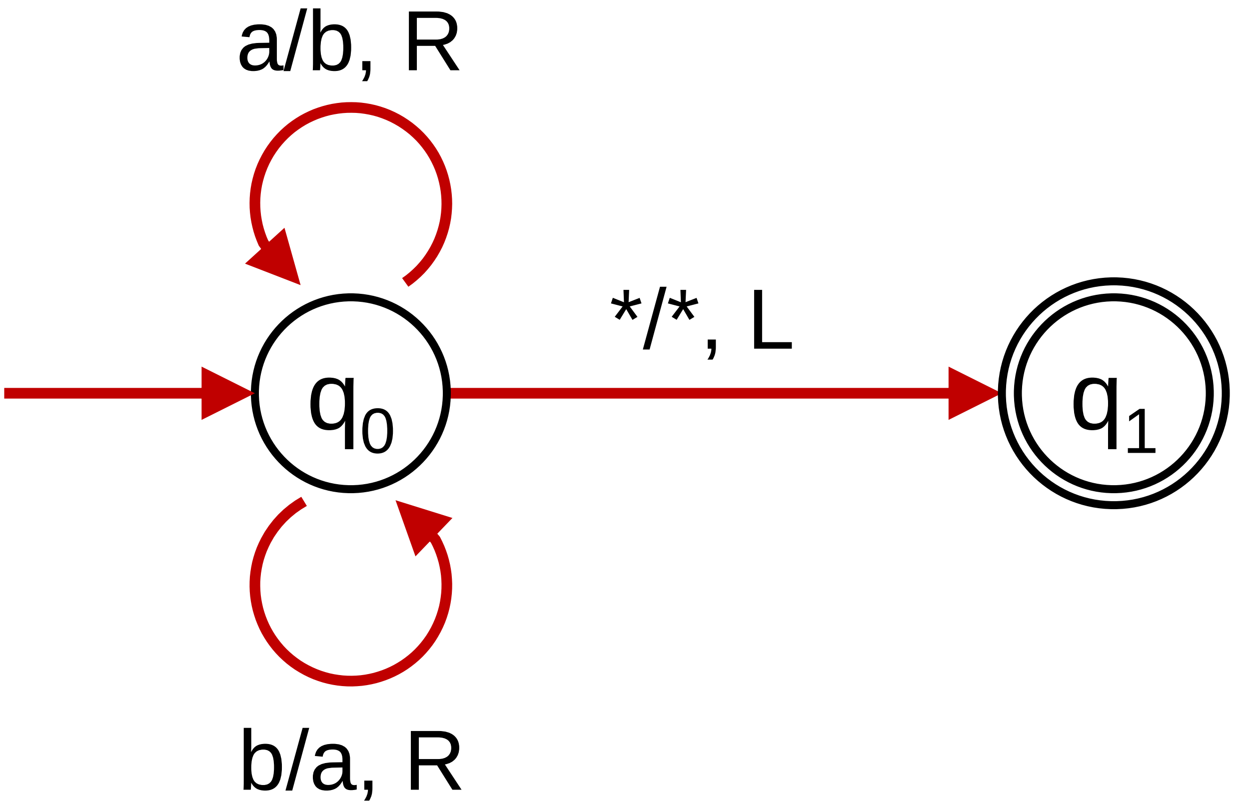
- (We will use the symbol B for blank positions)

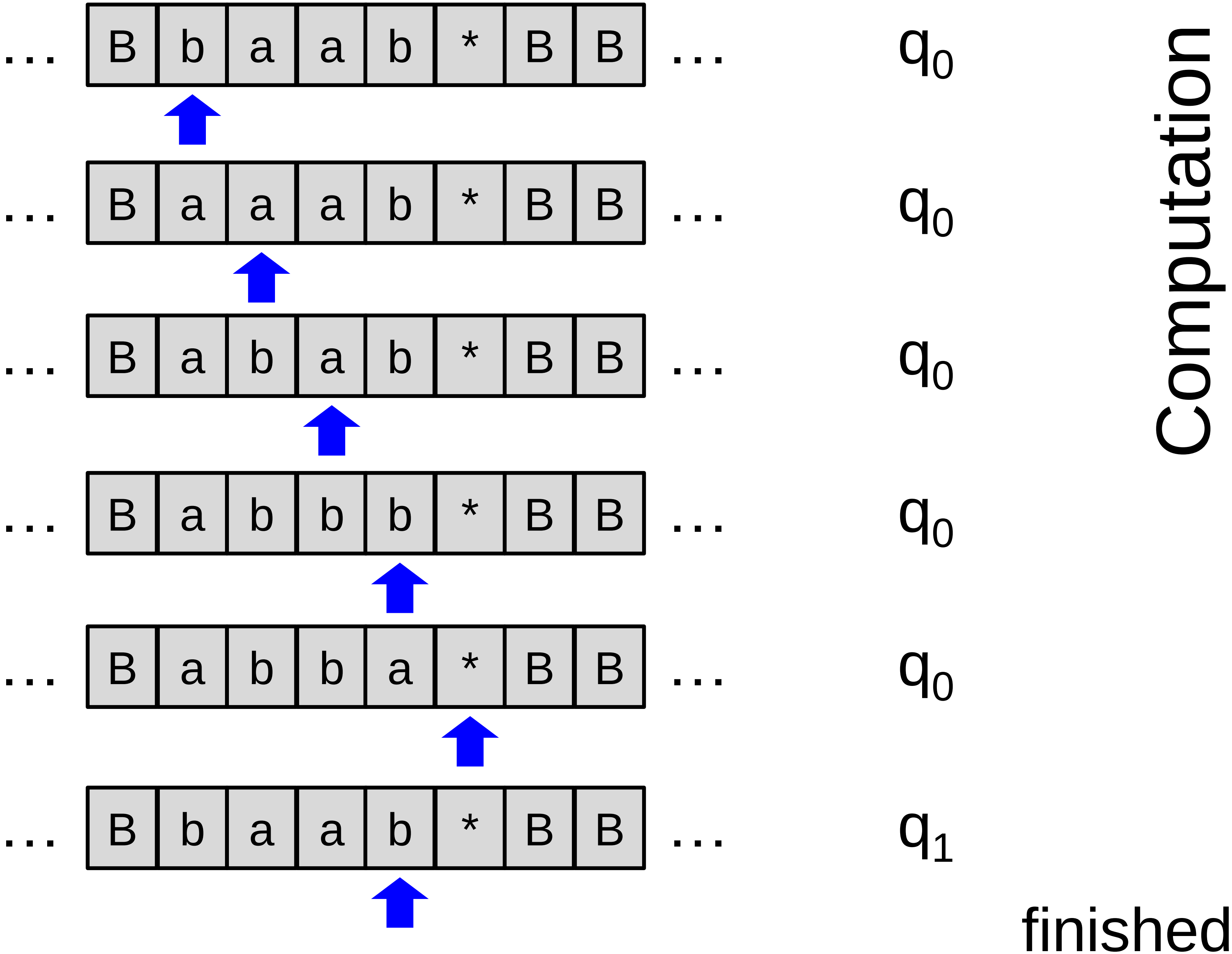
Transitions in TM

- Current state
- New state
- Symbol currently read
- New symbol to replace the read symbol
- Direction to move the tape head (left (L), right (R), stay (S))



Example



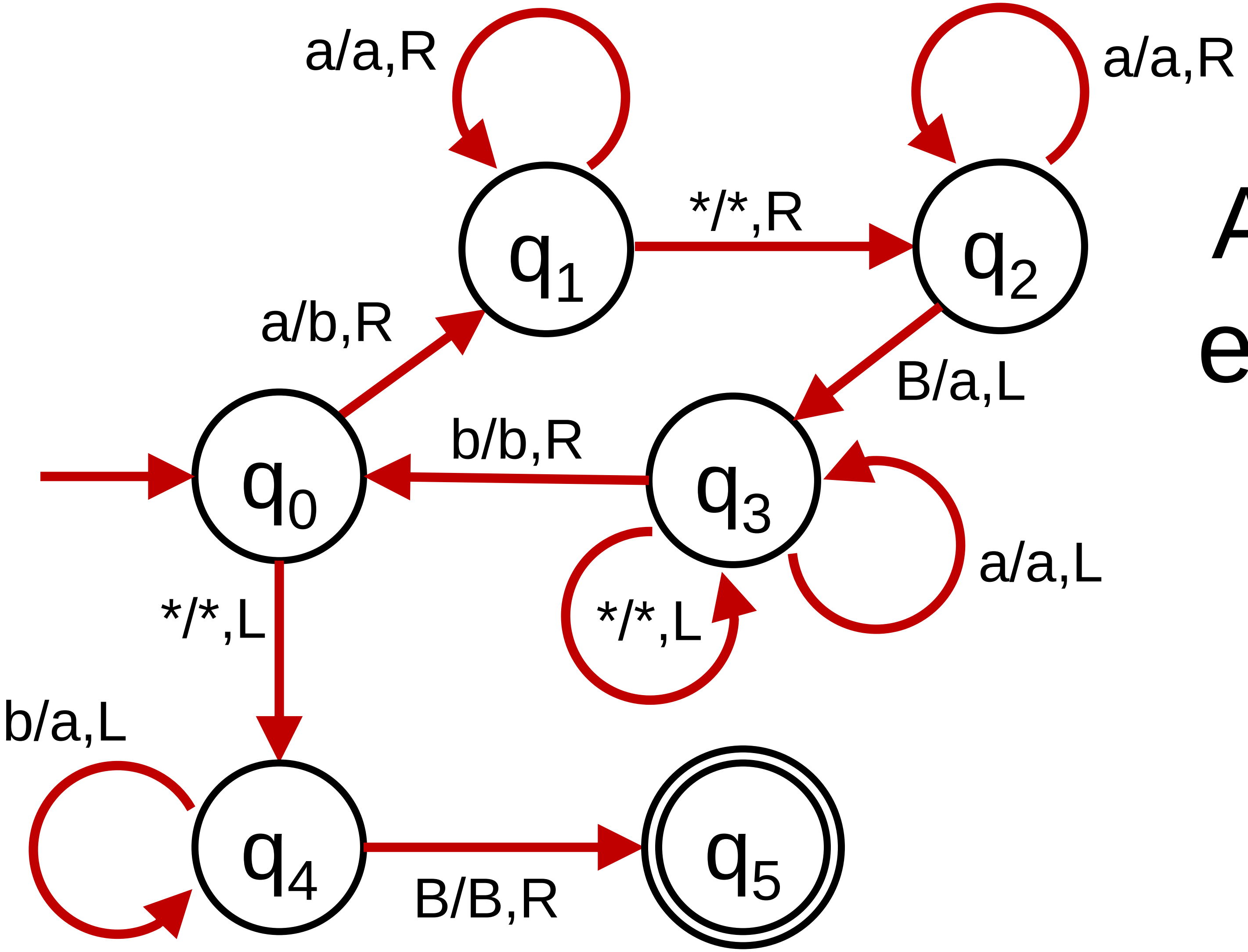


… B b a a b * B B … q_0
… B a a a b * B B … q_0
… B a b a b * B B … q_0
… B a b b b * B B … q_0
… B a b b a * B B … q_0
… B b a a b * B B … q_1
Computation
finished

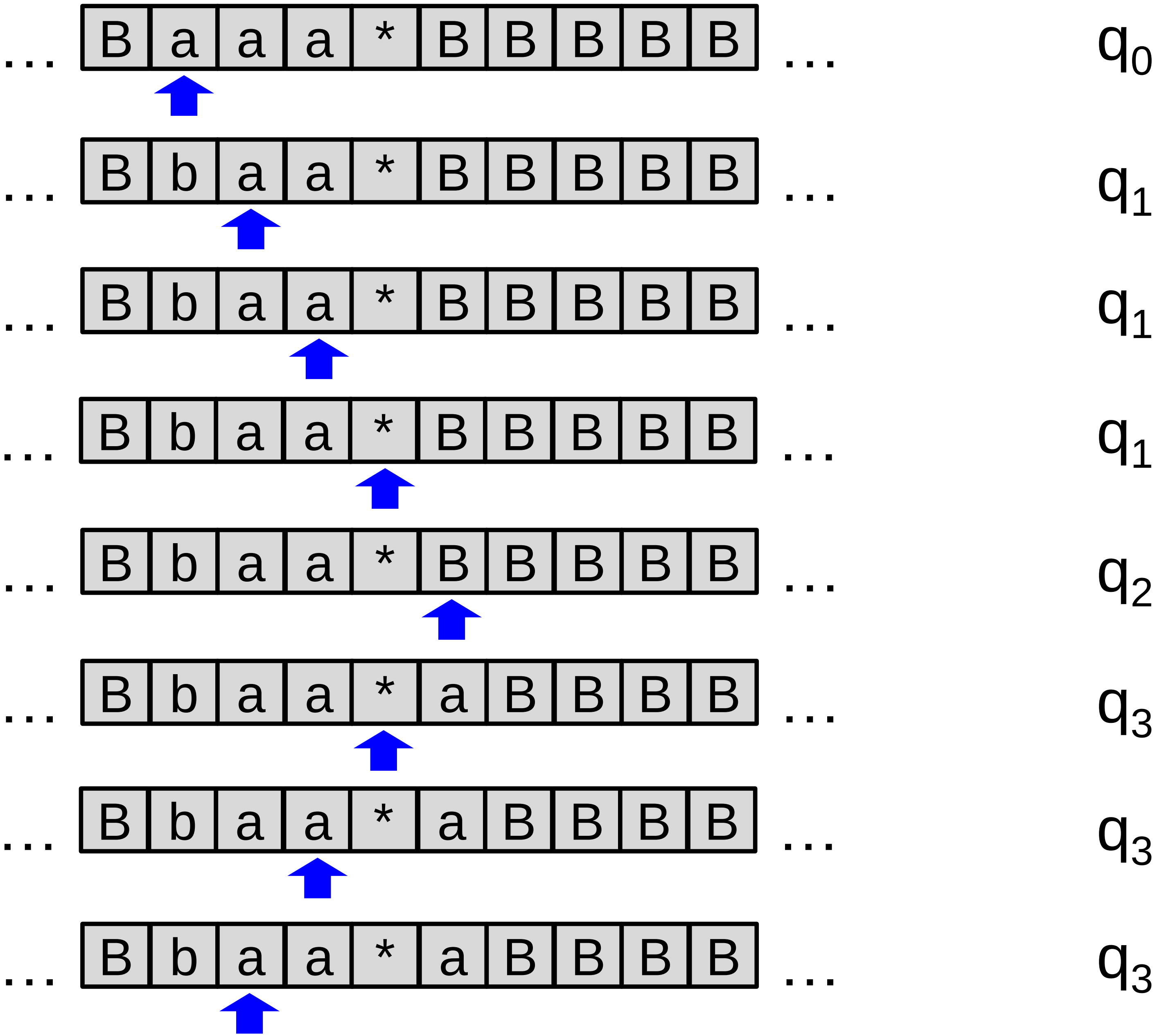
What did it do?

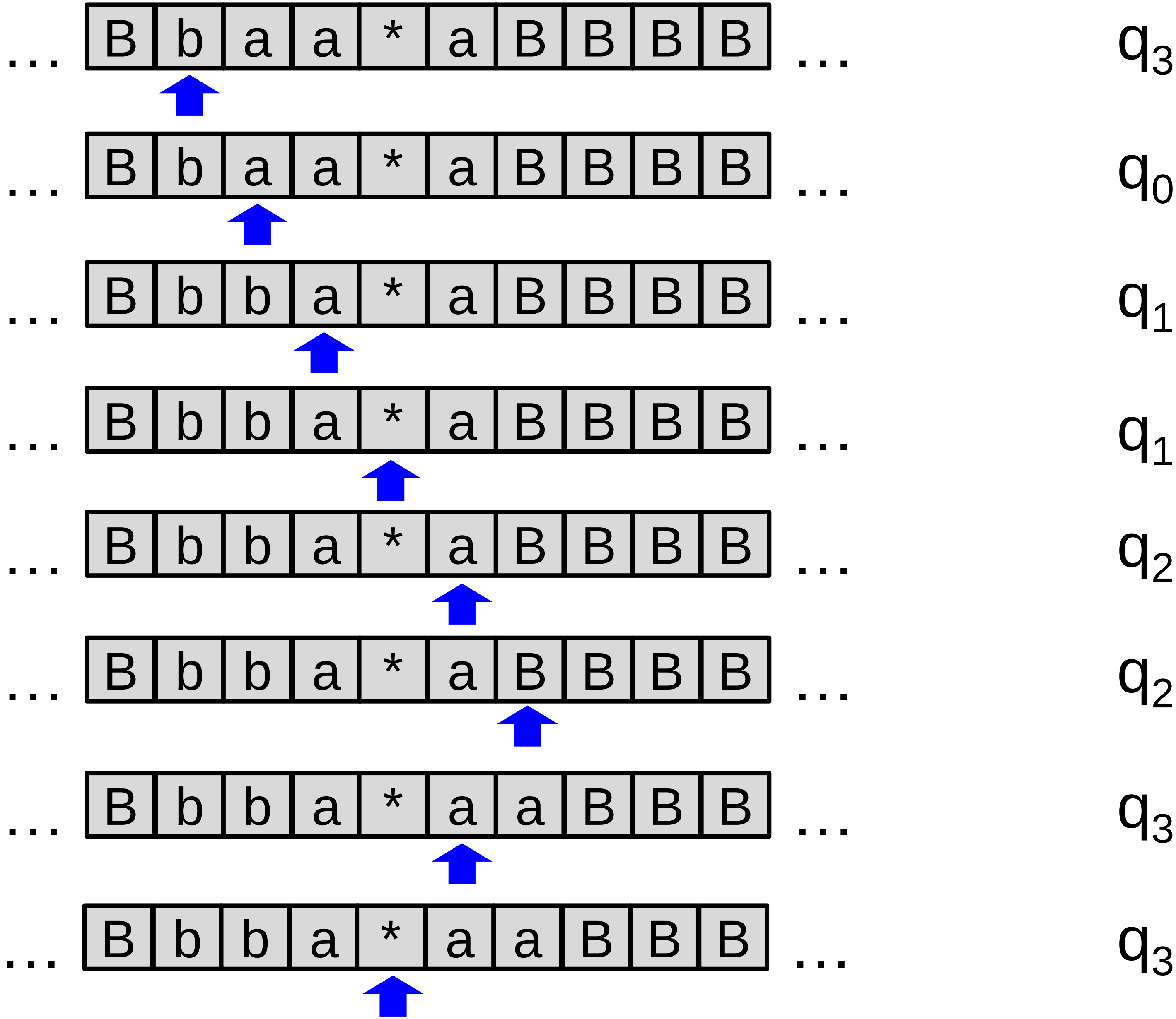
- *baab* became *abba*
- This machine swaps a to b and b to a until it finds a *

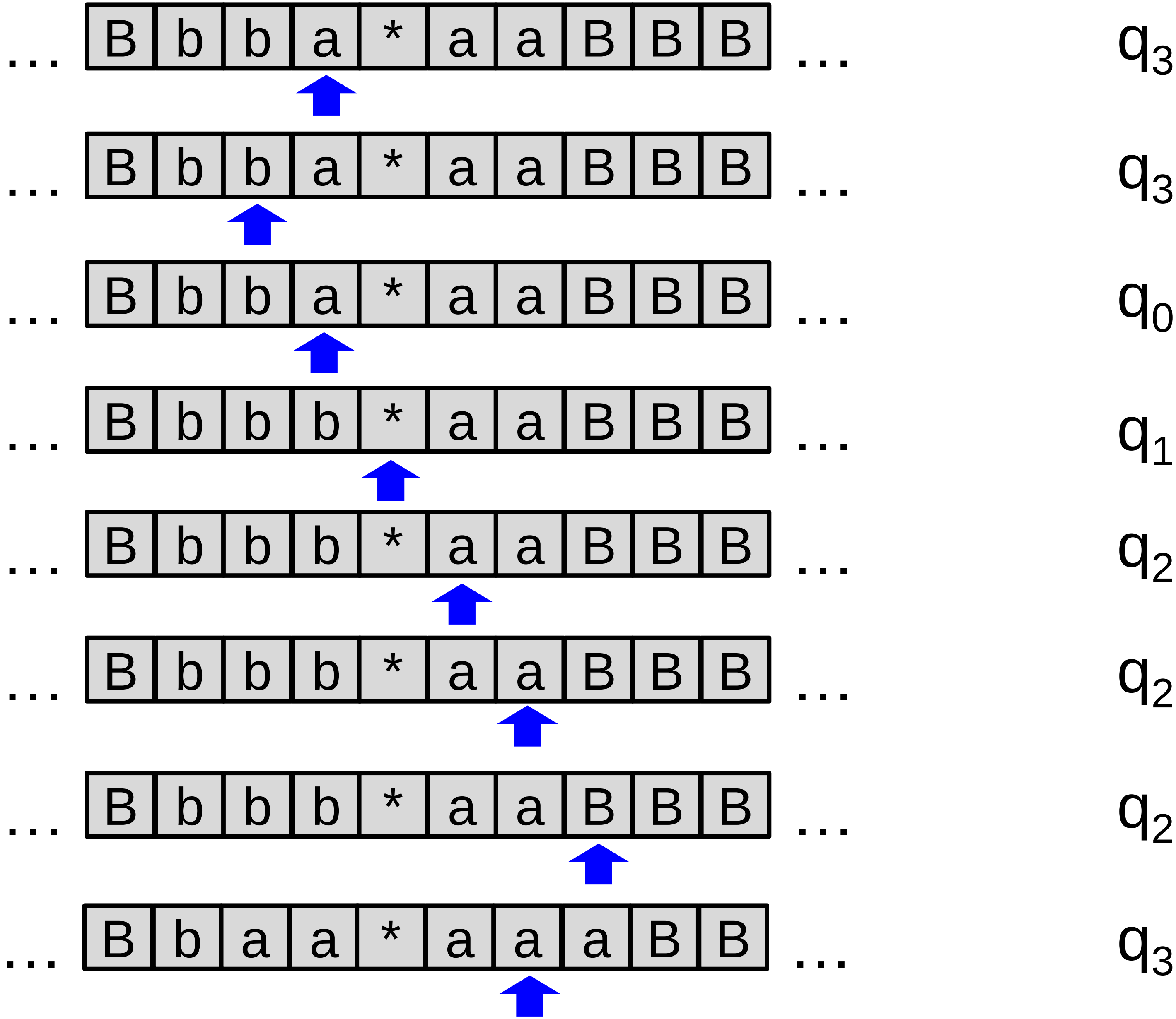
Another example

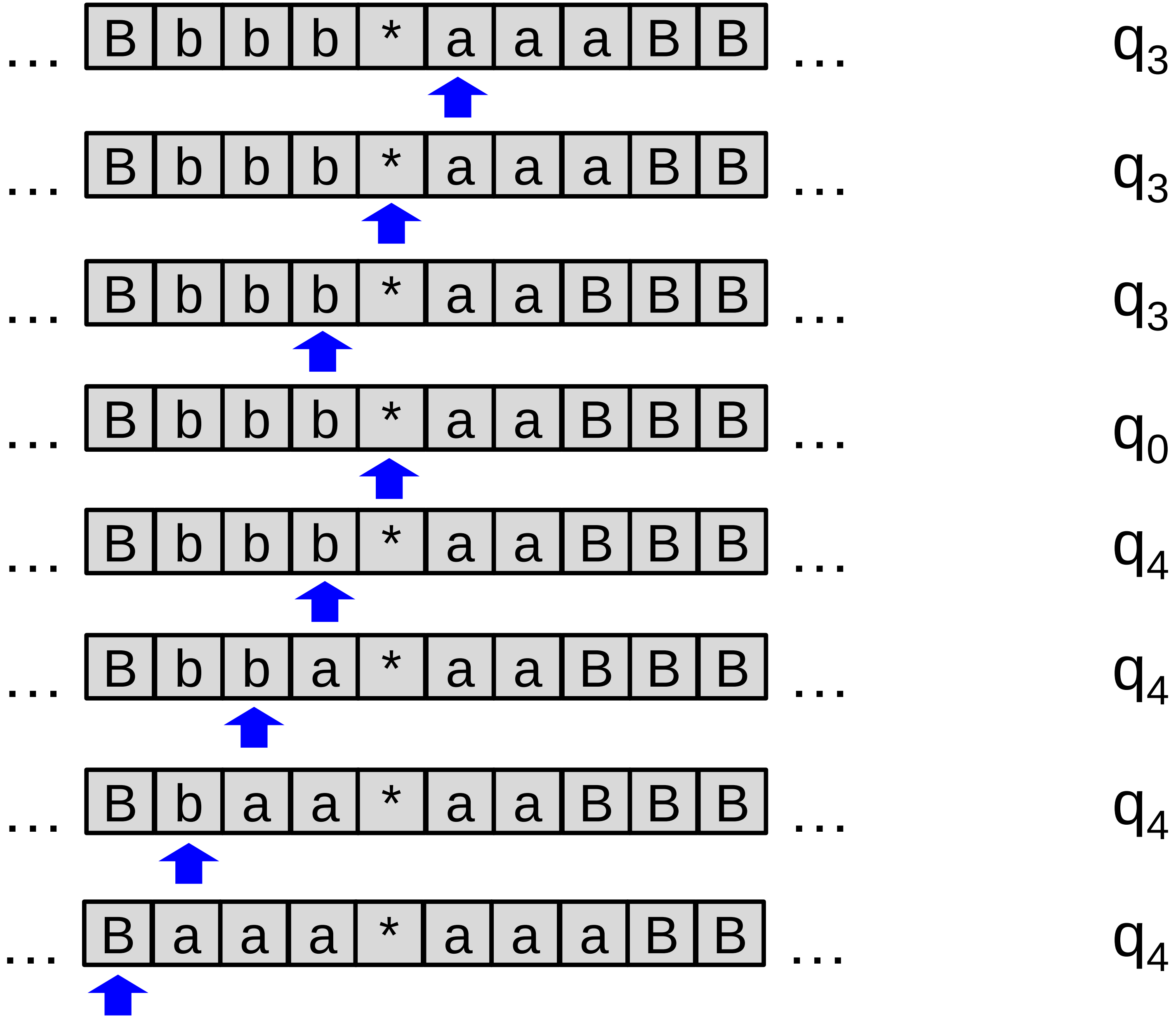


What does it do?


… B a a a * B B B B B … q0
… B b a a * B B B B B … q1
… B b a a * B B B B B … q1
… B b a a * B B B B B … q1
… B b a a * B B B B B … q2
… B b a a * a B B B B … q3
… B b a a * a B B B B … q3
… B b a a * a B B B B … q3


… B b a a * a B B B B … q3
… B b a a * a B B B B … q0
… B b b a * a B B B B … q1
… B b b a * a B B B B … q1
… B b b a * a B B B B … q2
… B b b a * a B B B B … q2
… B b b a * a a B B B … q3
… B b b a * a a B B B … q3


… B b b a * a a B B B … q_3
… B b b a * a a B B B … q_3
… B b b a * a a B B B … q_0
… B b b b * a a B B B … q_1
… B b b b * a a B B B … q_2
… B b b b * a a B B B … q_2
… B b b b * a a B B B … q_2
… B b a a * a a a B B … q_3



… B b b b * a a a B B … q3
… B b b b * a a a B B … q3
… B b b b * a a B B B … q3
… B b b b * a a B B B … q0
… B b b b * a a B B B … q4
… B b b a * a a B B B … q4
… B b a a * a a B B B … q4
… B a a a * a a a B B … q4

…	B	a	a	a	*	a	a	a	B	B	…

q_5

Final State

Initial State was:

…	B	a	a	a	*	B	B	B	B	B	…

q_0

- The machine makes a copy of n a’s and puts them behind the *

Multiplication

- We can use this machine to do “unary multiplication”:

…	B	a	a	a	*	a	a	%	B	B	B	B	B	B	B	…	q_0

- Multiply “number” before * by “number” between * and % (3 times 2 here):

…	B	a	a	a	*	a	a	%	a	a	a	a	a	a	B	…	q_{final}

- Can be done by adapting the discussed machine and using it repeatedly

Church/Turing Thesis

- Every computable function can be computed by a Turing Machine
- I.e.: Turing Machines are universal computing machines
- Every problem that can be solved by an algorithm can be solved by a Turing machine
- Where is the power coming from?

  The read/write input/output tape !

More about TM

- The tape can be used to record any data for later access
- There is always space available after last non-blank location
- There is no limit how often the tape is accessed
- You PC is less powerful than a TM – why?
  Because it has finite memory

Efficiency

- TM are universal but not efficient
- Progress can be really slow
- Looking up memory involves sequential access – the opposite of efficiency

Managing complexity

- One can encapsulate useful functionality in “separate” sub-routines
- Collection of states set aside for each subroutine
- (similar to structured programming approach)
- However, TM are mainly useful as a theoretical concept, not for solving real world problems!

Variations

- There are common variants of TM:
  - Multiple tapes
  - Single-side infinite tape
  - Non-deterministic TM
- It can be shown that these have all equivalent power to the TM discussed here.

Example: Non-deterministic TM

- To simulate non-deterministic TM:
  - 3 tapes:
  - One tape for original input
  - One tape for the choice sequence: (2,3,1,2)
  - One tape to run on current choice sequence
- For this to work we need to enumerate all possible sequences of choices (ok, as states are finite)

Another equivalence

- The “2 pushdown” automaton is equivalent to the Turing Machine:
  - One pushdown holds tape contents to left of tape head
  - One pushdown holds tape contents to the right of tape head
  - As tape head moves, symbols shift across from one pushdown to another

More generally …

- Chomsky Hierarchy (for language classes):
  - Type 0: Languages accepted by Turing Machines
  - Type 1: Languages accepted by Turing Machines with linear bounded storage
  - Type 2: Languages accepted by Pushdown Automata
  - Type 3: Languages accepted by Finite State Automata

Alternative Characterization

- Equivalent grammar formalisms
  - Type 0: Languages generated by unrestricted grammars
  - Type 1: Languages generated by context-sensitive grammars
  - Type 2: Languages generated by context-free grammars
  - Type 3: Languages generated by regular grammars

Equivalence and Inclusions