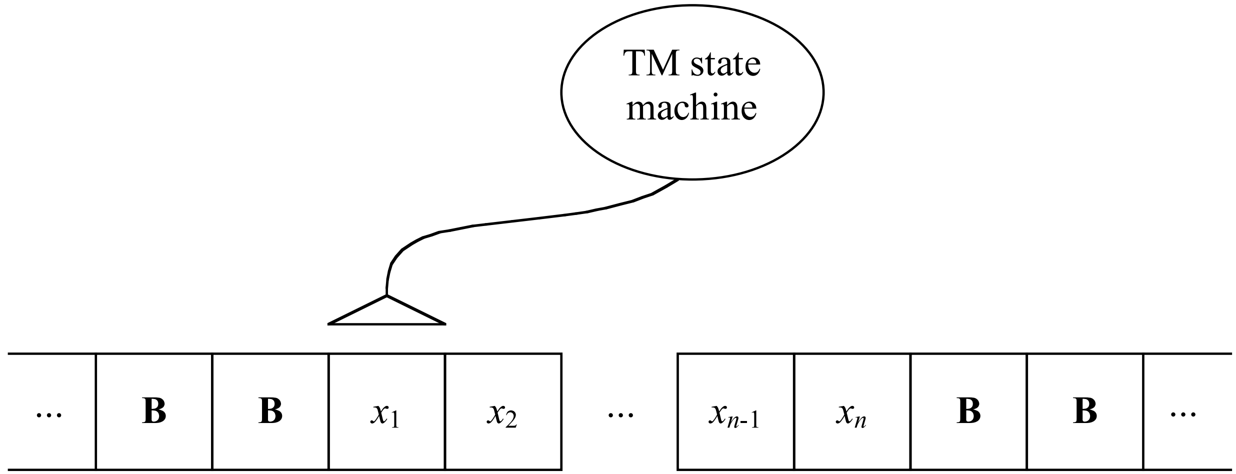
Chapter Sixteen: Turing Machines

We now turn to the most powerful kind of automaton we will study: the Turing machine. Although it is only slightly more complicated than a finite-state machine, a Turing machine can do much more. It is, in fact, so powerful that it is the accepted champion of automata. No other, more powerful model exists.

The TM is the strongest of computational mechanisms, the automaton of steel. But like all superheroes it does have one weakness, revealed in this chapter: it can get stuck in an infinite loop.

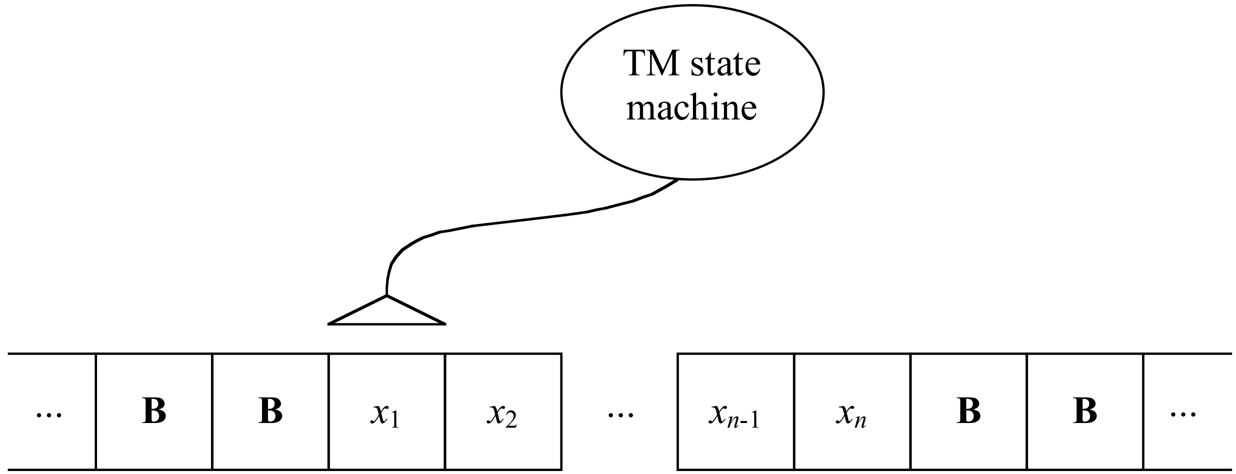
Outline

Turing Machine (TM)



- Input is a tape with one symbol at each position
- Tape extends infinitely in both directions
- Those positions not occupied by the input contain a special blank-cell symbol **B**
- The TM has a head that can read and write, and can move in both directions
- Head starts at the first input symbol

Turing Machine (TM)



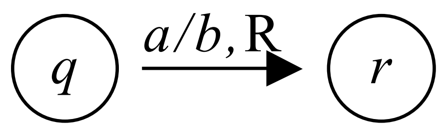
- A state machine controls the read/write head
- Move is determined by current state and symbol
- On each move: write a symbol, make a state transition, and move the head one place, left or right
- If it enters an accepting state, it halts and accepts

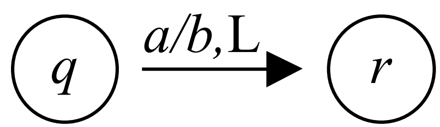
Difference From DFA

- TMs are like DFAs, but with:
  - Ability to move both left and right, unboundedly
  - Ability to write as well as read
- Important difference about how they accept:
  - A DFA reads to the end of the input, then accepts if that last state is accepting
  - A TM accepts the moment it enters an accepting state; final tape and head position don't matter; it doesn't even have to read all its input
  - Transitions leaving an accepting state are never used, so there is never any need for more than one accepting state

TM Transitions

- State-transition diagrams





- Right moves: if in state *q*, and the current tape symbol is *a*, write *b* over the *a*, move one place to the right, and go to state *r*
- Left moves: same, but move left

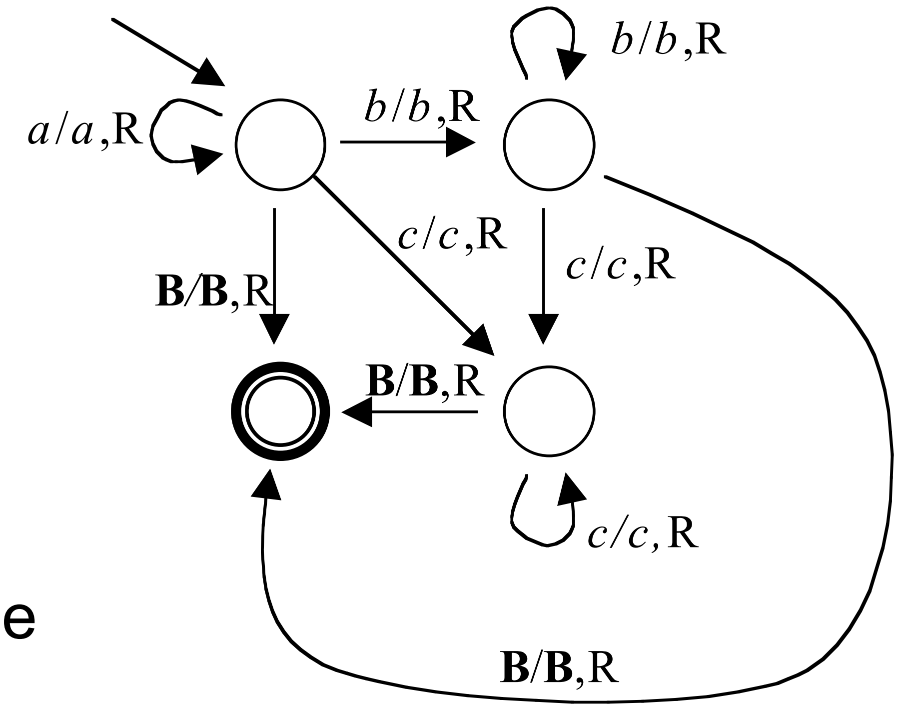
Outline

- 16.1 Turing Machine Basics
- 16.2 Simple TMs
- 16.3 A TM for $\{a^n b^n c^n\}$
- 16.4 The 7-Tuple
- 16.5 The Languages Defined By A TM
- 16.6 To Halt Or Not To Halt
- 16.7 A TM for $\{xcx \mid x \in \{a,b\}^*\}$
- 16.8 Three Tapes
- 16.9 Simulating DFAs
- 16.10 Simulating Other Automata

TMs For Regular Languages

- TMs can easily handle all regular languages
- In fact, a TM that always moves the head to the right works much like a DFA
- Except for that difference about the mechanism for accepting
- A TM only enters its accepting state when it has reached a final decision

$L(a^*b^*c^*)$



- Like a DFA:
  - Always moves right
  - Does not change the tape (always writes what it just read)
- Since it never moves left, it really doesn't matter what it writes
- It could write **B** on every move, erasing as it reads, and still accept the same language

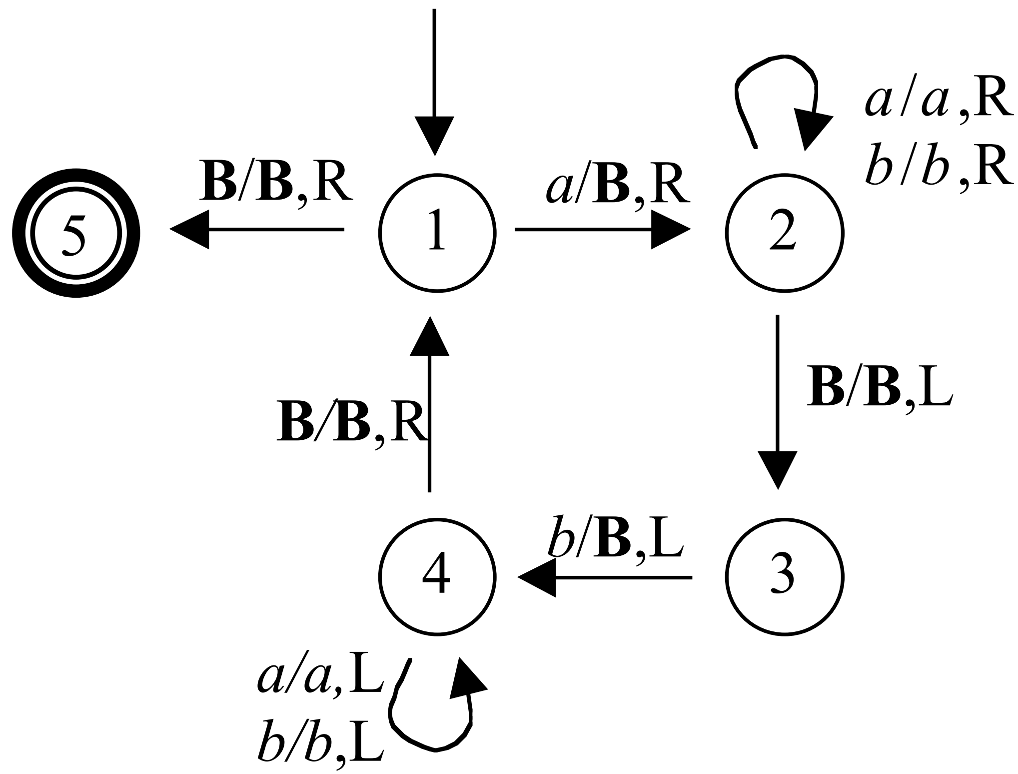
TMs For Context-Free Languages

- TMs can also easily handle all CFLs
- It is possible to take any stack machine and convert it into an equivalent TM that uses the infinite tape as a stack
- We'll demonstrate this more generally later
- But it is often easier to find some non-stack-oriented approach
- For example, $\{a^n b^n\}$…

Strategy For $\{a^n b^n\}$

- Repeatedly erase first (*a*) and last (*b*); if the string was in $\{a^n b^n\}$, this leaves nothing
- Five steps:
  1. If the current symbol is **B**, go to step 5. If the current symbol is *a*, write a **B** over it and go to step 2.
  2. Move right past any *a*s and *b*s. At the first **B**, move left one symbol and go to step 3.
  3. If the current symbol is *b*, write a **B** over it and go to step 4.
  4. Move left past any *a*s and *b*s. At the first **B**, move right one symbol and go to step 1.
  5. Accept.

$\{a^n b^n\}$

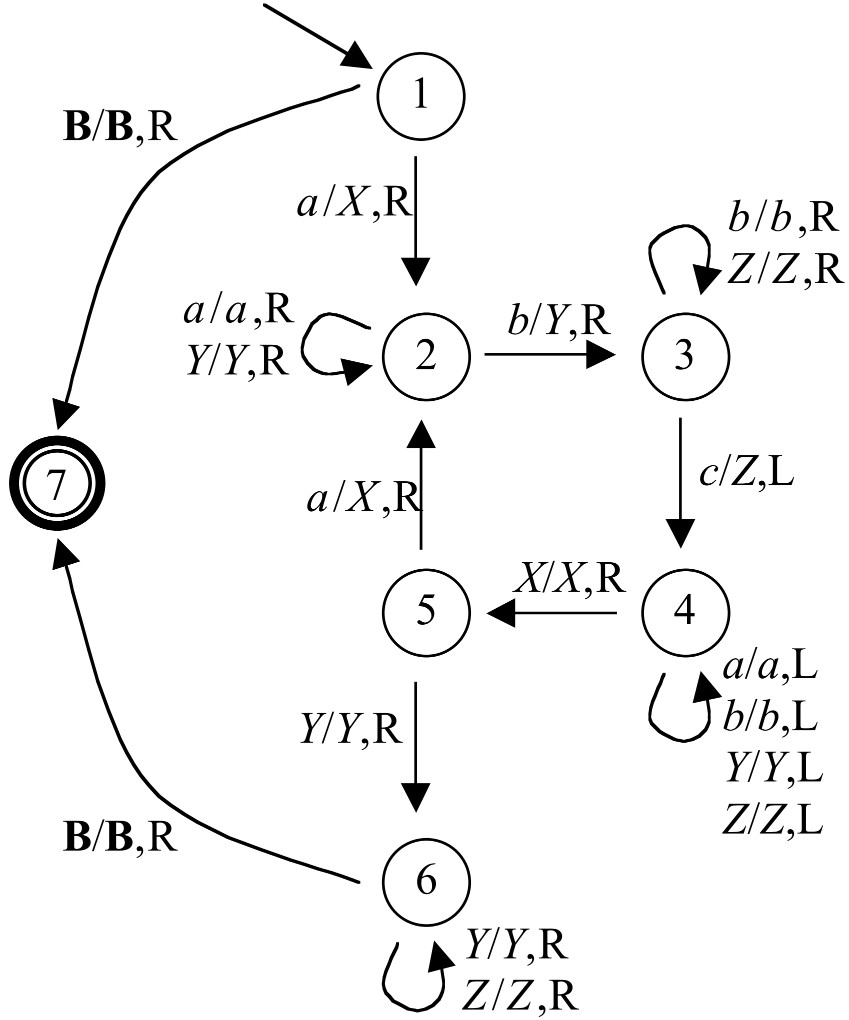


Outline

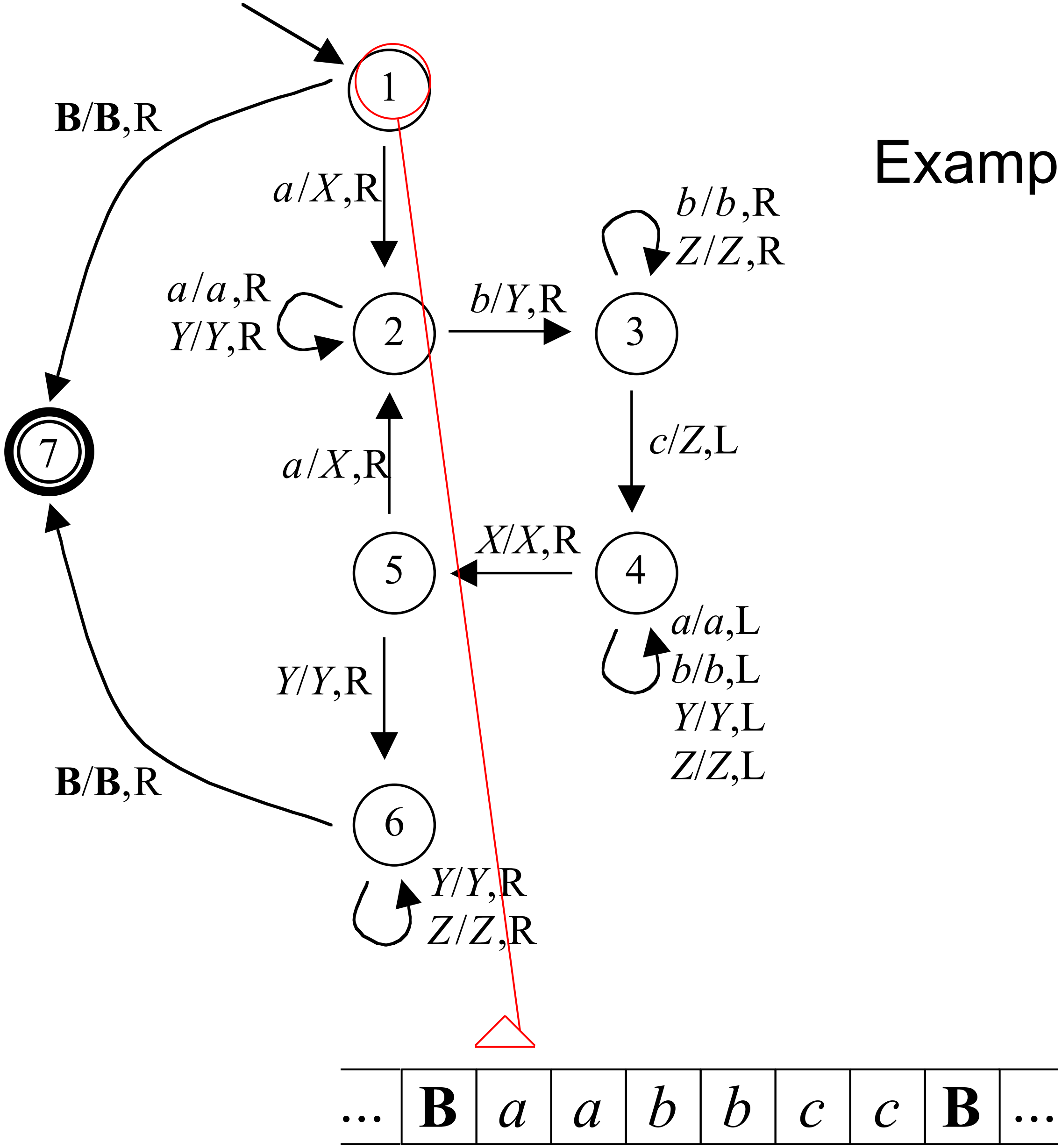
- 16.1 Turing Machine Basics
- 16.2 Simple TMs
- 16.3 A TM for $\{a^n b^n c^n\}$
- 16.4 The 7-Tuple
- 16.5 The Languages Defined By A TM
- 16.6 To Halt Or Not To Halt
- 16.7 A TM for $\{xcx \mid x \in \{a,b\}^*\}$
- 16.8 Three Tapes
- 16.9 Simulating DFAs
- 16.10 Simulating Other Automata

Strategy For $\{a^n b^n c^n\}$

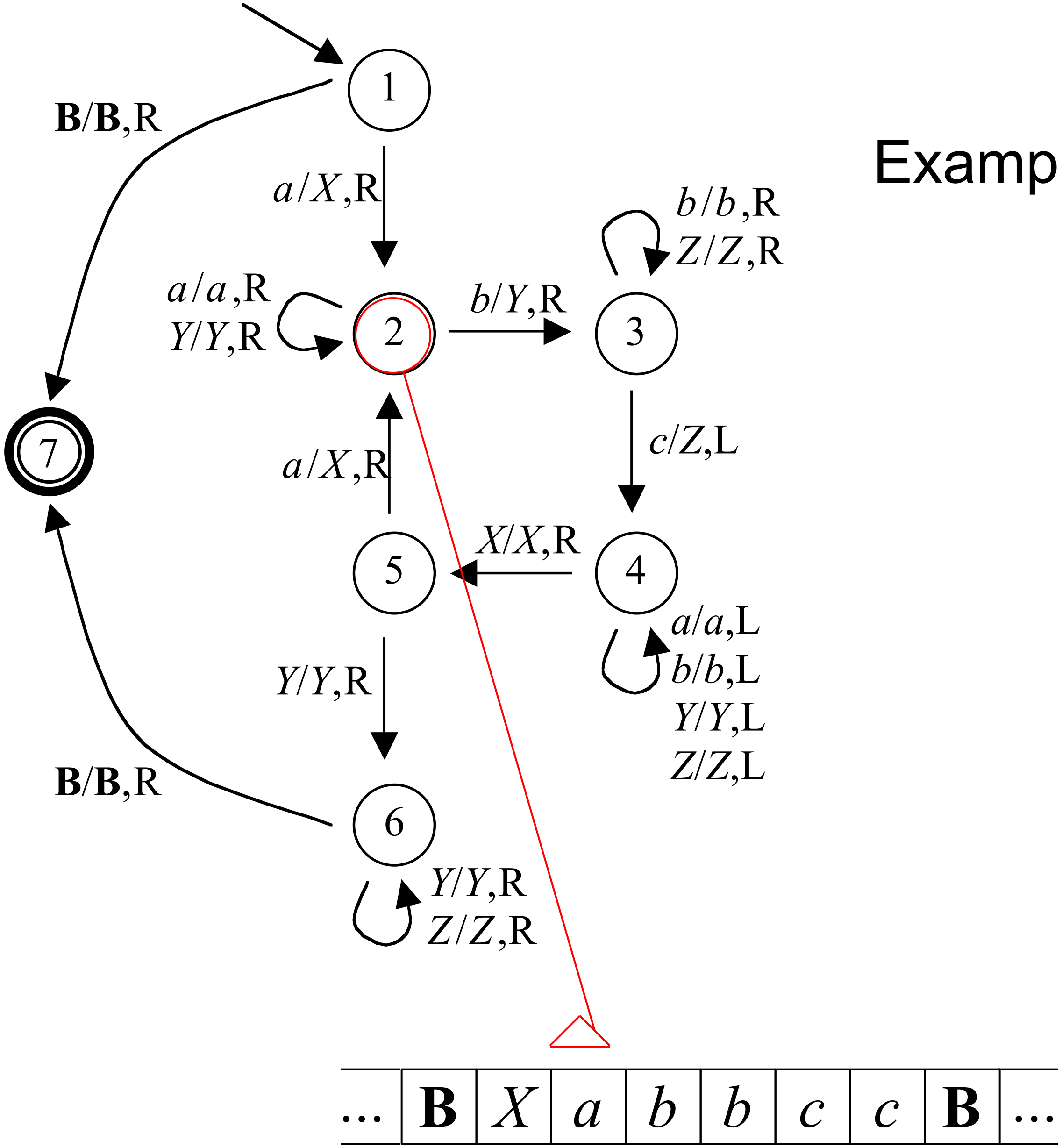
1. If the current symbol is **B**, go to step 7. If the current symbol is *a*, write an *X* over it and go to step 2.
2. Move right past any *a*s and *Y*s. At the first *b*, write *Y* over it and go to step 3.
3. Move right past any *b*s and *Z*s. At the first *c*, write *Z* over it and go to step 4.
4. Move left past any *a*s, *b*s, *Z*s, and *Y*s. At the first *X*, move right 1 symbol and go to step 5.
5. If the current symbol is *a*, write *X* and go to step 2. If the current symbol is *Y* go to step 6.
6. Move right past any *Y*s and *Z*s. At the first **B**, go to step 7.
7. Accept.


1
B/B,R
a/X,R
b/b,R
Z/Z,R
a/a,R
Y/Y,R
2
b/Y,R
3
7
a/X,R
c/Z,L
X/X,R
5
4
a/a,L
b/b,L
Y/Y,L
Z/Z,L
Y/Y,R
B/B,R
6
Y/Y,R
Z/Z,R

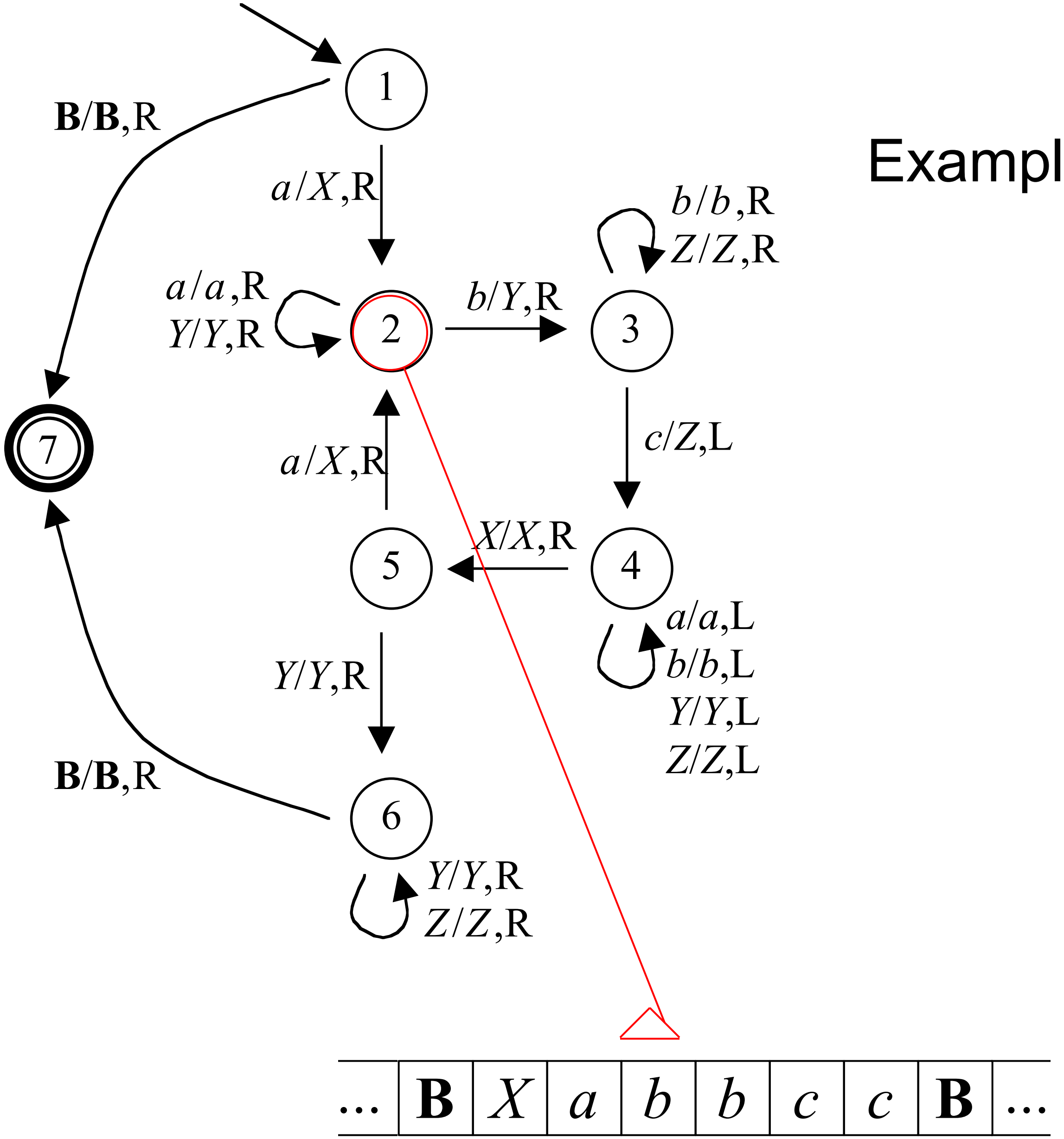
Example: *aabbcc*



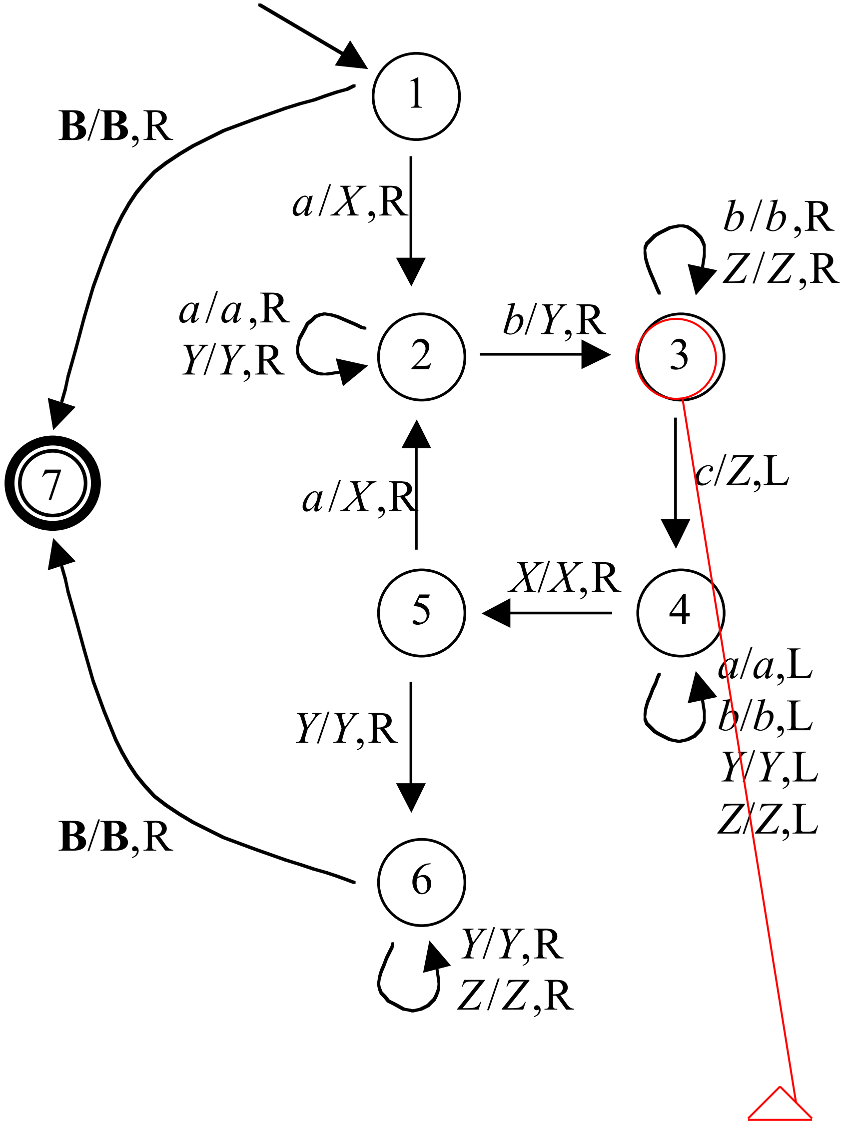
Example: *aabbcc*

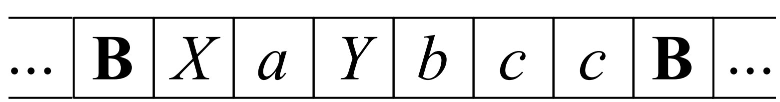


Example: *aabbcc*

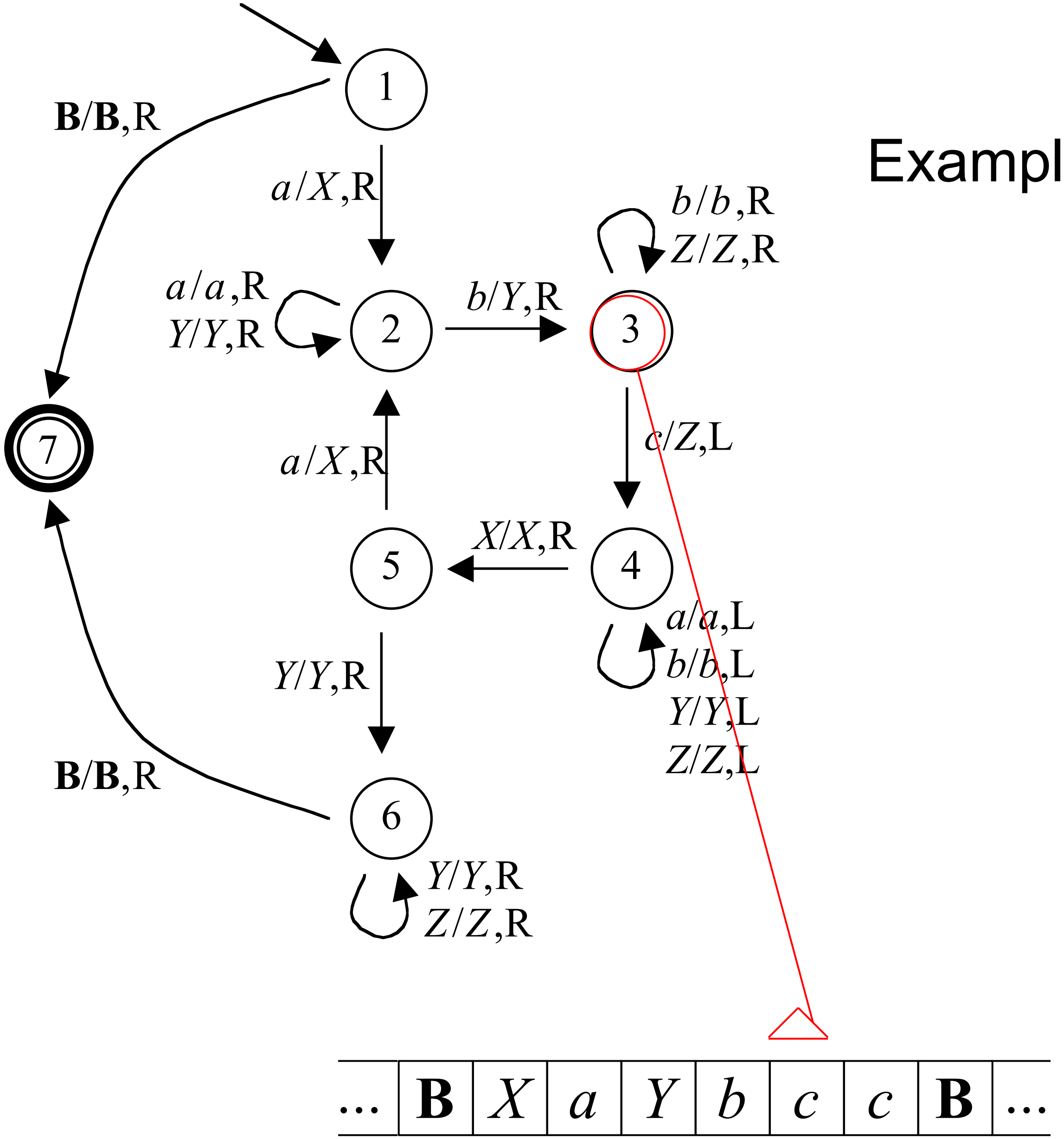


Example: *aabbcc*

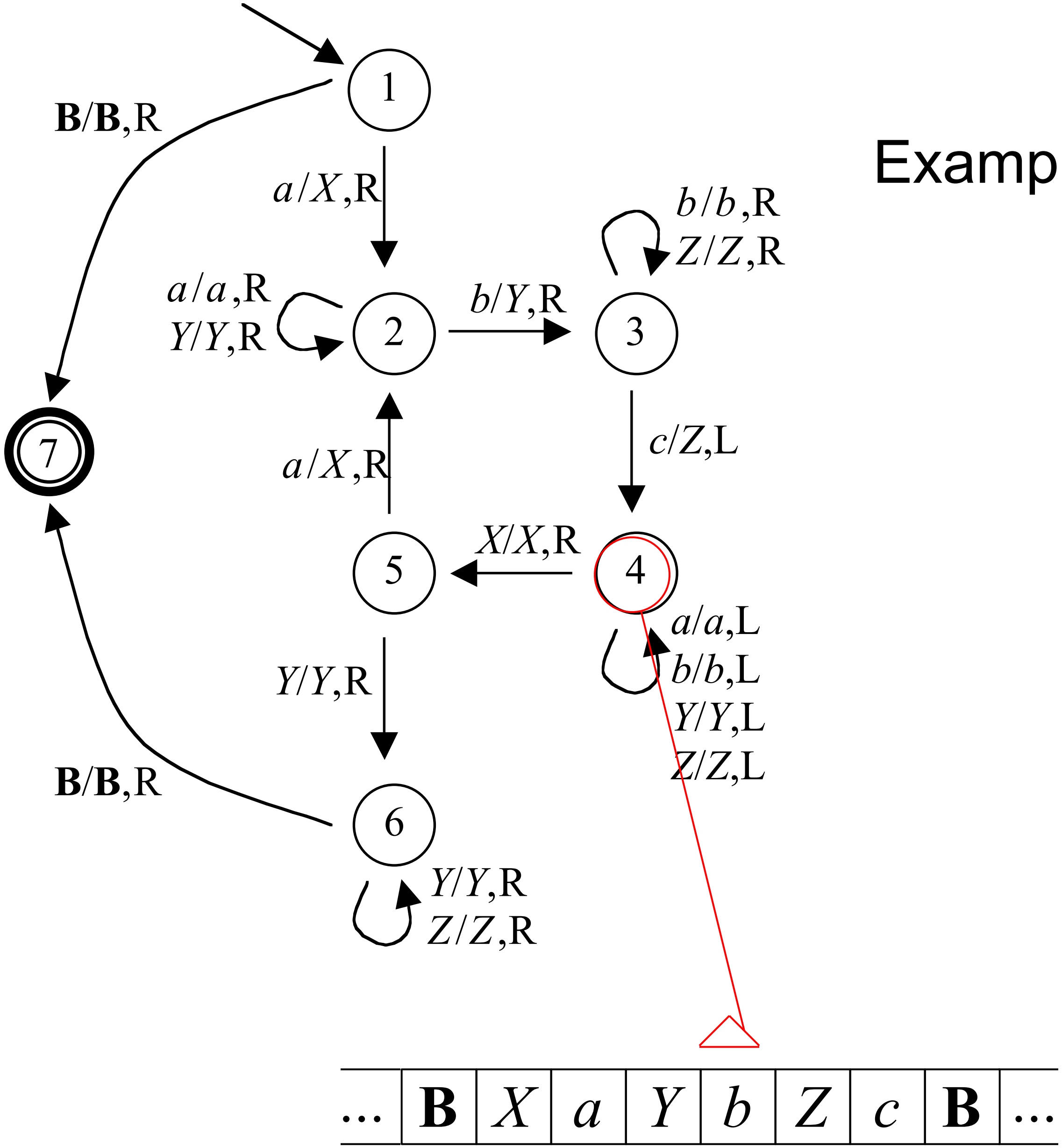




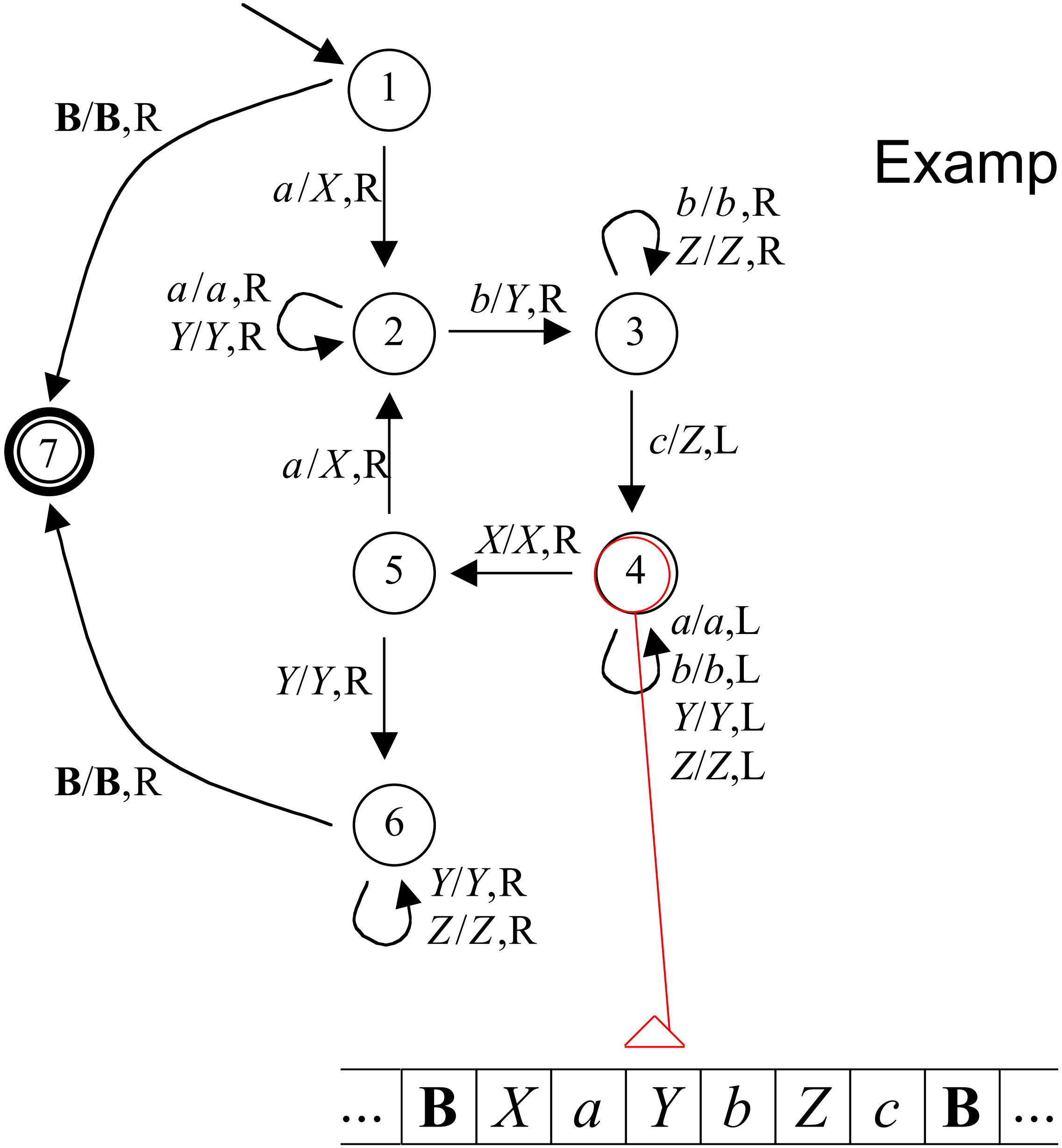
Example: *aabbcc*



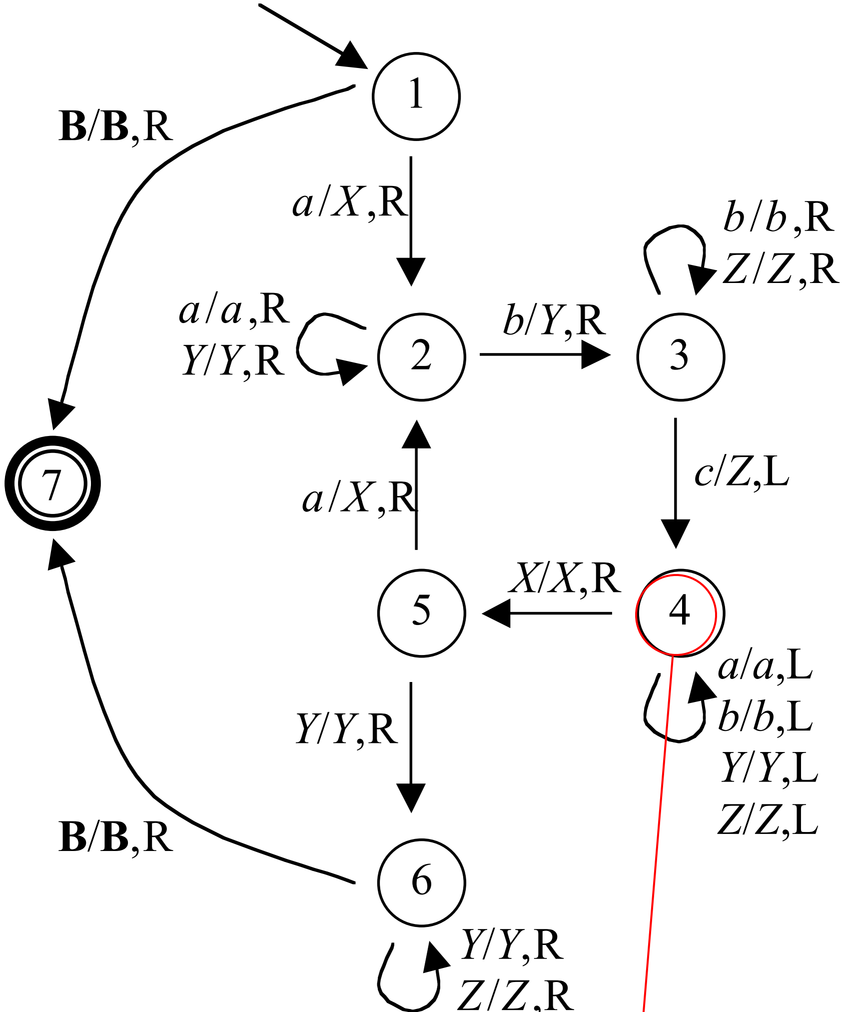
Example: *aabbcc*



Example: *aabbcc*

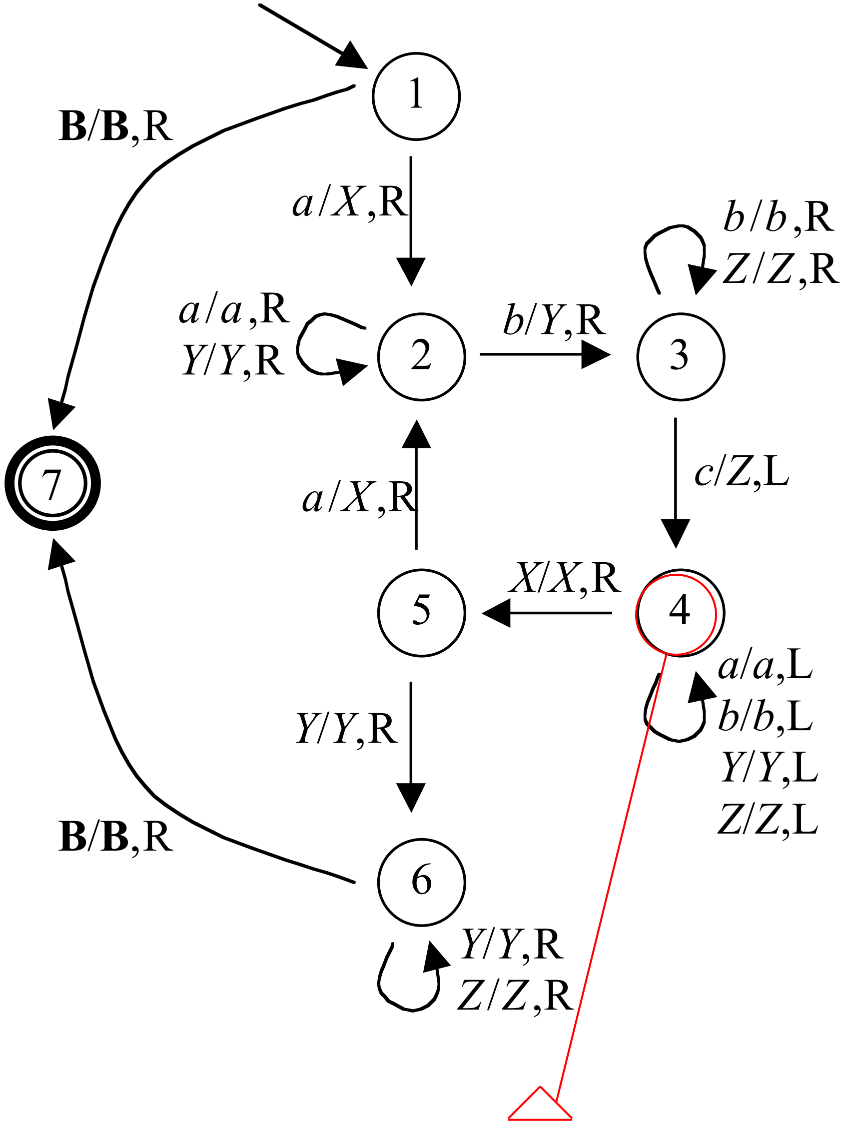


Example: *aabbcc*

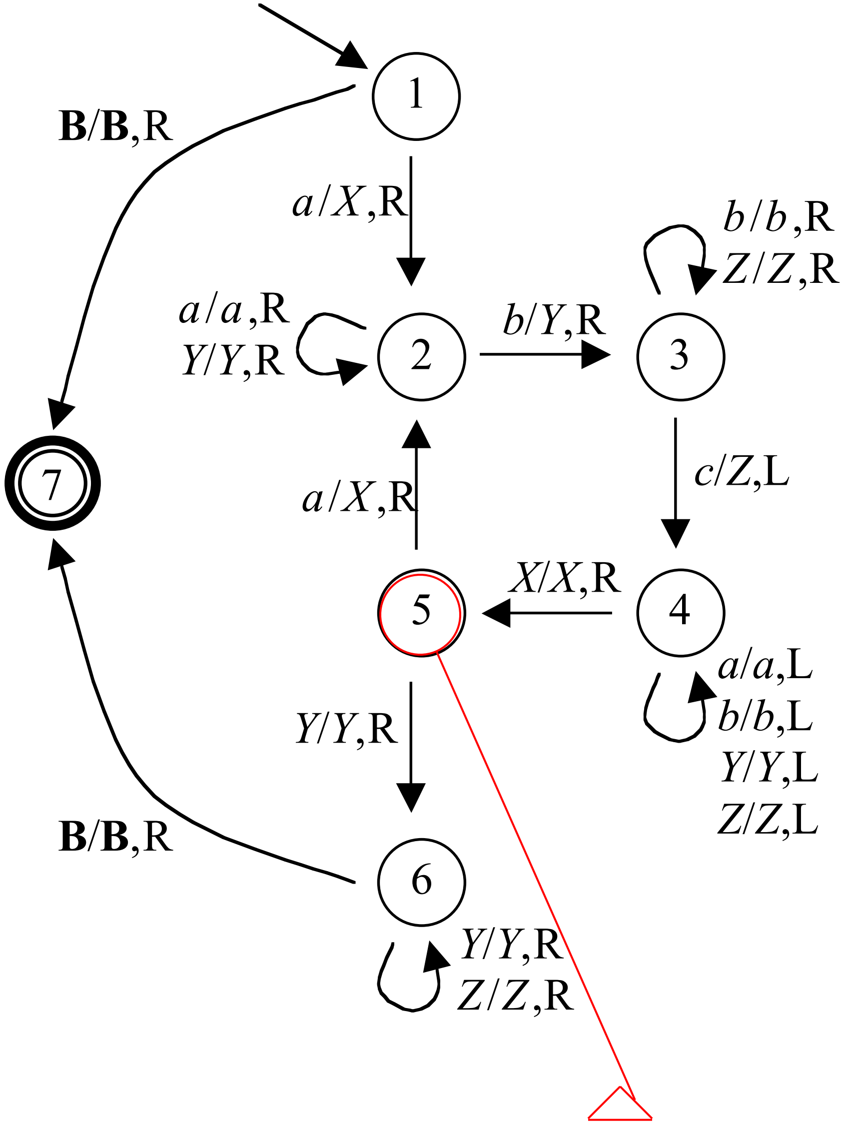


…	**B**	*X*	*a*	*Y*	*b*	*Z*	*c*	**B**	…

Example: *aabbcc*



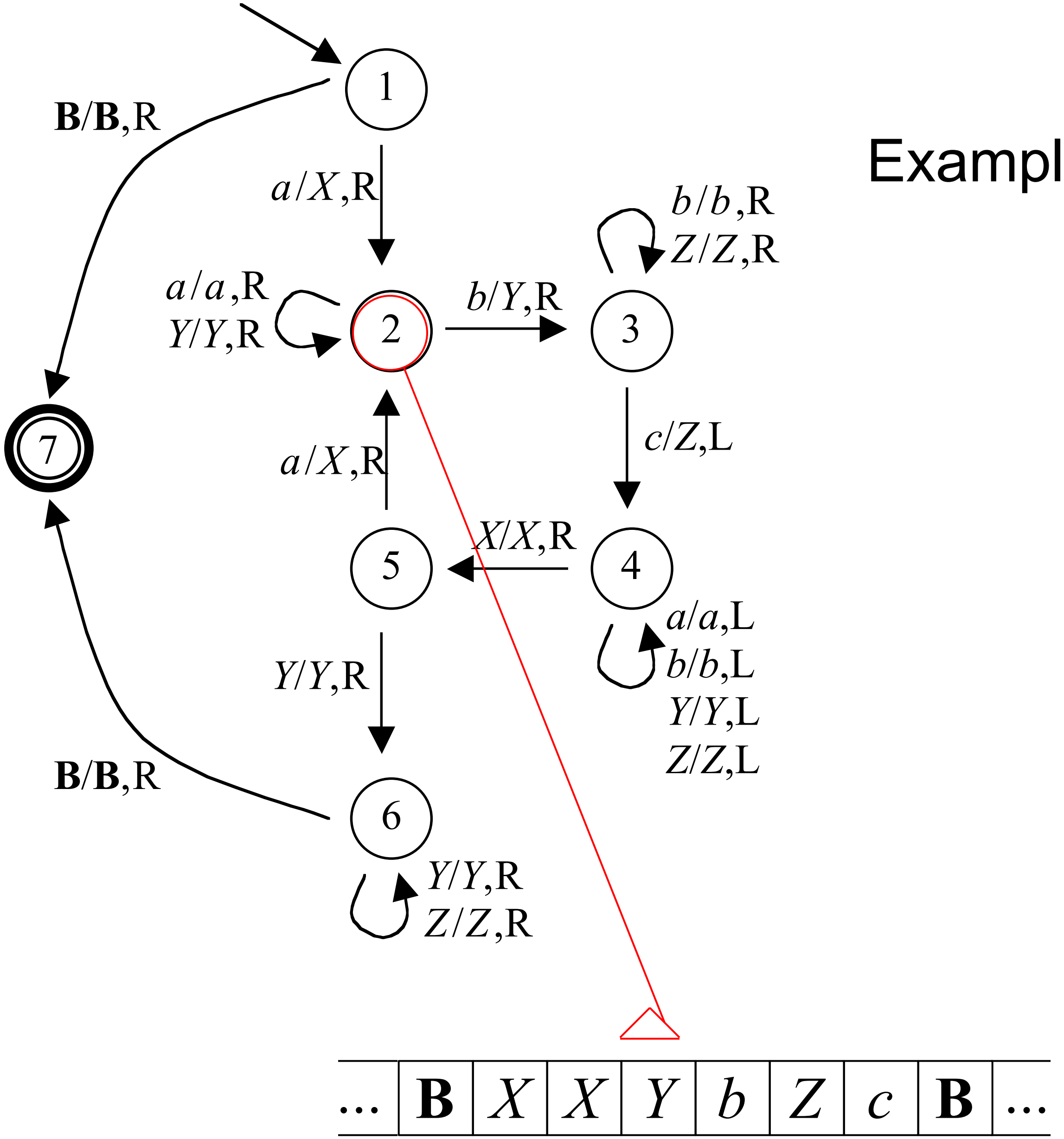
…	**B**	*X*	*a*	*Y*	*b*	*Z*	*c*	**B**	…



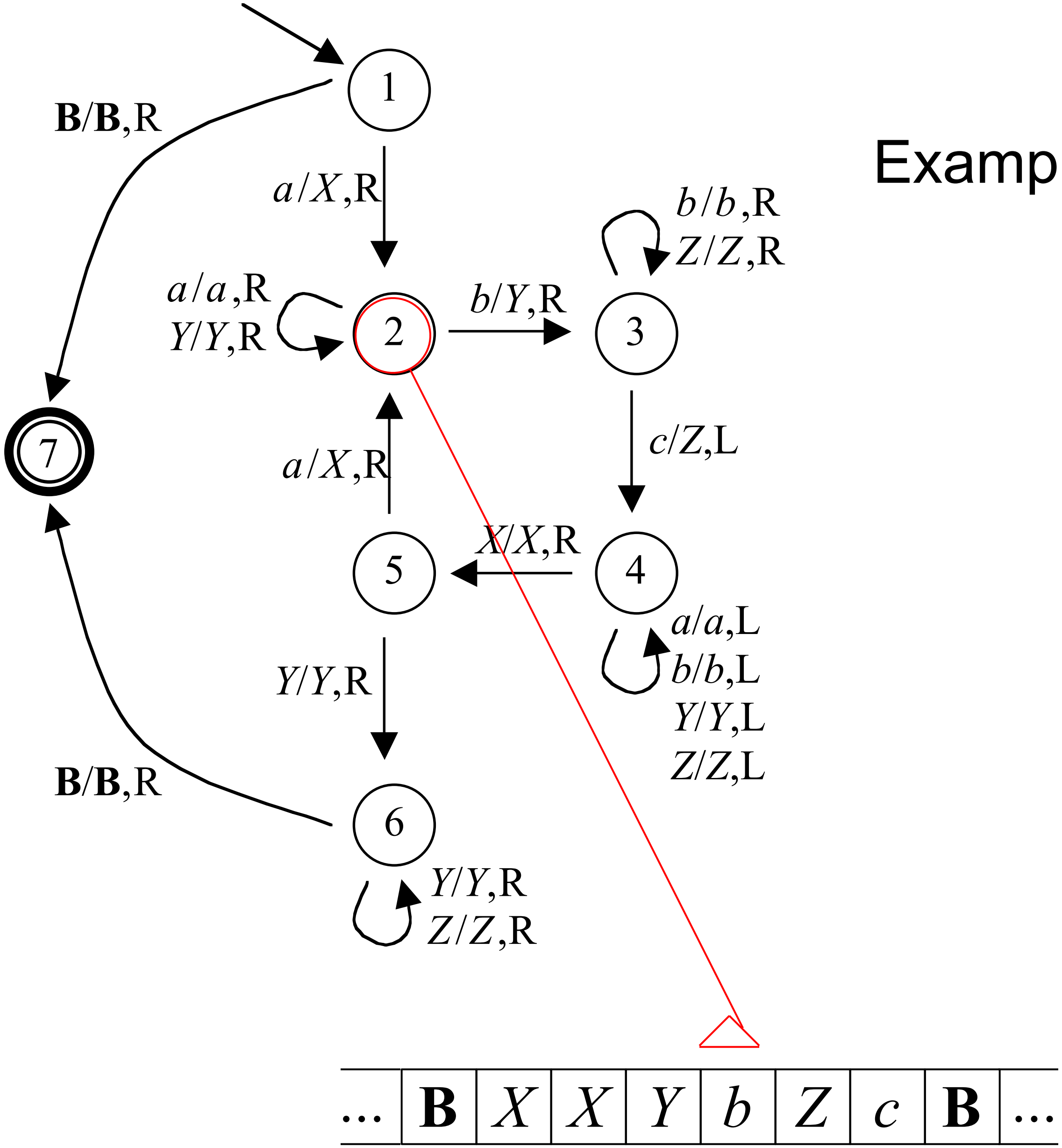
Example: *aabbcc*

…	**B**	*X*	*a*	*Y*	*b*	*Z*	*c*	**B**	…

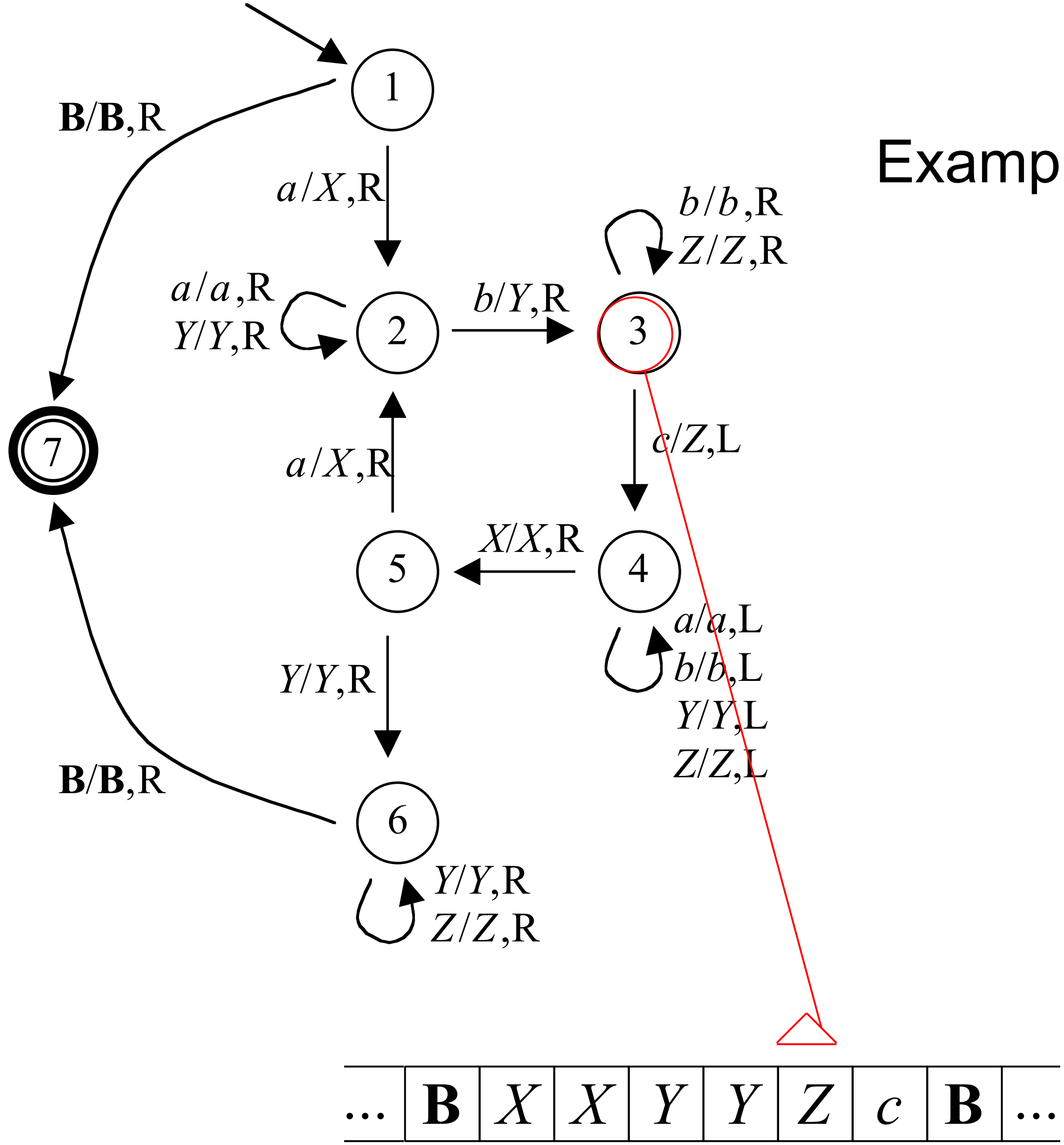
Example: *aabbcc*



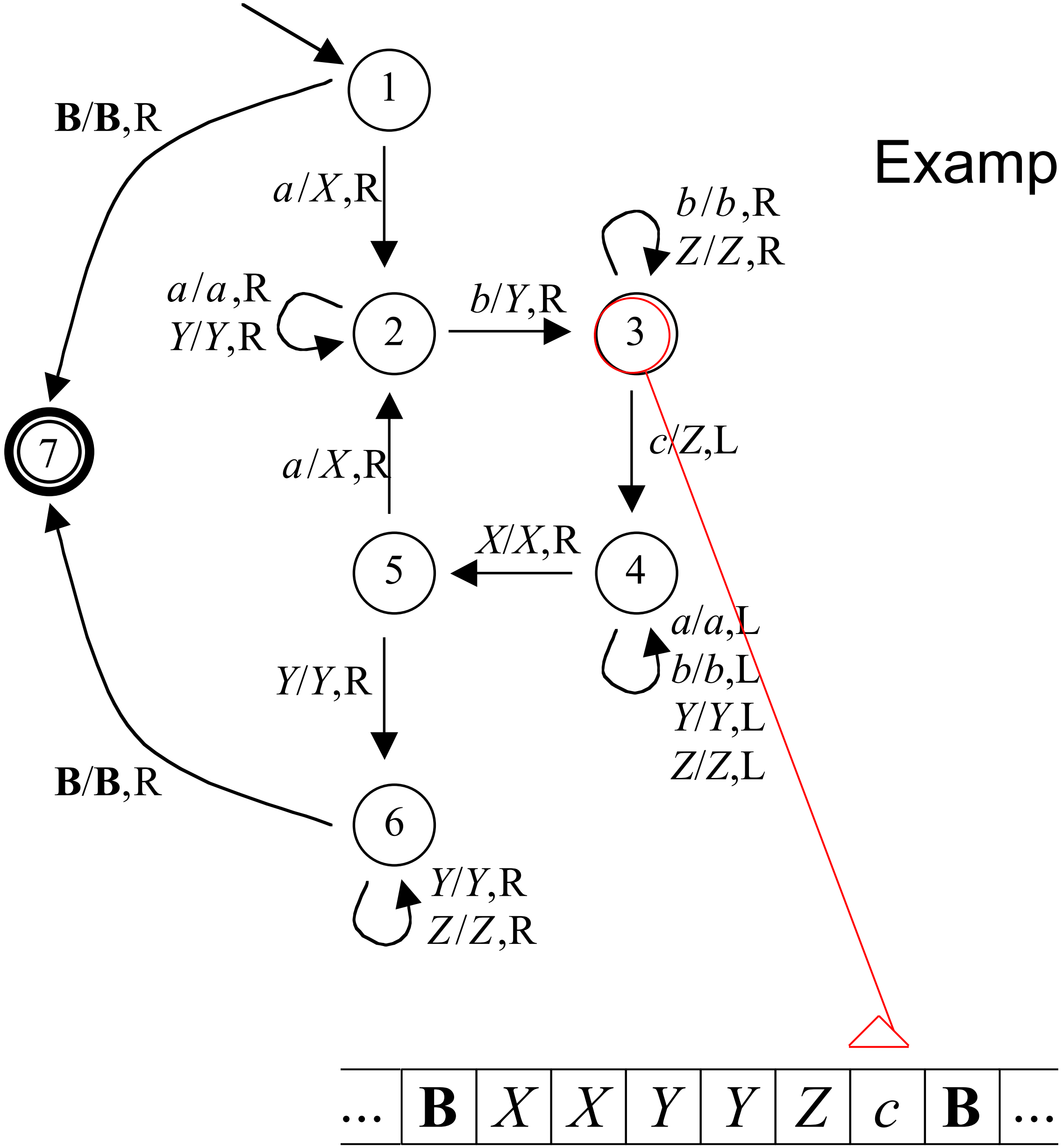
Example: *aabbcc*



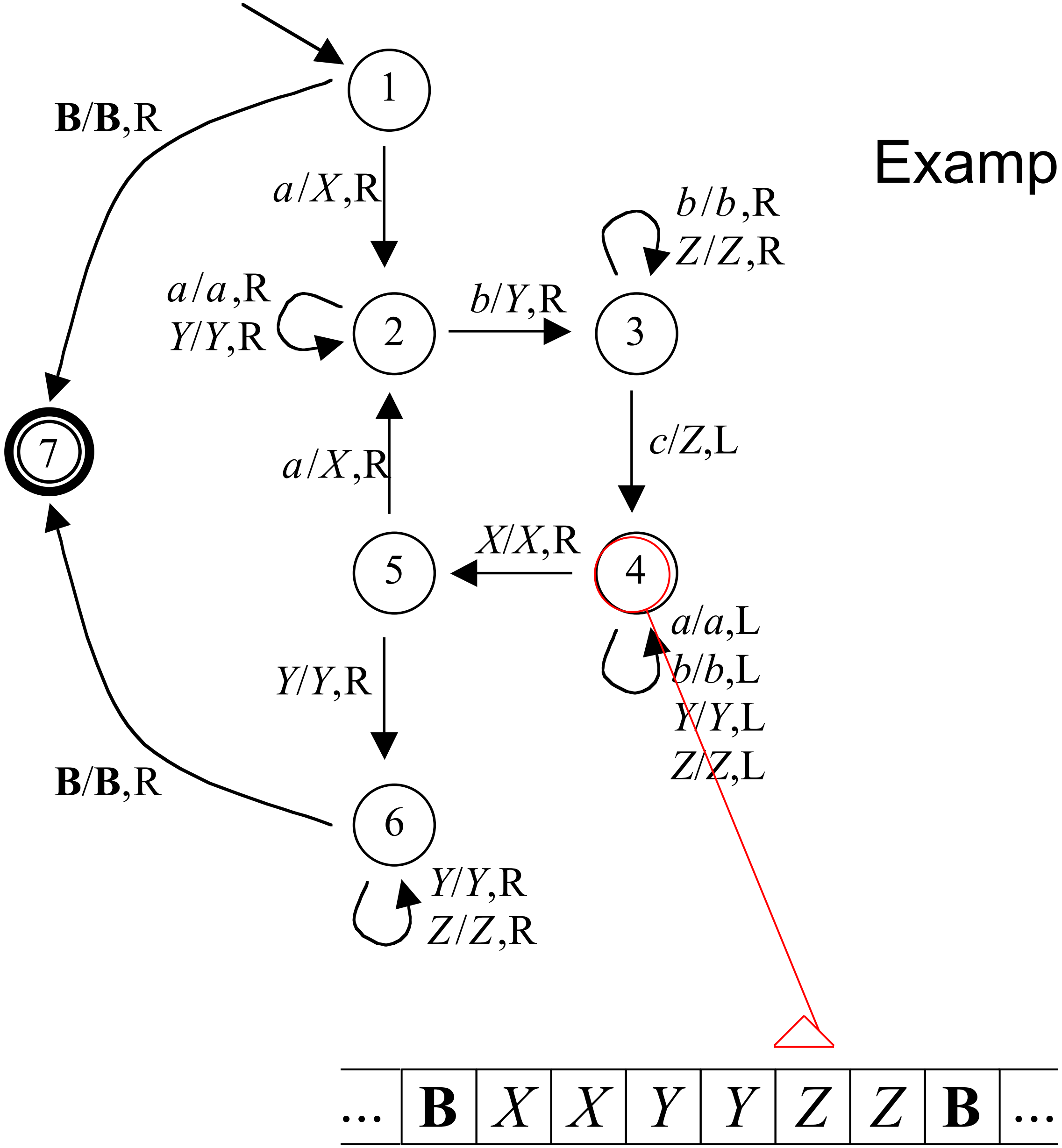
Example: *aabbcc*



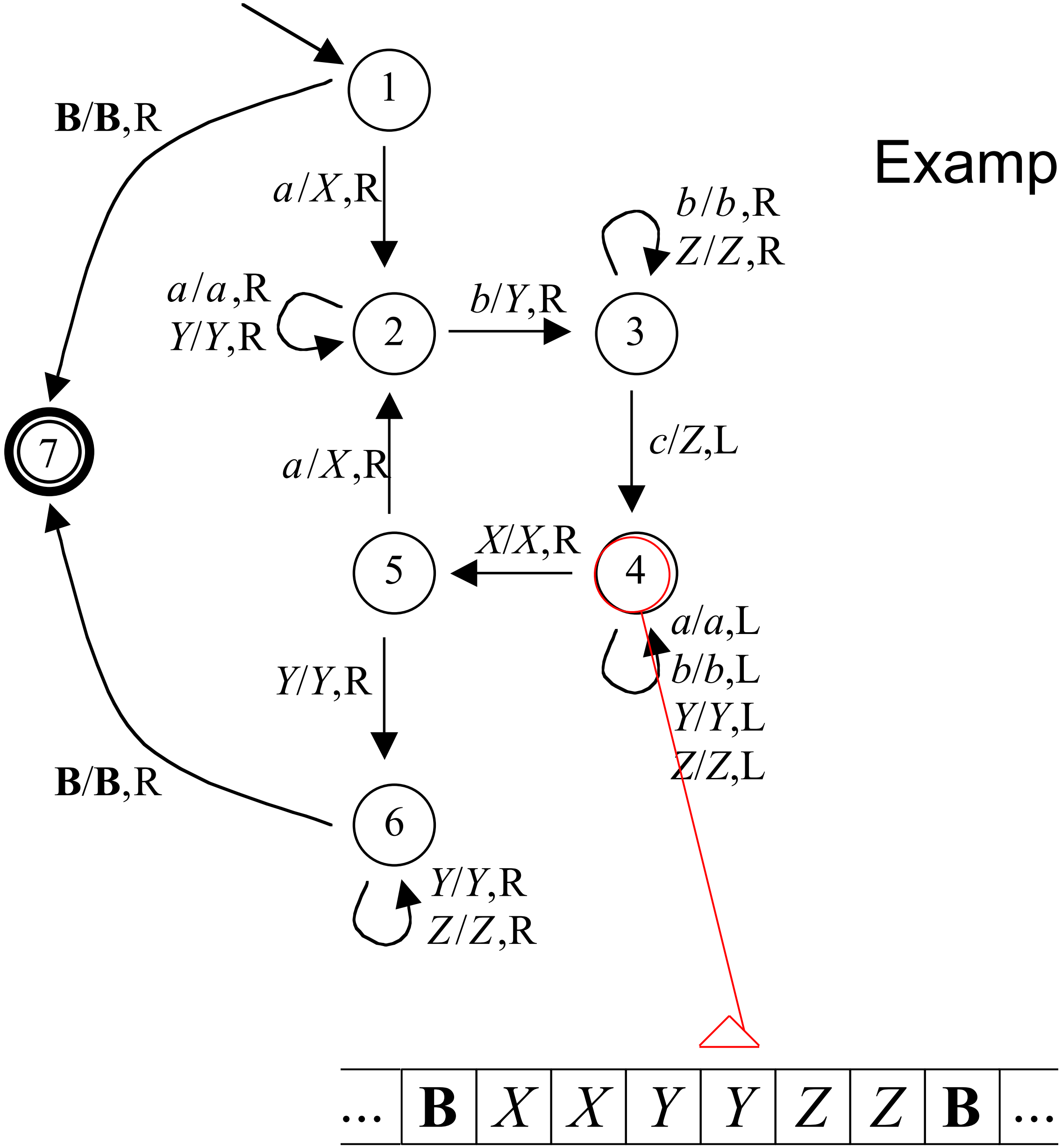
Example: *aabbcc*



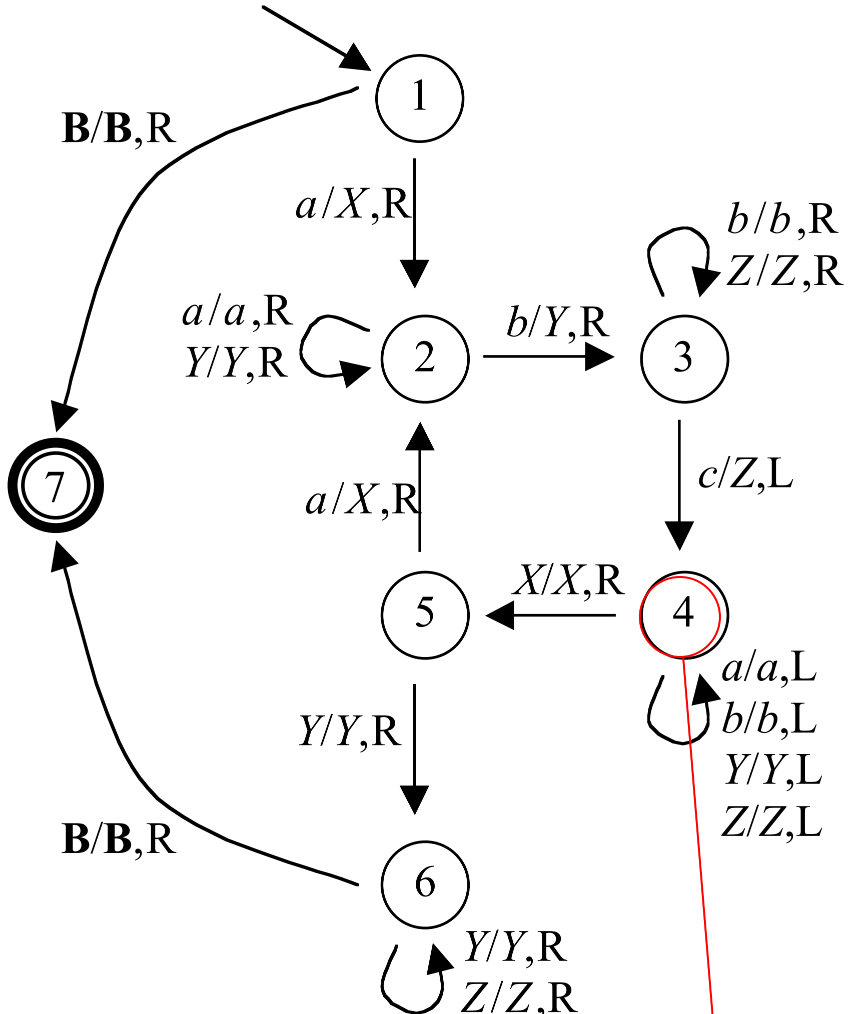
Example: *aabbcc*



Example: *aabbcc*

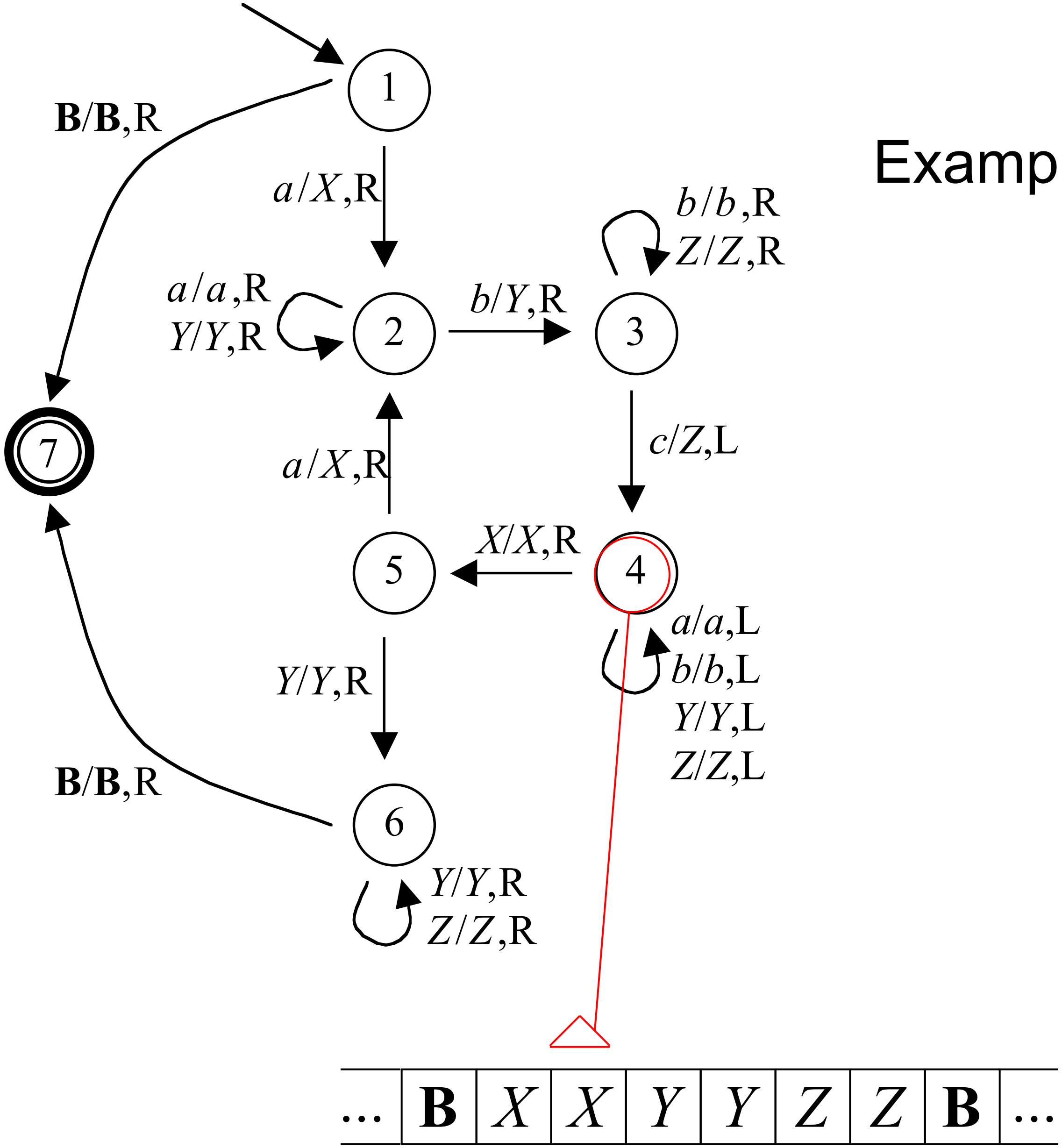


Example: *aabbcc*

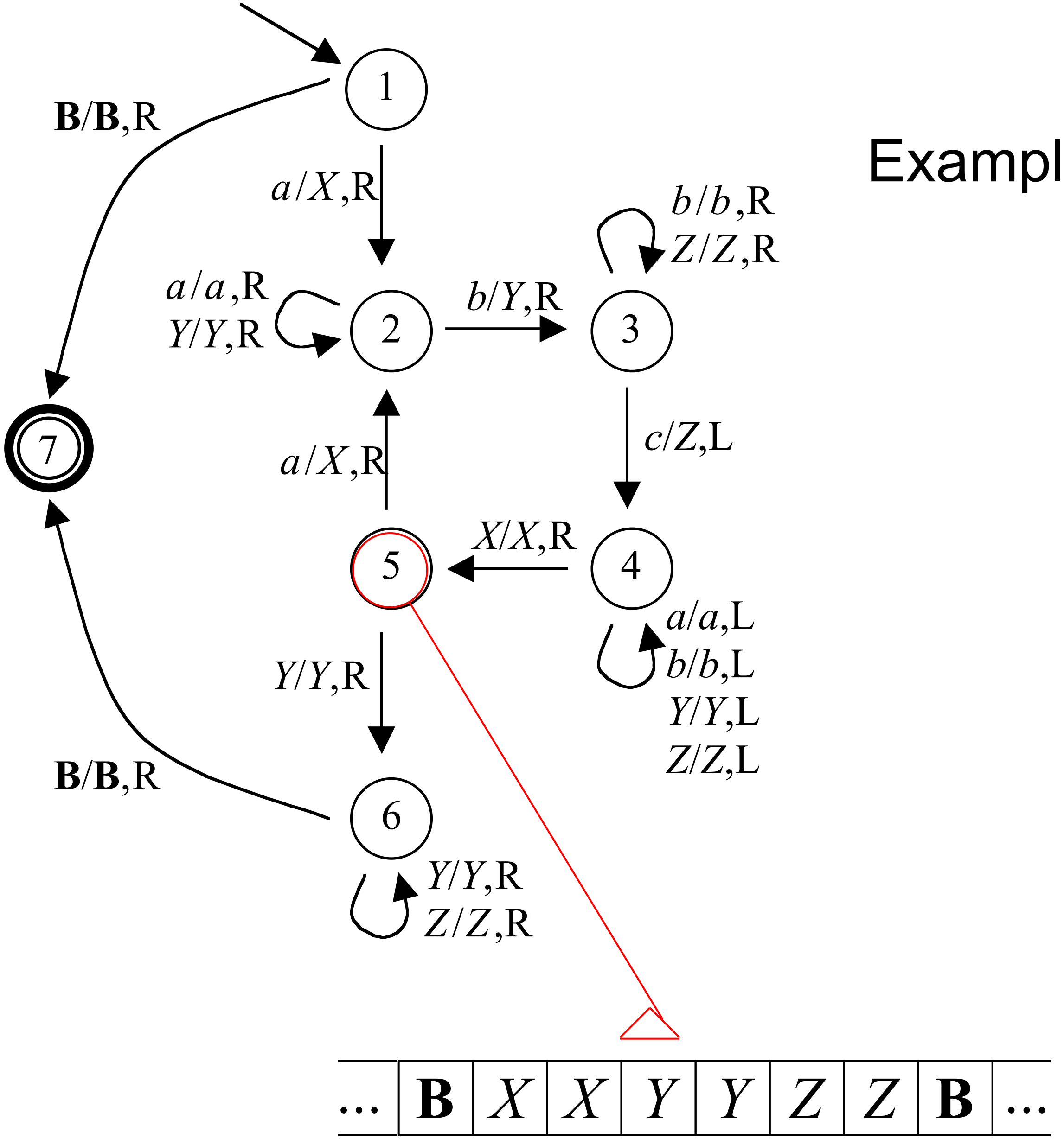


…	**B**	*X*	*X*	*Y*	*Y*	*Z*	*Z*	**B**	…

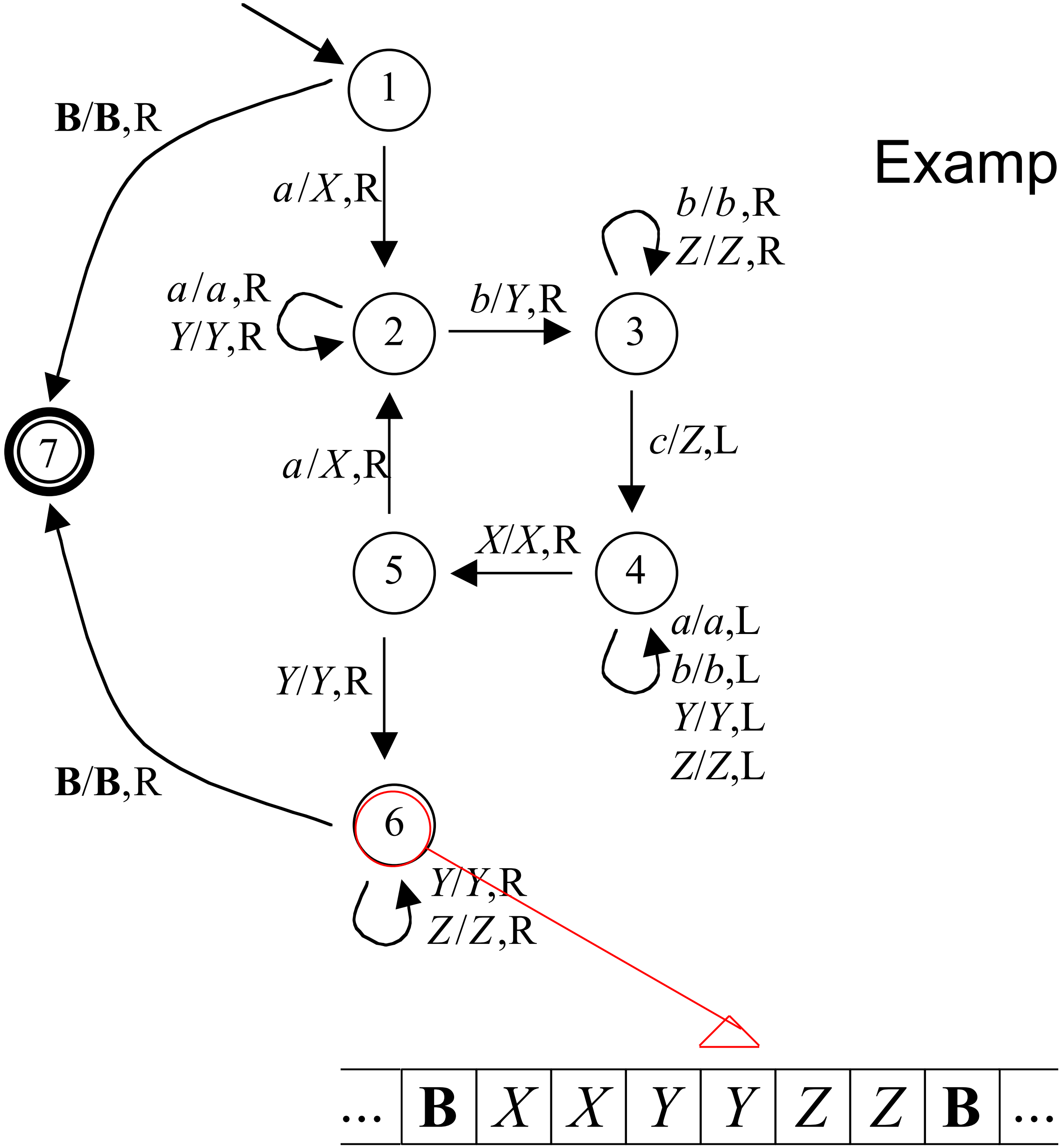
Example: *aabbcc*

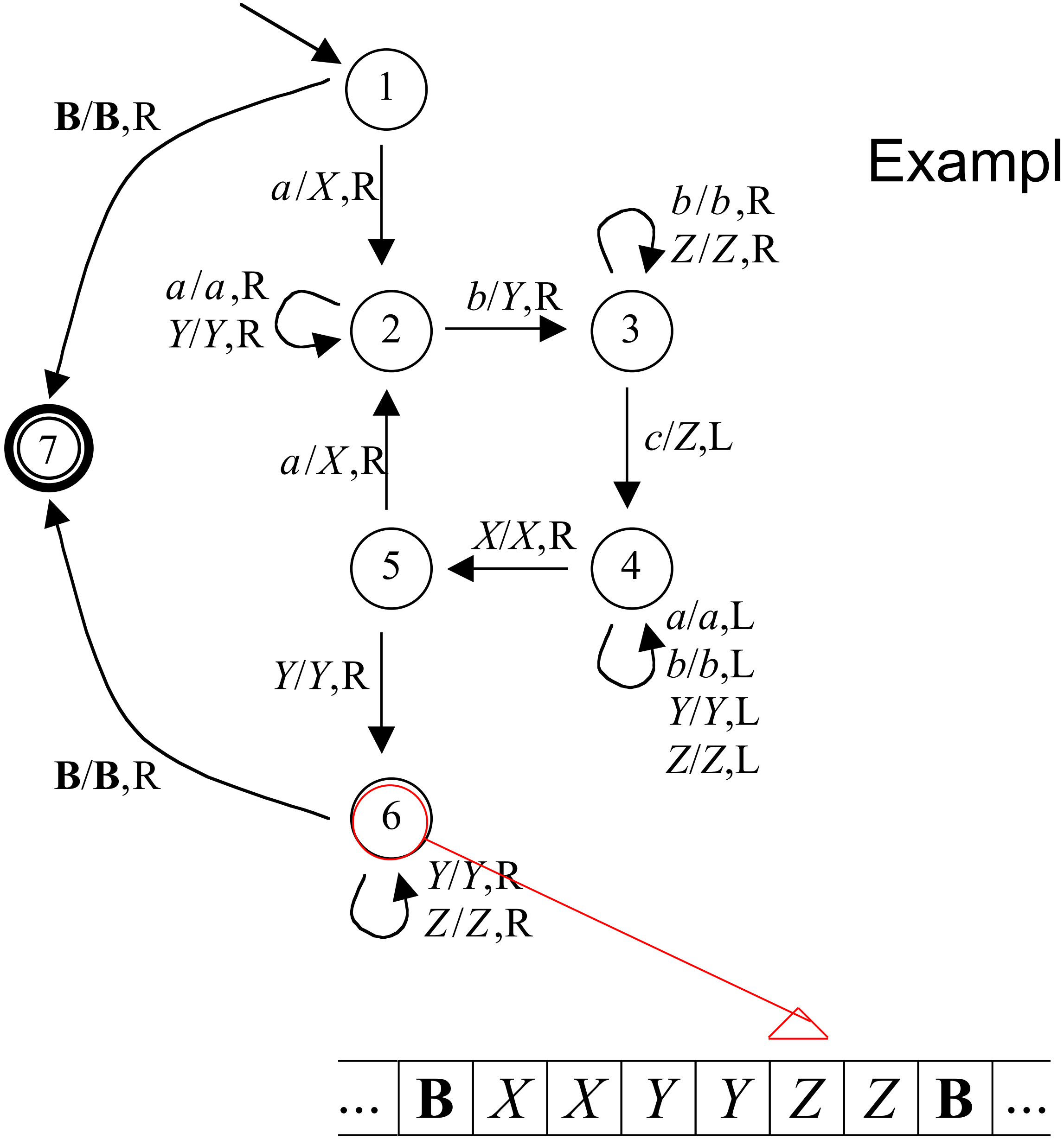


Example: *aabbcc*



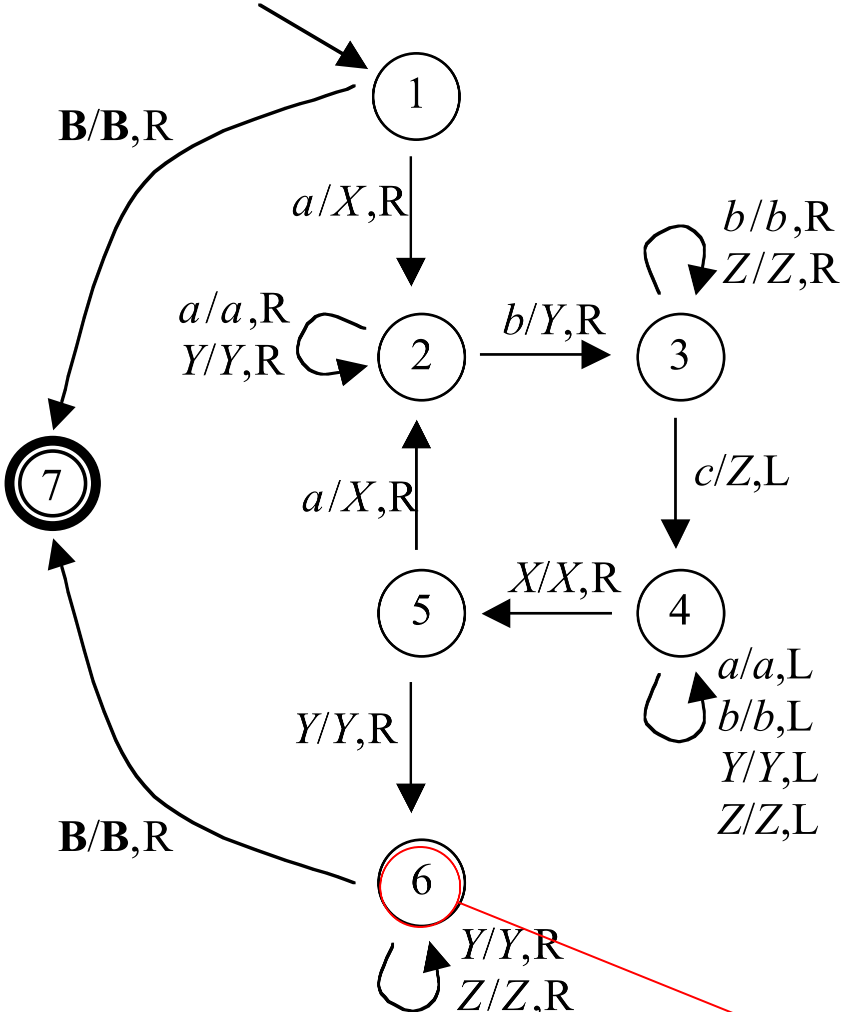
Example: *aabbcc*

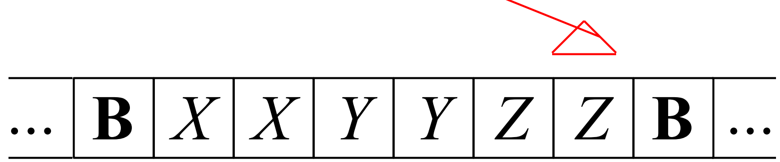




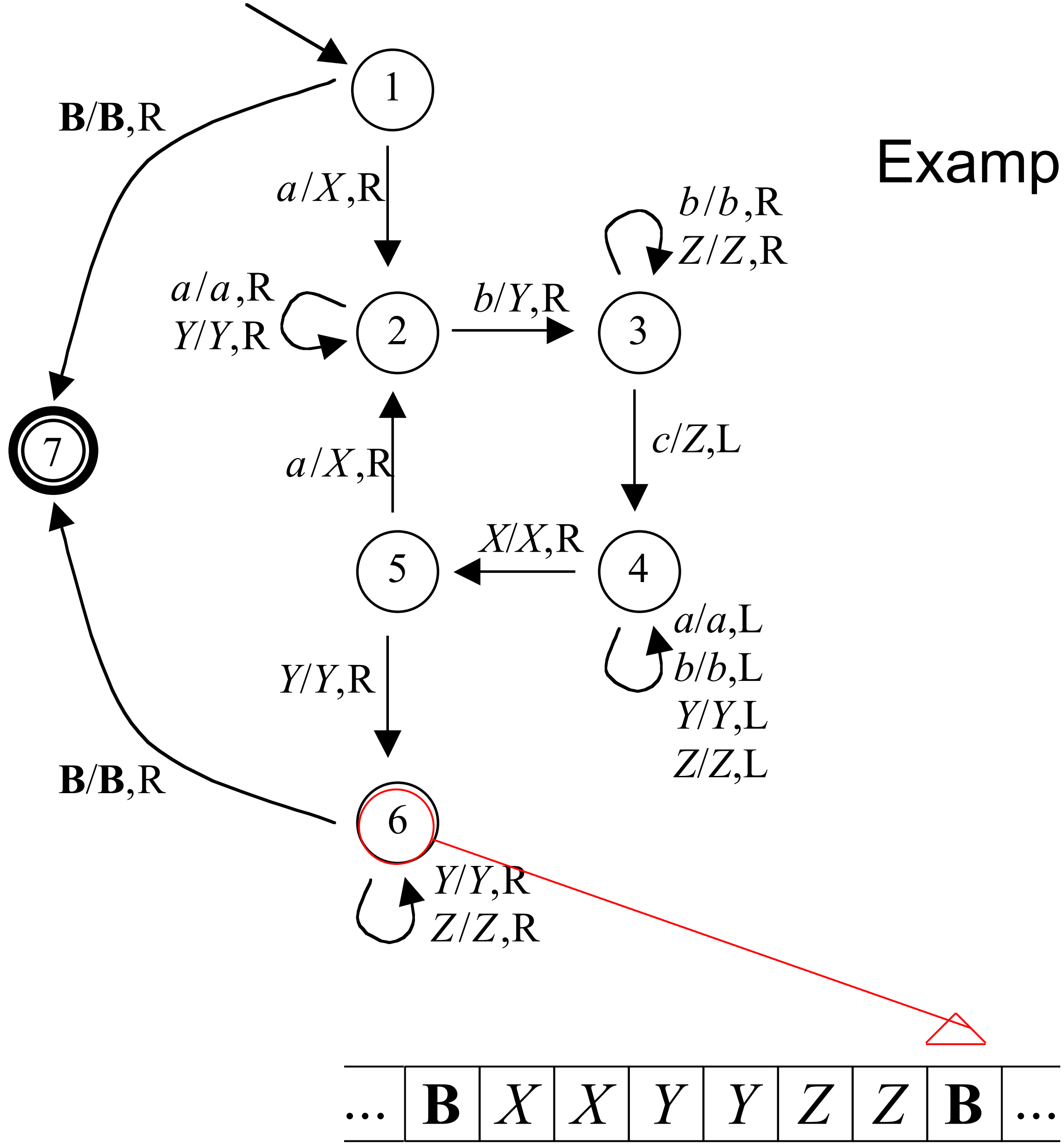
Example: *aabbcc*

Example: *aabbcc*

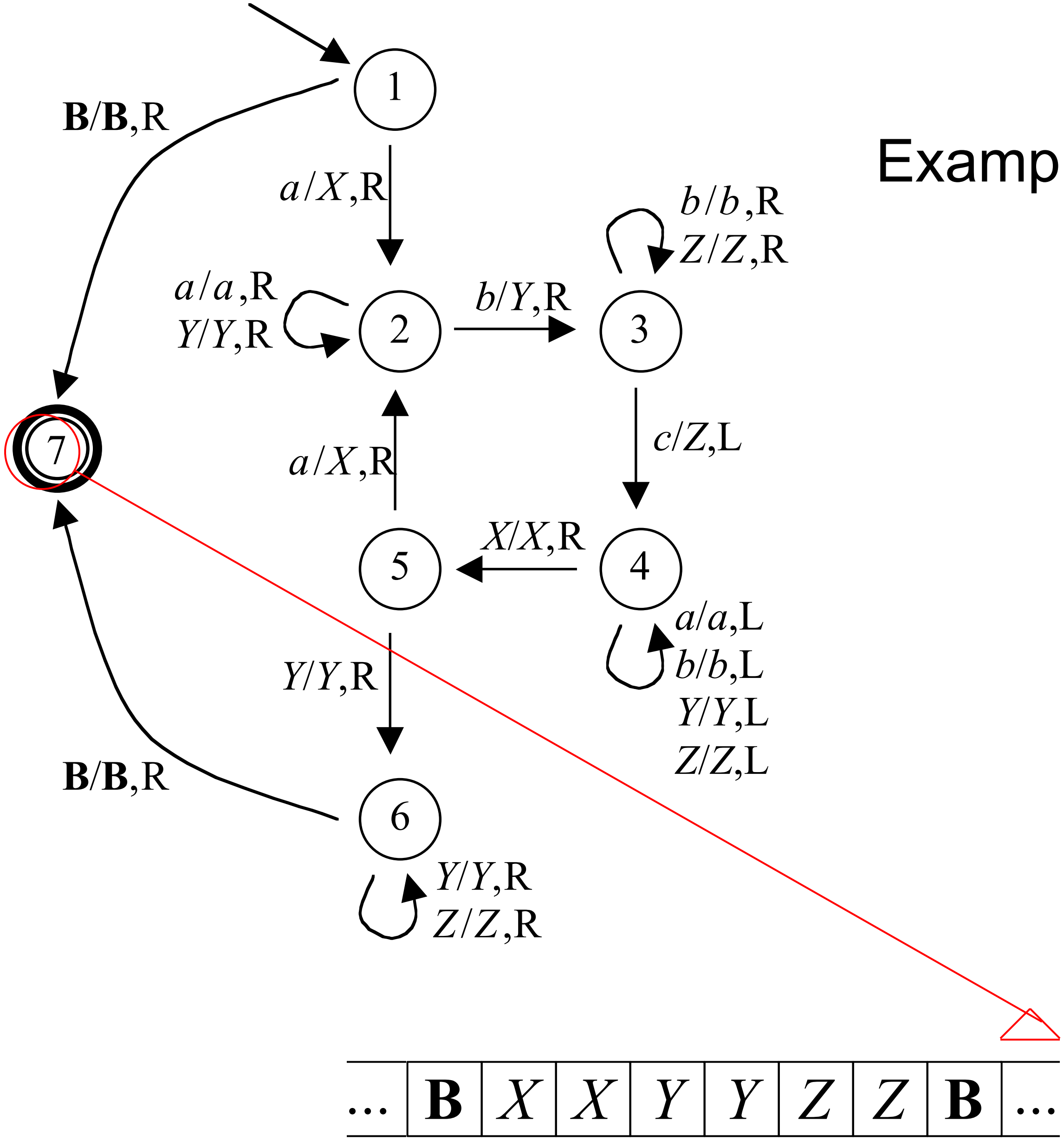




Example: *aabbcc*



Example: *aabbcc*



Outline

- 16.1 Turing Machine Basics
- 16.2 Simple TMs
- 16.3 A TM for $\{a^n b^n c^n\}$
- 16.4 The 7-Tuple
- 16.5 The Languages Defined By A TM
- 16.6 To Halt Or Not To Halt
- 16.7 A TM for $\{xcx \mid x \in \{a,b\}^*\}$
- 16.8 Three Tapes
- 16.9 Simulating DFAs
- 16.10 Simulating Other Automata

The 7-Tuple

- A TM M is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, \mathbf{B}, q_0, F)$:
  - Q is the finite set of states
  - Σ is the input alphabet
  - Γ is the tape alphabet, with $\Sigma \subset \Gamma$ and $Q \cap \Gamma = \{\}$
  - $\delta \in (Q \times \Gamma \to Q \times \Gamma \times \{L,R\})$ is the transition function
  - $\mathbf{B}$ is the blank symbol, $\mathbf{B} \in \Gamma$, $\mathbf{B} \notin \Sigma$
  - $q_0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of accepting states

The Transition Function

- $\delta \in (Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\})$
- That is, $\delta(q,X) = (p,Y,D)$, where
  - Input q is the current state
  - Input X is the symbol at the current head position
  - Output p is the next state
  - Output Y is the symbol to write at the current head position (over the X)
  - Output D is a direction, L or R, to move the head
- δ is deterministic: at most one move from each configuration
- δ need not be defined over its whole domain, so there may be some q and X with no move $\delta(q,X)$

Outline

- 16.1 Turing Machine Basics
- 16.2 Simple TMs
- 16.3 A TM for $\{a^n b^n c^n\}$
- 16.4 The 7-Tuple
- 16.5 The Languages Defined By A TM
- 16.6 To Halt Or Not To Halt
- 16.7 A TM for $\{xcx \mid x \in \{a,b\}^*\}$
- 16.8 Three Tapes
- 16.9 Simulating DFAs
- 16.10 Simulating Other Automata

TM Instantaneous Description

- An instantaneous description (ID) for a TM is a string *xqy*:
  - $x \in \Gamma^*$ represents the tape to the left of the head
  - *q* is the current state
  - $y \in \Gamma^*$ represents the tape at and to the right of the head
- ID strings are normalized as follows:
  - In *x*, leading **B**s are omitted, except when the left part is all **B**s, in which case *x* = **B**
  - In *y*, trailing **B**s are omitted, except when the right part is all **B**s, in which case *y* = **B**
  - We define a function *idfix*(*z*) that normalizes an ID string *z* in this way, removing (or adding) leading and trailing **B**s as necessary

A One-Move Relation On IDs

- We will write $I \mapsto J$ if I is an ID and J is an ID that follows from I after one move of the TM
- Technically: $\mapsto$ is a relation on IDs, defined by the δ function for the TM
- Then for any $x \in \Gamma^*$, $c \in \Gamma$, $q \in Q$, $a \in \Gamma$, and $y \in \Gamma^*$, we have two kinds of moves:
  - Left moves: if $\delta(q,a) = (p,b,L)$ then $xcqay \mapsto idfix(xpcby)$
  - Right moves: if $\delta(q,a) = (p,b,R)$ then $xcqay \mapsto idfix(xcbpy)$

Zero-Or-More-Move Relation

- As we did with grammars, NFAs, and stack machines, we extend this to a zero-or-more-move $\mapsto^*$
- Technically, $\mapsto^*$ is a relation on IDs, with $I \mapsto^* J$ if and only if there is a sequence of zero or more relations that starts with I and ends with J
- Note this is reflexive by definition: we always have $I \mapsto^* I$ by a sequence of zero moves

The Language Accepted By A TM

- *idfix*(q_0x) is the initial ID of *M*, given input $x \in \Sigma^*$
- (*idfix*(q_0x) = $\mathbf{B}q_0x$ if $x \neq \varepsilon$, or $\mathbf{B}q_0x\mathbf{B}$ if x = ε)
- Then *x* is accepted if and only if *M* has a sequence of zero or moves from *idfix*(q_0x) to some ID in an accepting state
  - Regardless of what is left on the tape
  - Regardless of the final position of the head
- Technically,

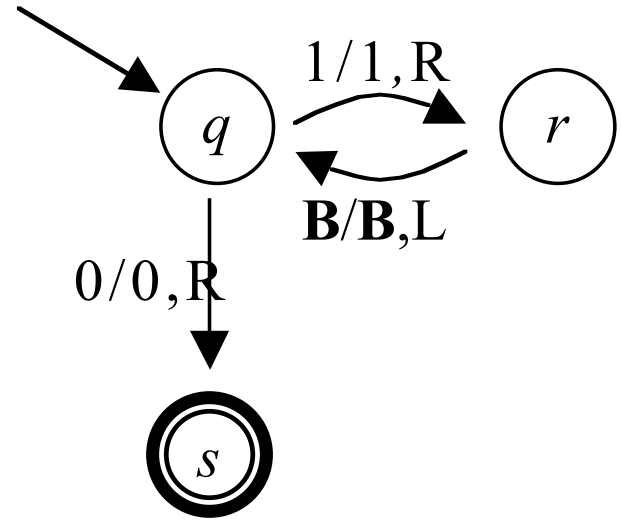
$$L(M) = \{x \in \Sigma^* \mid idfix(q_0x) \mapsto^* ypz \text{ for some } p \in F\}$$

Outline

Three Possible Outcomes

- We've seen that a TM accepts if it ever reaches an accepting state
- We'll say that it halts in that case:
  - It halts and accepts even if there are transitions leaving the accepting state, since they have no effect on $L(M)$
- It also halts if it gets stuck:
  - It halts and rejects if does not reach an accepting state, but gets into an ID I with no J for which $I \mapsto J$
- And there is a third possibility: it runs forever:
  - It always has a move, but never reaches an accepting state

Example:



- M = ($\{q, r, s\}$, {0,1}, {0,1,**B**}, δ, **B**, q, $\{s\}$):
  - $\delta(q,1) = (r,1,\mathrm{R})$
  - $\delta(q,0) = (s,0,\mathrm{R})$
  - $\delta(r,\mathbf{B}) = (q,\mathbf{B},\mathrm{L})$
- Given input 0, M halts and accepts:
  - $\mathbf{B}q0 \mapsto 0s\mathbf{B}$
  - In fact, $L(M) = L(0(0+1)^*)$
- Given input ε, M halts and rejects:
  - $\mathbf{B}q\mathbf{B} \mapsto$?
- Given input 1, M runs forever:
  - $\mathbf{B}q1 \mapsto 1r\mathbf{B} \mapsto \mathbf{B}q1 \mapsto 1r\mathbf{B} \mapsto \mathbf{B}q1 \mapsto 1r\mathbf{B} \mapsto \ldots$

Running Forever

- We can make a TM for $L(0(0+1)^*)$ that halts on all inputs
- In general, though, it is not always possible for TMs to avoid infinite loops, as we will prove in later chapters
- The risk of running forever is the price TMs pay for their great power

Earlier Infinite Loops

- NFAs could in some sense run forever, since they can contain cycles of ε-transitions
- Same with stack machines
- But these were always avoidable:
  - Any NFA could be converted into one without cycles of ε-transitions (in fact, into a DFA, without ε-transitions at all)
  - Similarly, one can show that for any CFL there is a stack machine without cycles of ε-transitions
- For TMs they are not avoidable…

Three-Way Partition

- The three possible TM outcomes partition Σ^* into three subsets
- So instead of just defining $L(M)$, a TM really defines three languages:
  - The language accepted by a TM:
    $L(M) = \{x \in \Sigma^* \mid idfix(q_0x) \mapsto^* ypz \text{ for some } p \in F\}$
  - The language rejected by a TM:
    $R(M) = \{x \in \Sigma^* \mid x \notin L(M) \text{ and there is some ID } I \text{ with } idfix(q_0x) \mapsto^* I \text{ and no } J \text{ with } I \mapsto J\}$
  - The language on which a TM runs forever:
    $F(M) = \{x \in \Sigma^* \mid x \notin L(M) \text{ and } x \notin R(M)\}$

Recursive and RE

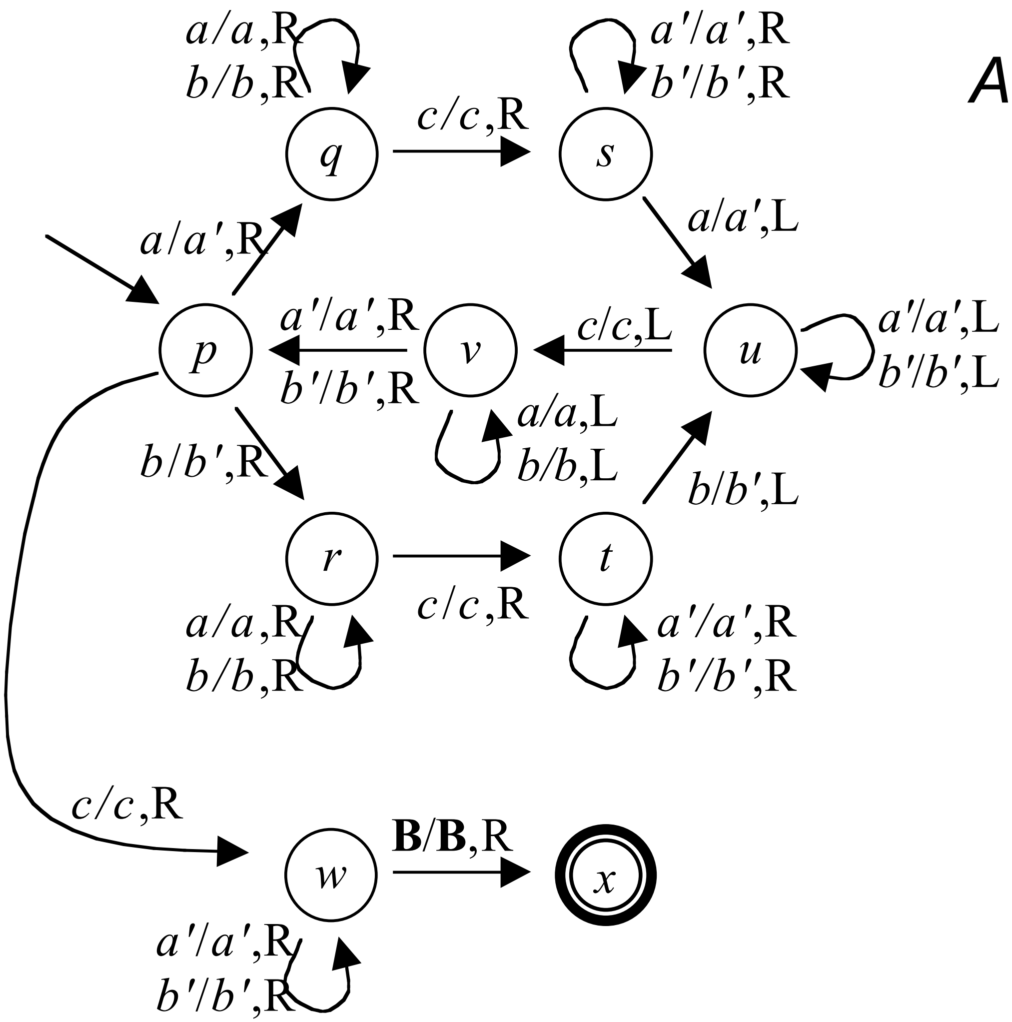
- A TM M is a *total TM* if and only if $F(M) = \{\}$
- A *recursively enumerable* (RE) language is one that is $L(M)$ for some TM M
- A *recursive* language is one that is $L(M)$ for some total TM M
- We will see that these two sets of languages are not the same; some languages are RE but not recursive
- The names are odd, but standard:
  - RE and recursive languages were identified in mathematical studies of computability using recursive function theory

Outline

- 16.1 Turing Machine Basics
- 16.2 Simple TMs
- 16.3 A TM for $\{a^n b^n c^n\}$
- 16.4 The 7-Tuple
- 16.5 The Languages Defined By A TM
- 16.6 To Halt Or Not To Halt
- 16.7 A TM for $\{xcx \mid x \in \{a,b\}^*\}$
- 16.8 Three Tapes
- 16.9 Simulating DFAs
- 16.10 Simulating Other Automata

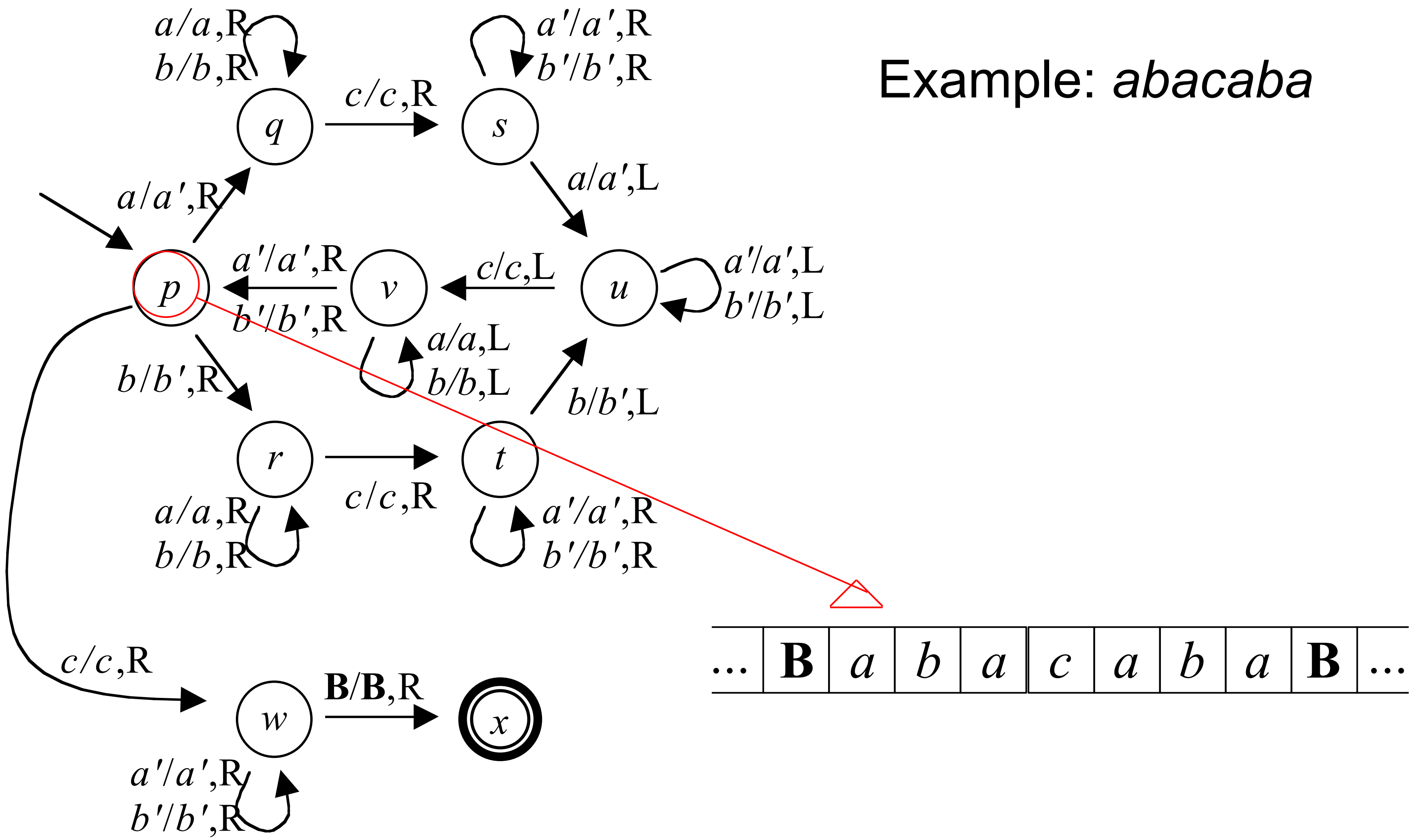
$\{xcx \mid x \in \{a,b\}^*\}$

- Not context-free, as we've seen
- A TM for this language can work by checking each symbol in the first *x* against the corresponding symbol in the second *x*
- To keep track of where it is, it will *mark* those symbols that have already been checked
- That is, it will overwrite them with marked versions of the same symbols, for instance by overwriting *a* with *a'*

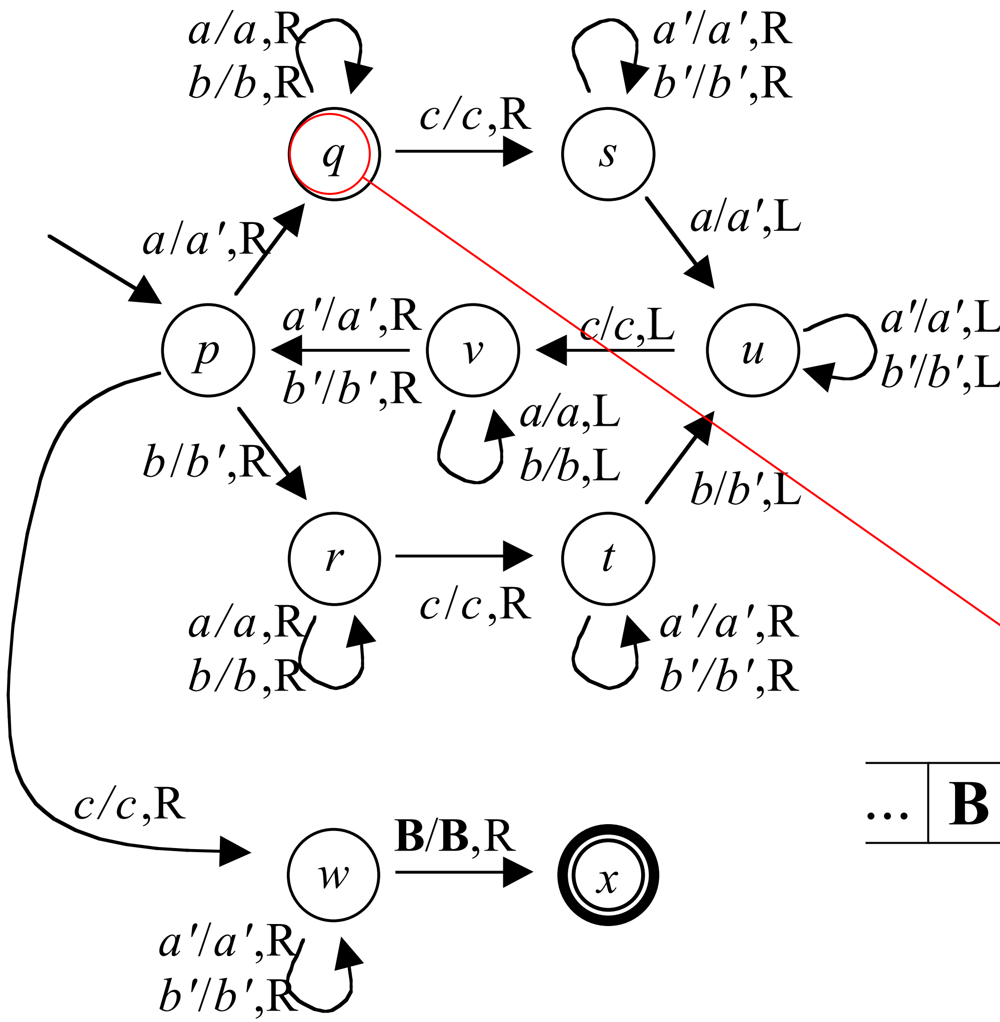


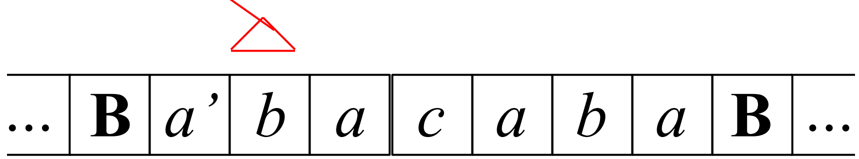
A TM For $\{xcx \mid x \in \{a,b\}^*\}$

Example: *abacaba*

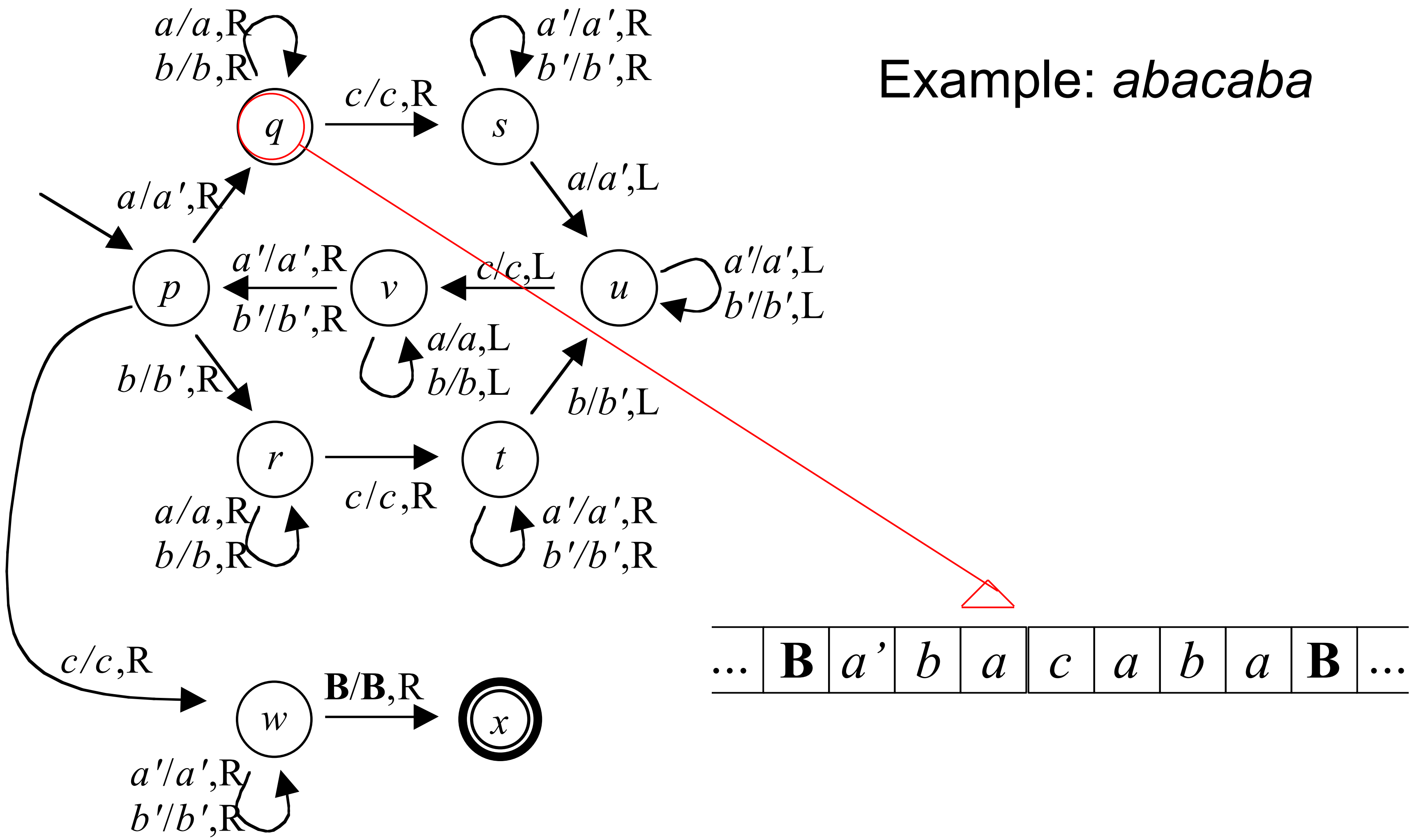


Example: *abacaba*

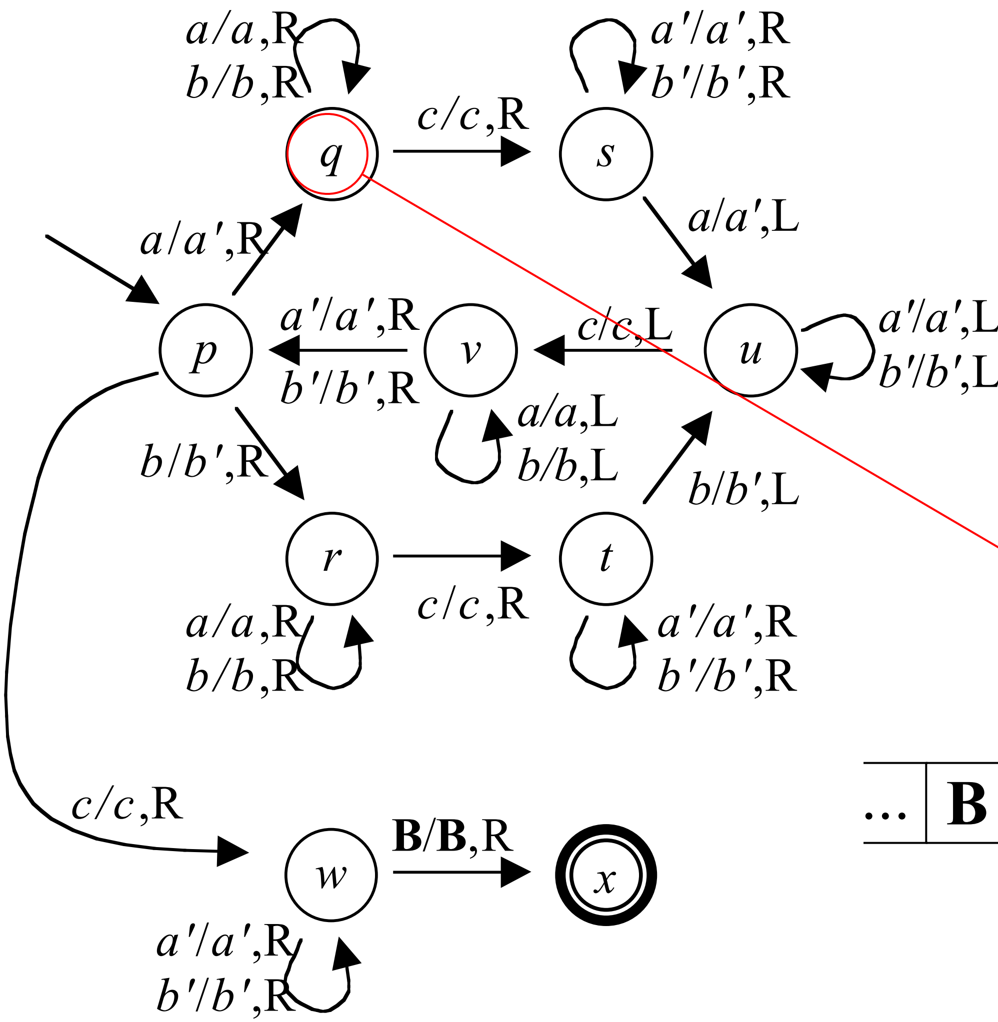


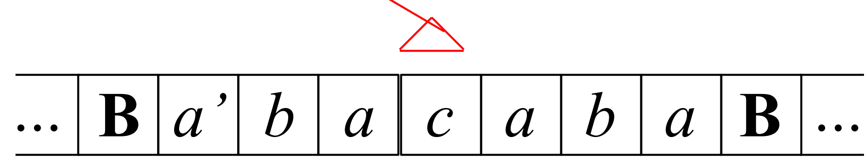


Example: *abacaba*

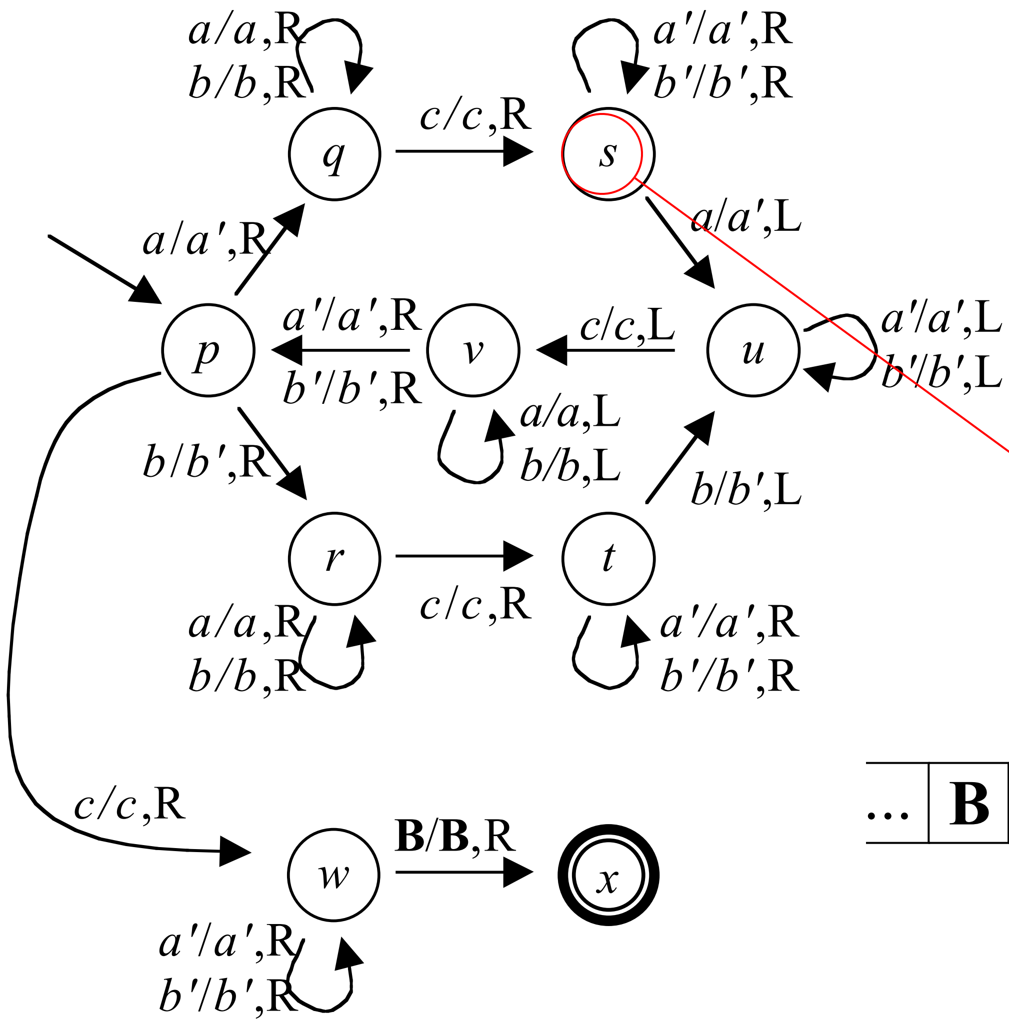


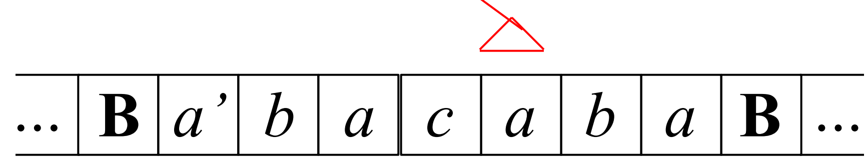
Example: *abacaba*

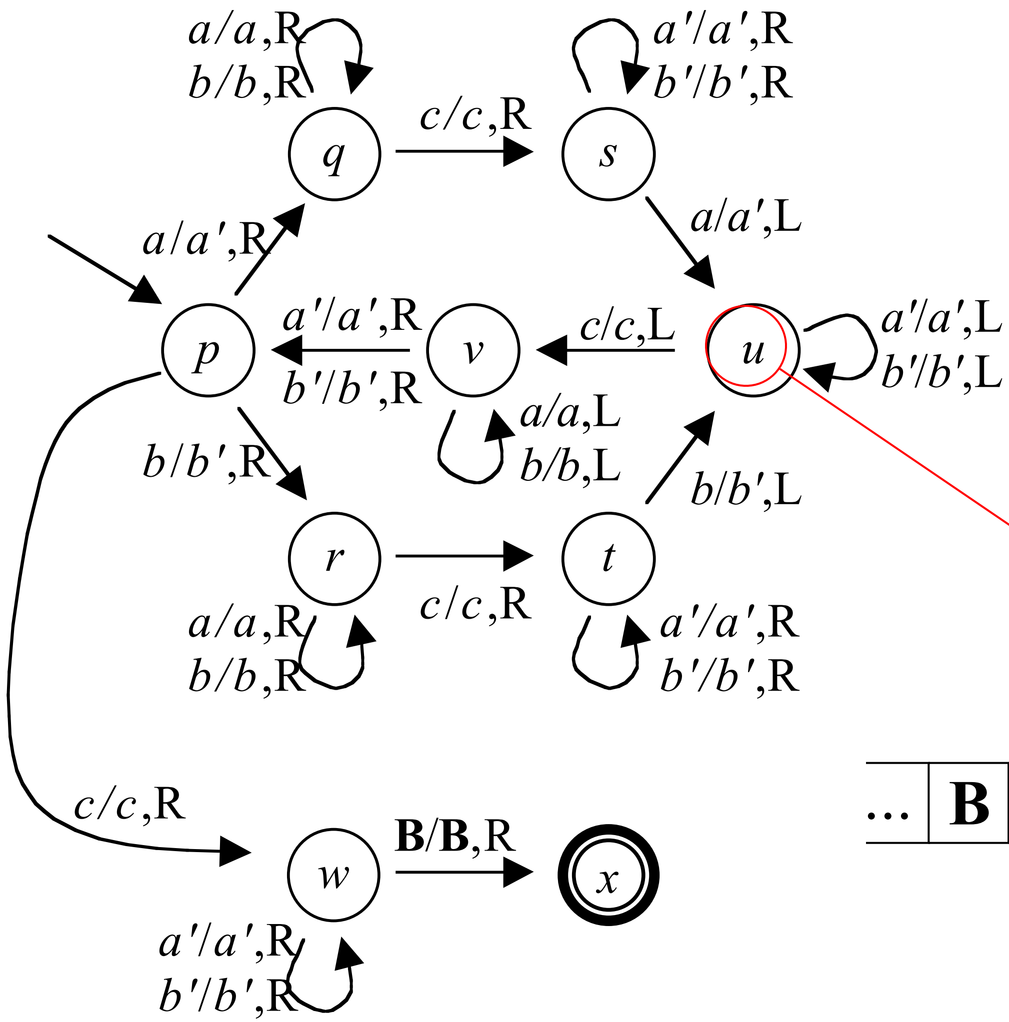




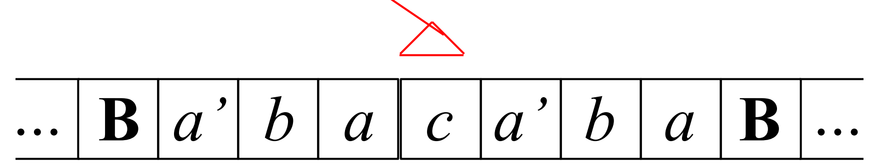
Example: *abacaba*



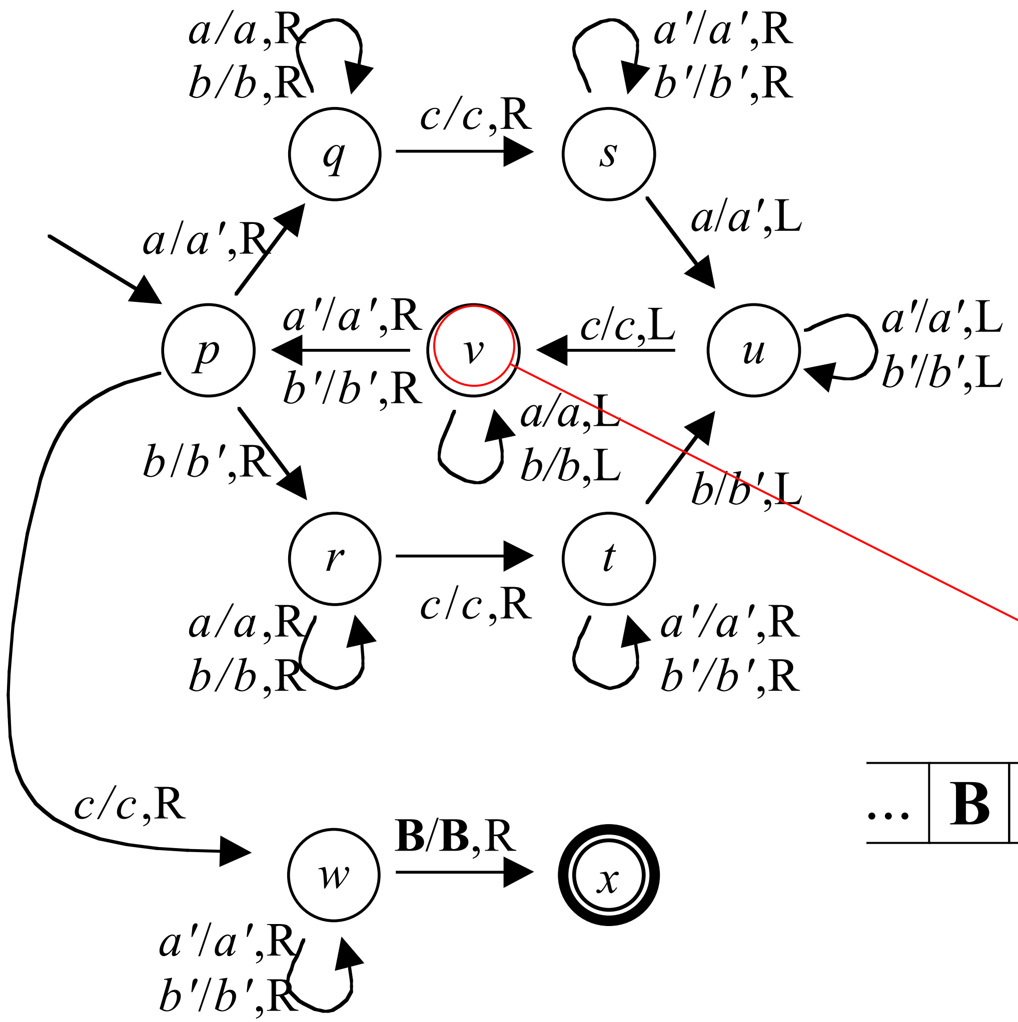


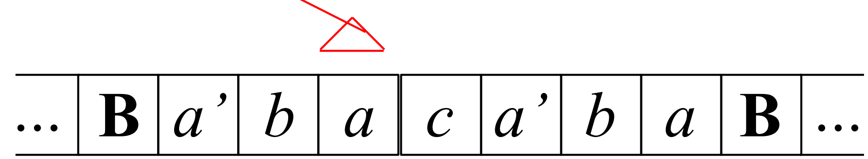


Example: *abacaba*

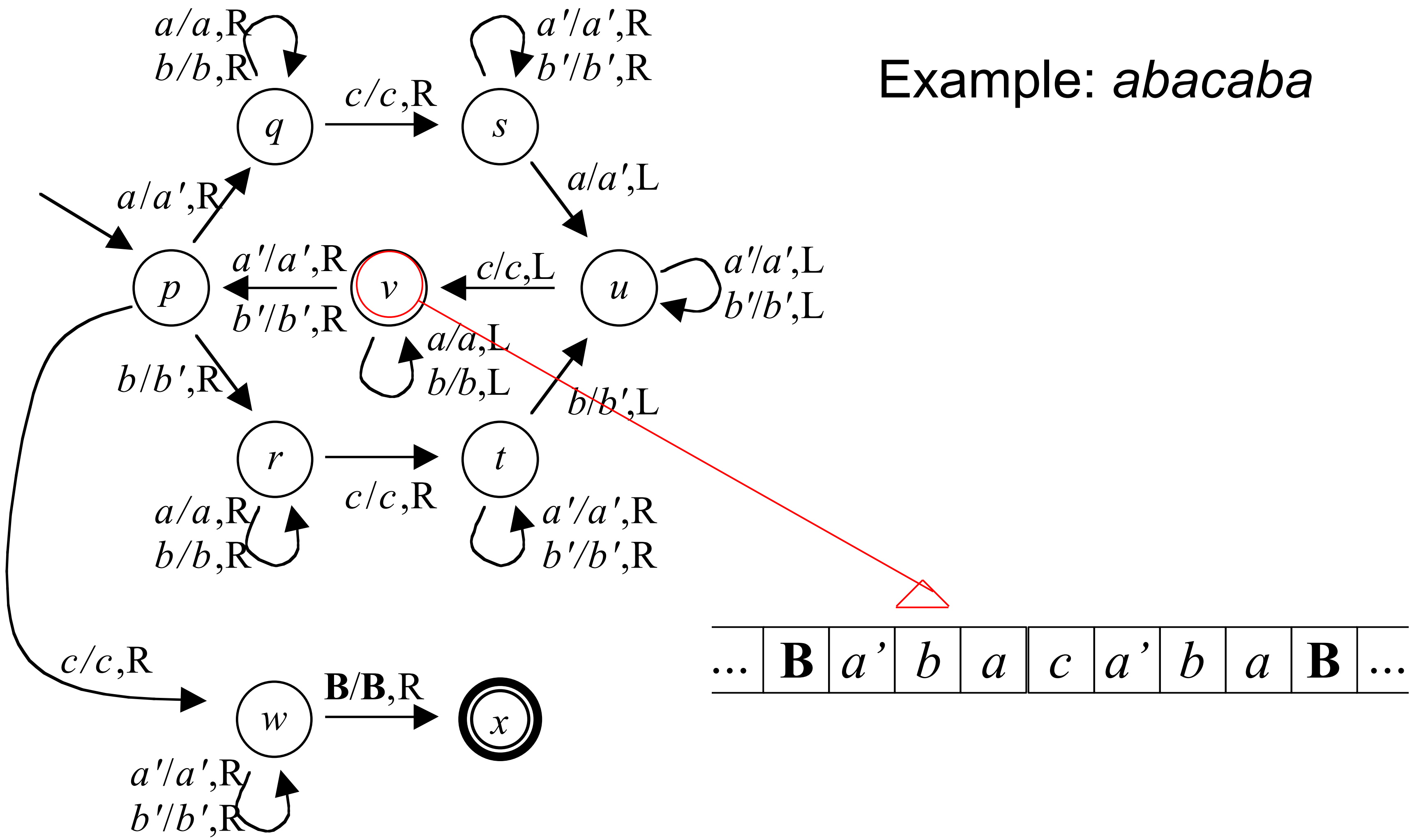


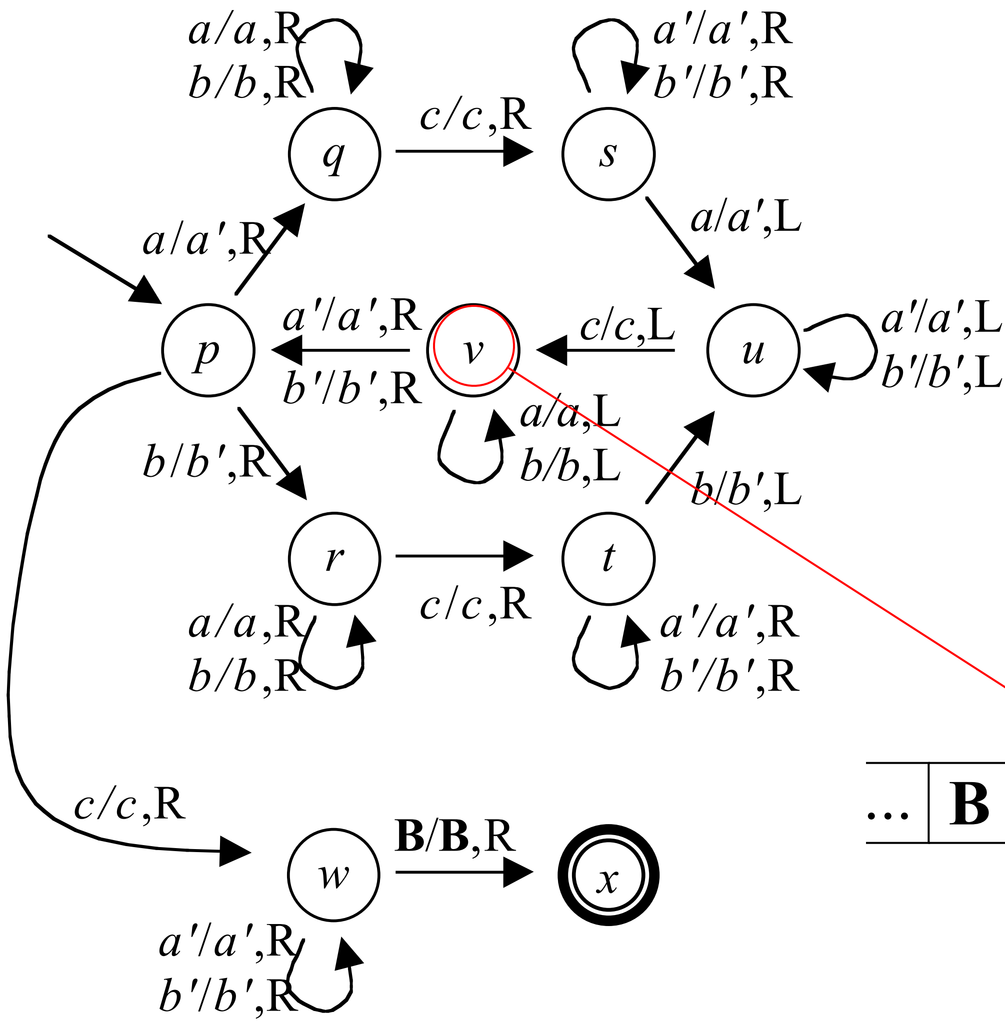
Example: *abacaba*



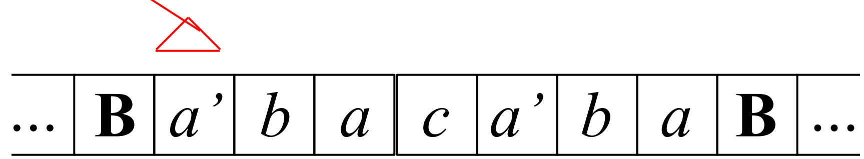


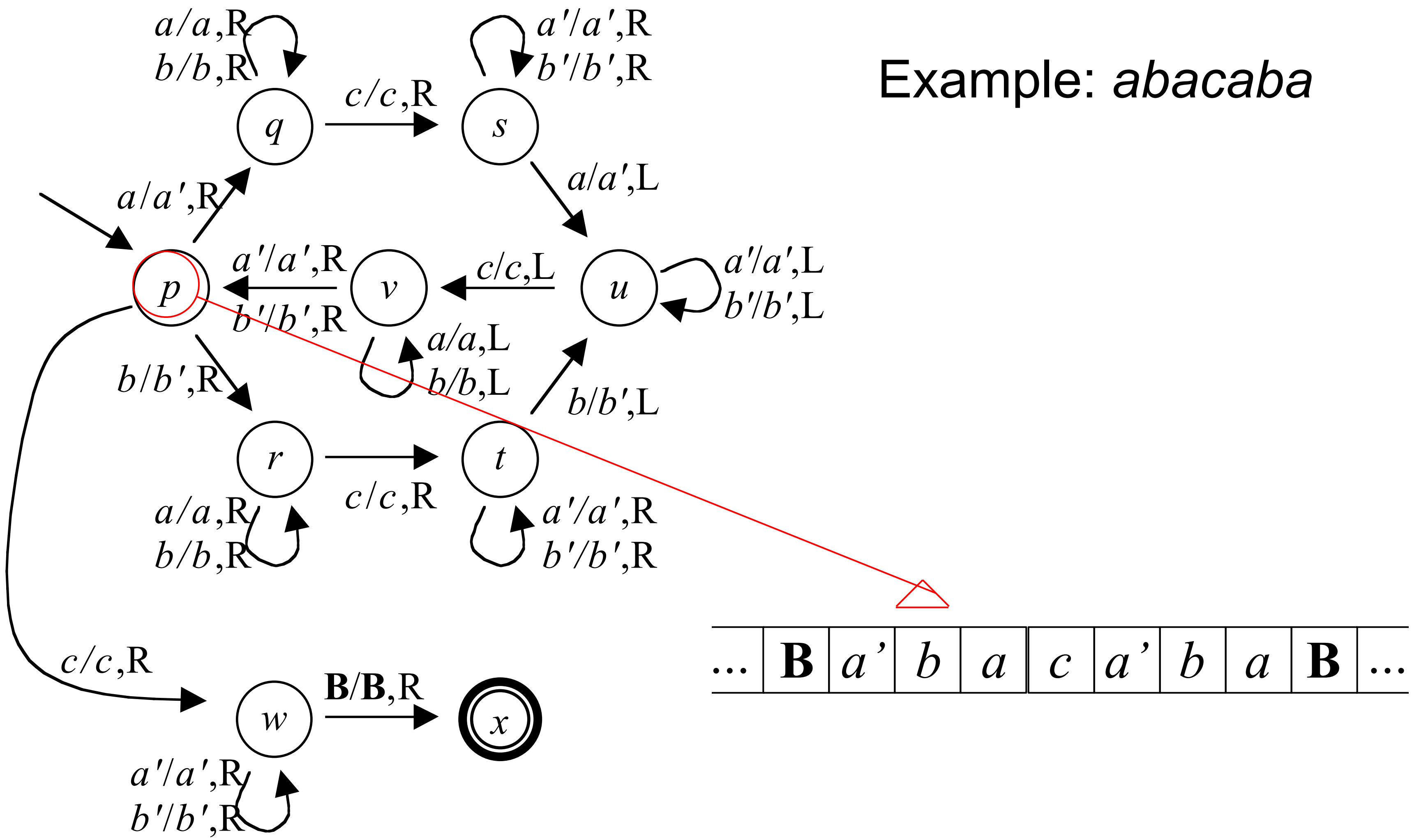
Example: *abacaba*





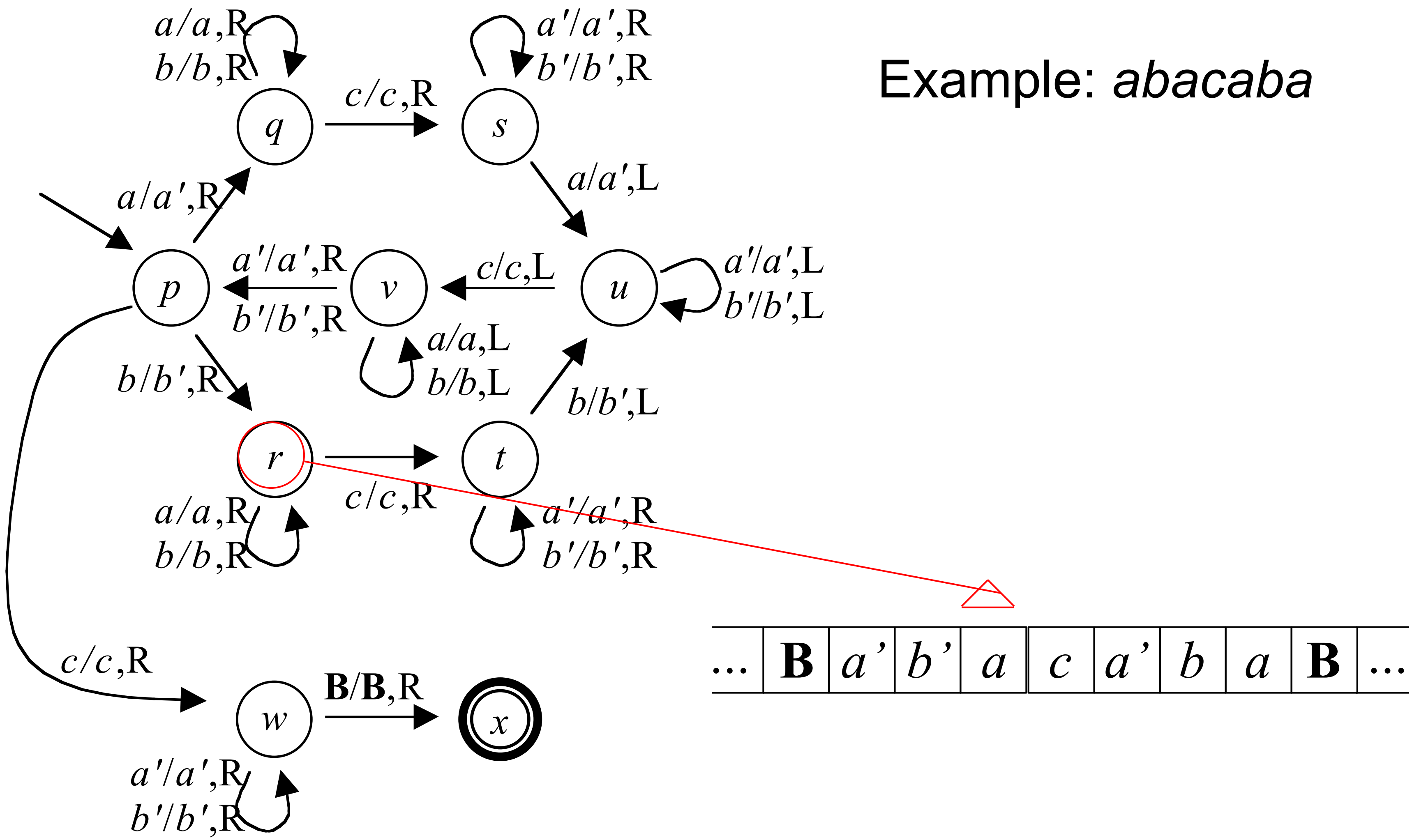
Example: *abacaba*



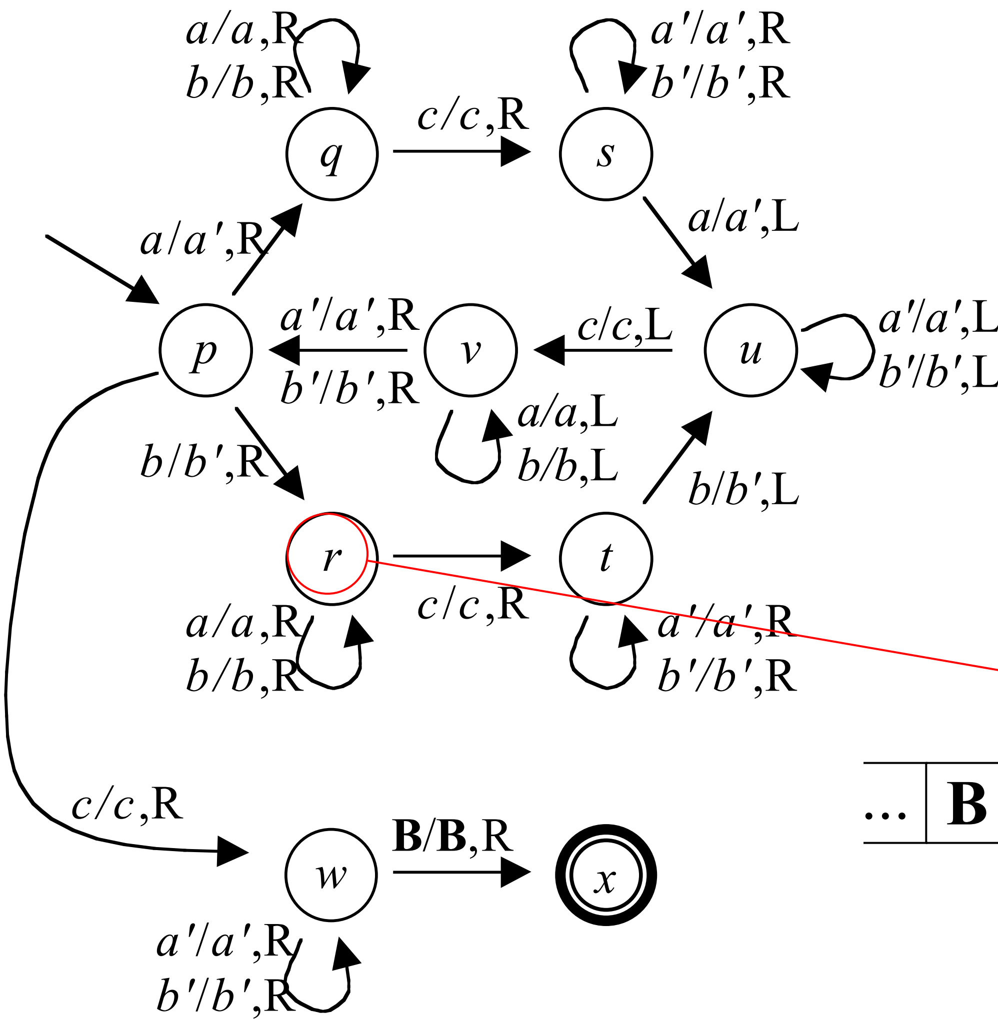


Example: *abacaba*

Example: *abacaba*

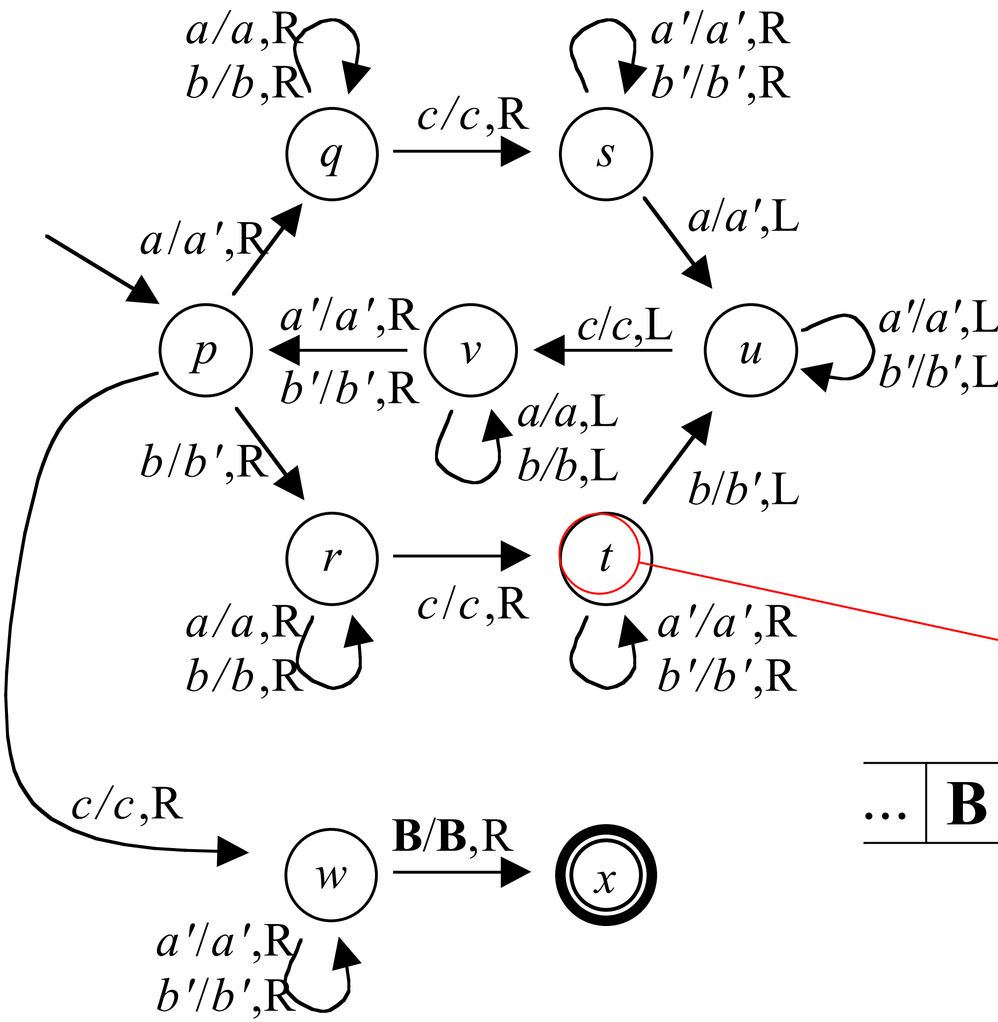


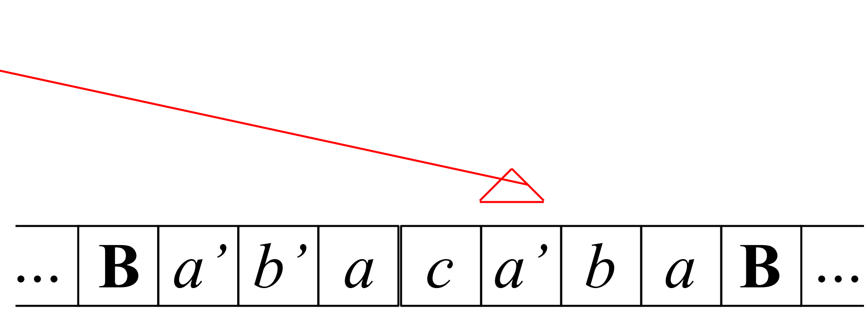
Example: *abacaba*



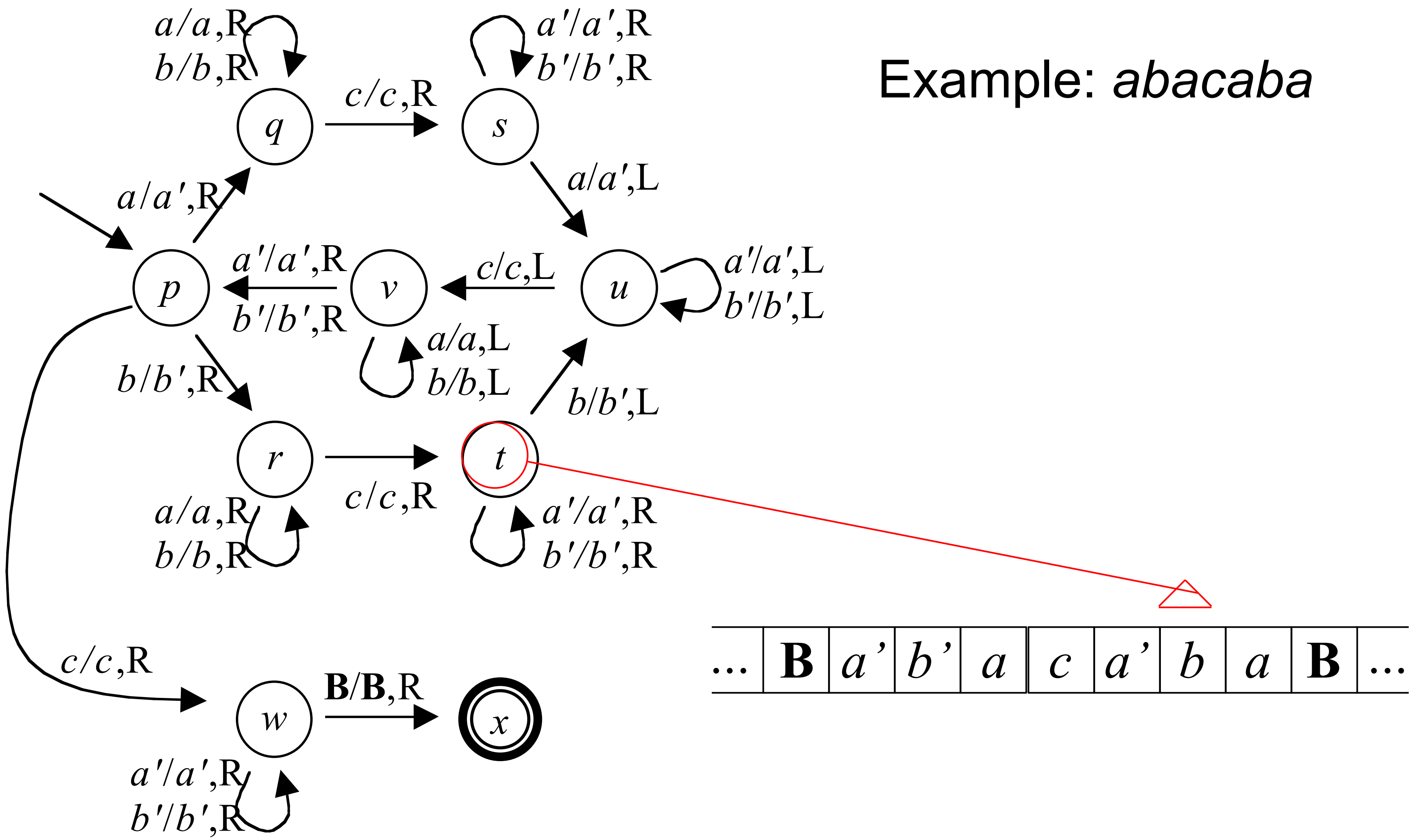
...	**B**	*a'*	*b'*	*a*	*c*	*a'*	*b*	*a*	**B**	...

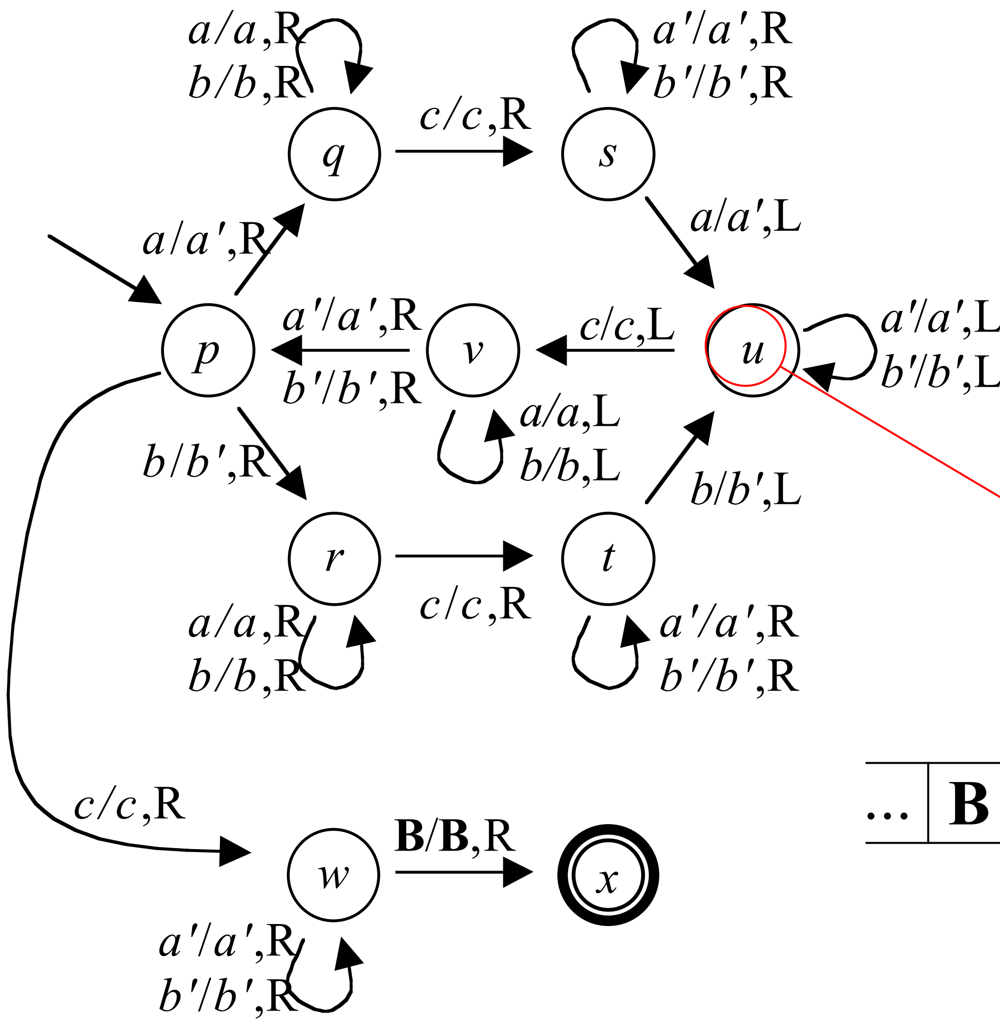
Example: *abacaba*



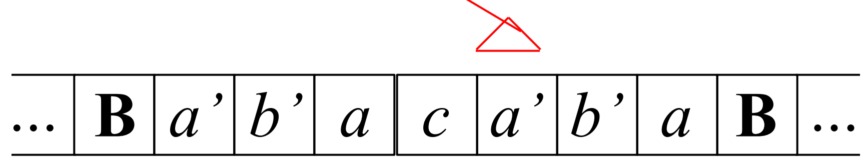


Example: *abacaba*

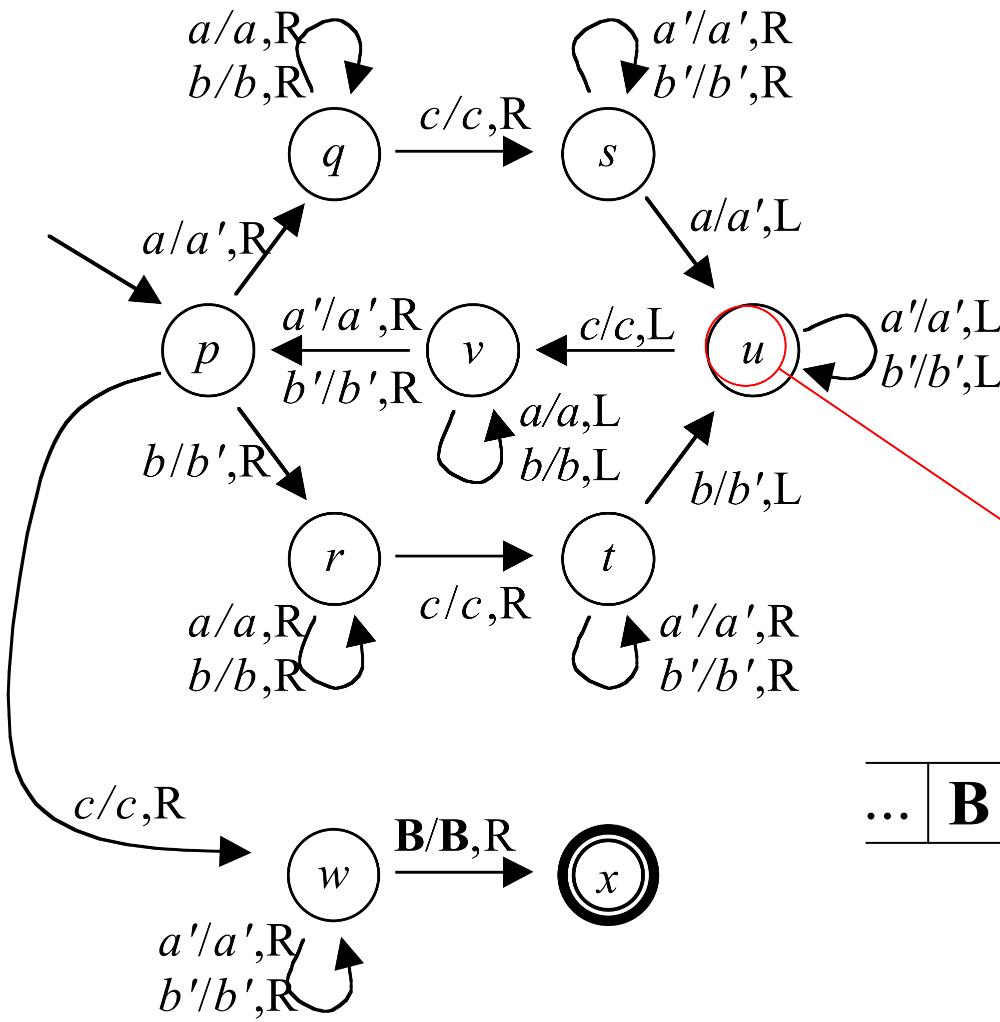


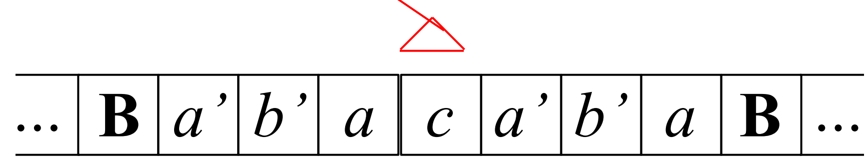


Example: *abacaba*

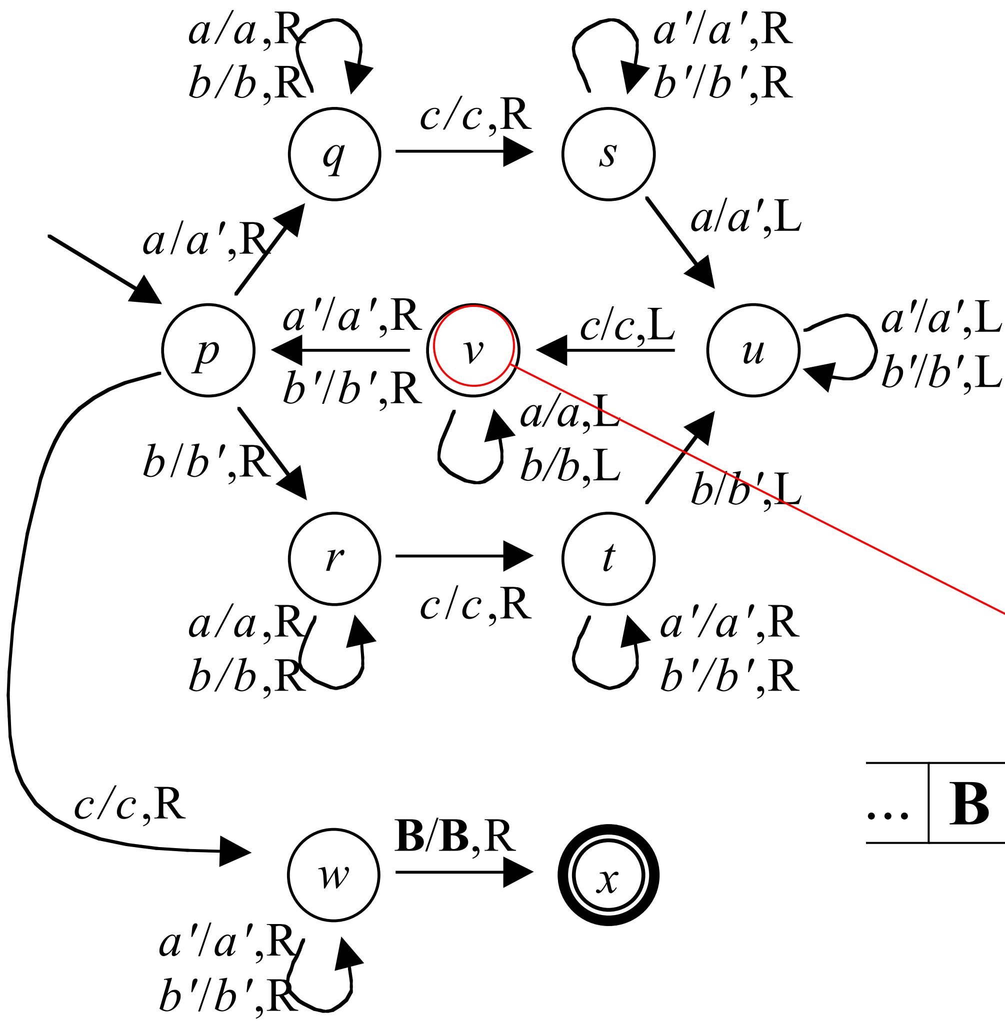


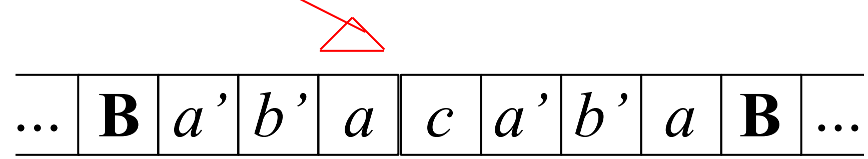
Example: *abacaba*

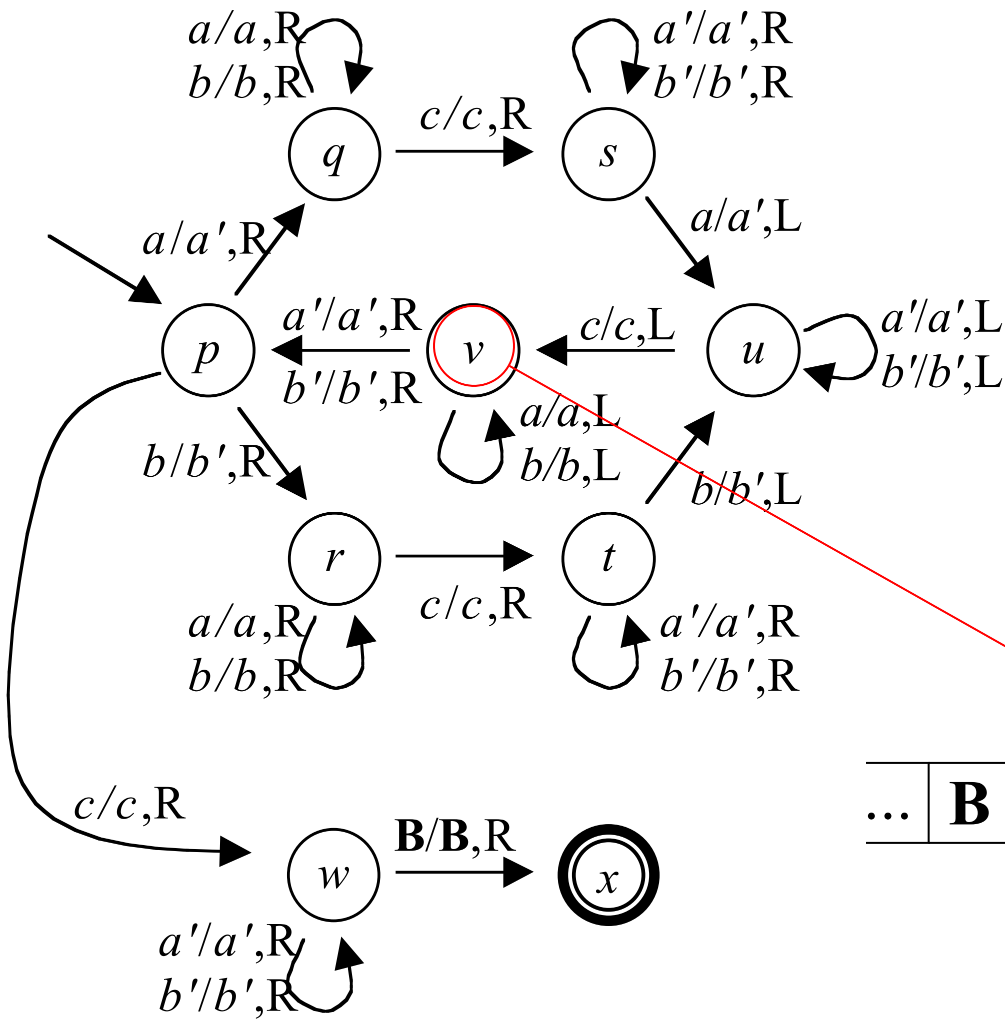




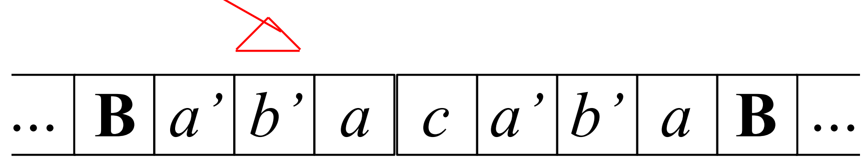
Example: *abacaba*



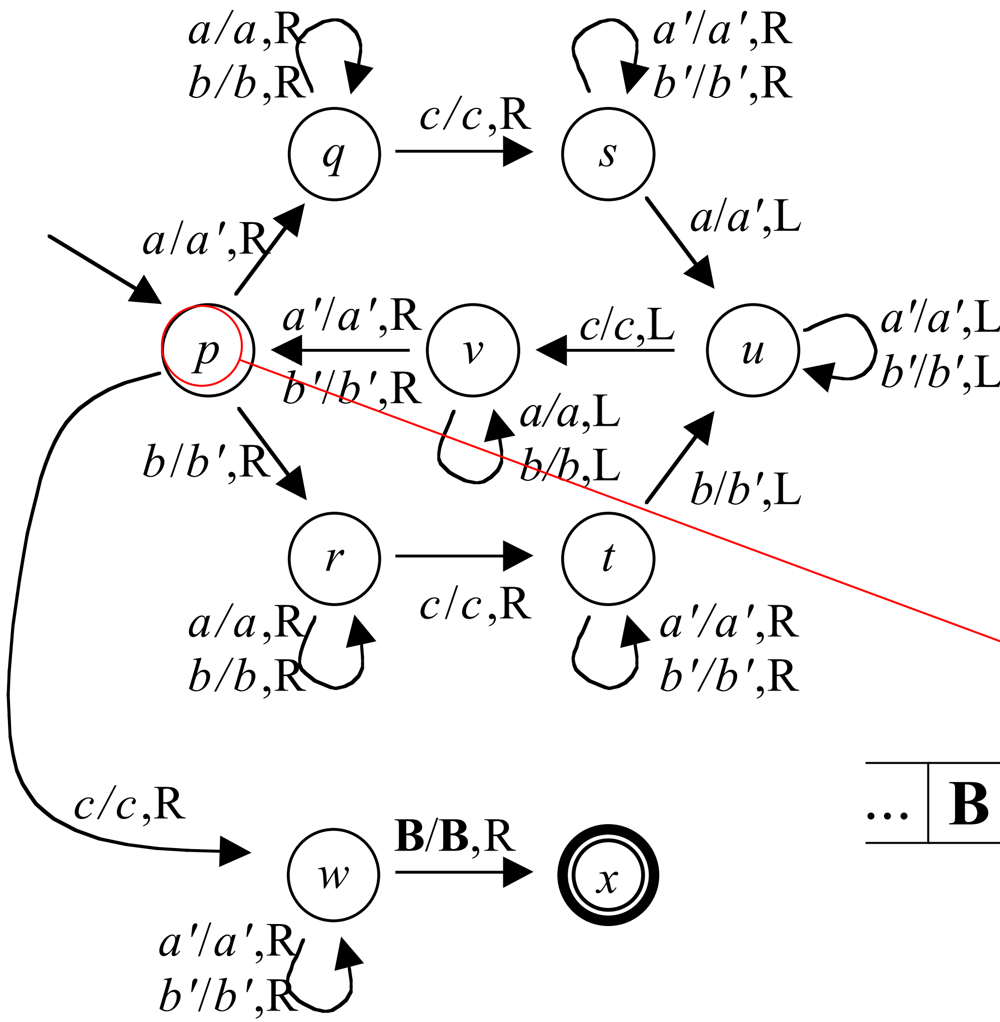




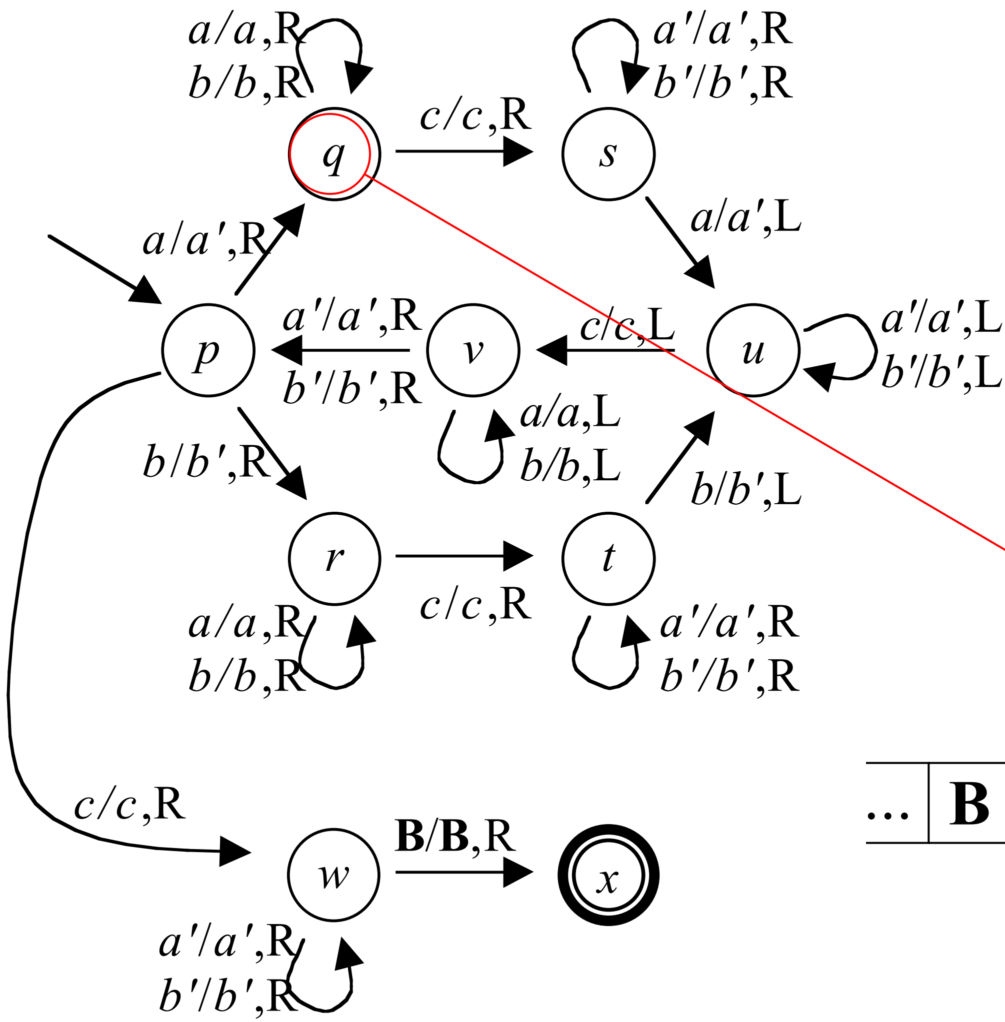
Example: *abacaba*



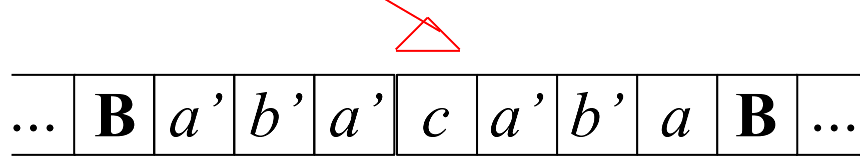
Example: *abacaba*



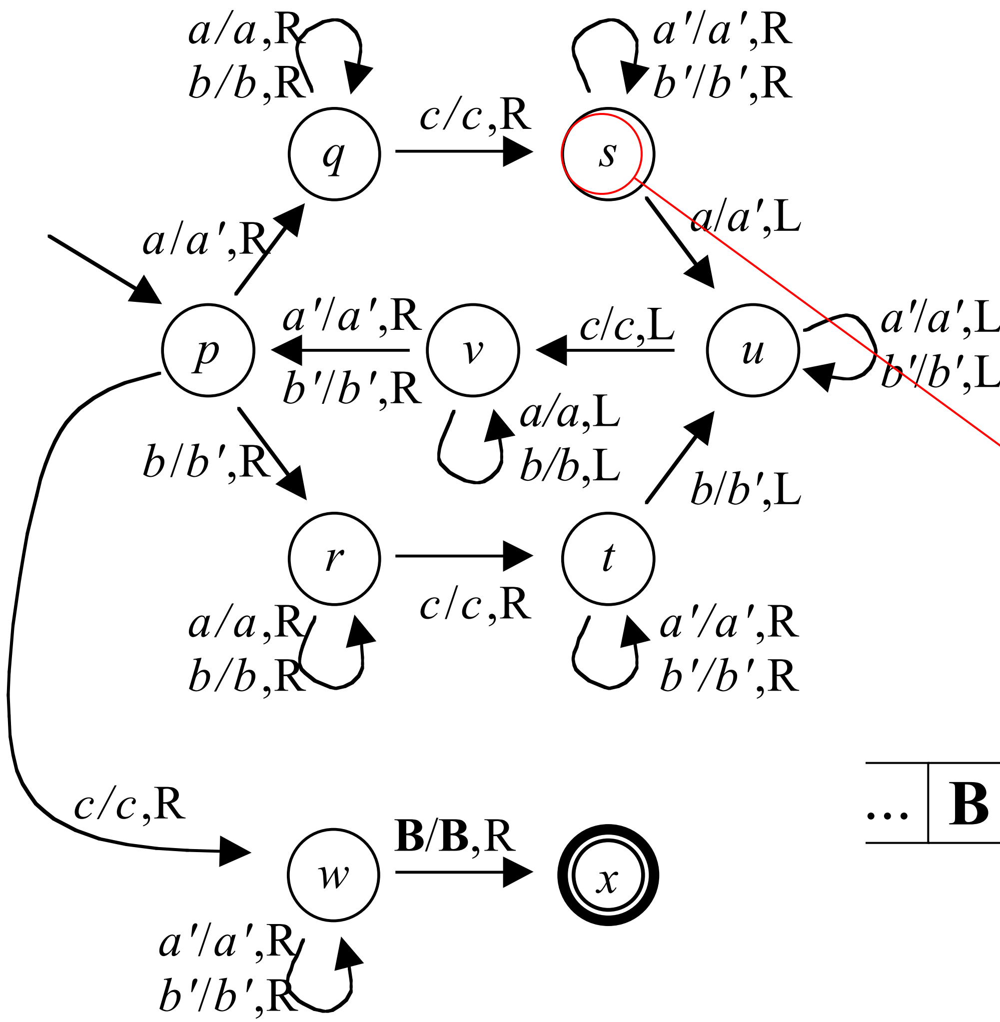
…	**B**	*a'*	*b'*	*a*	*c*	*a'*	*b'*	*a*	**B**	…

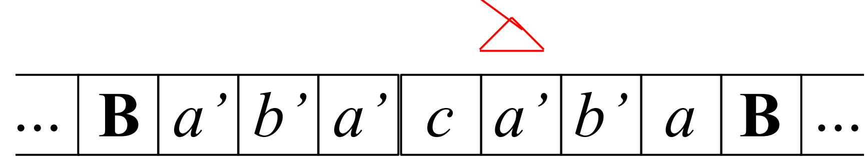


Example: *abacaba*

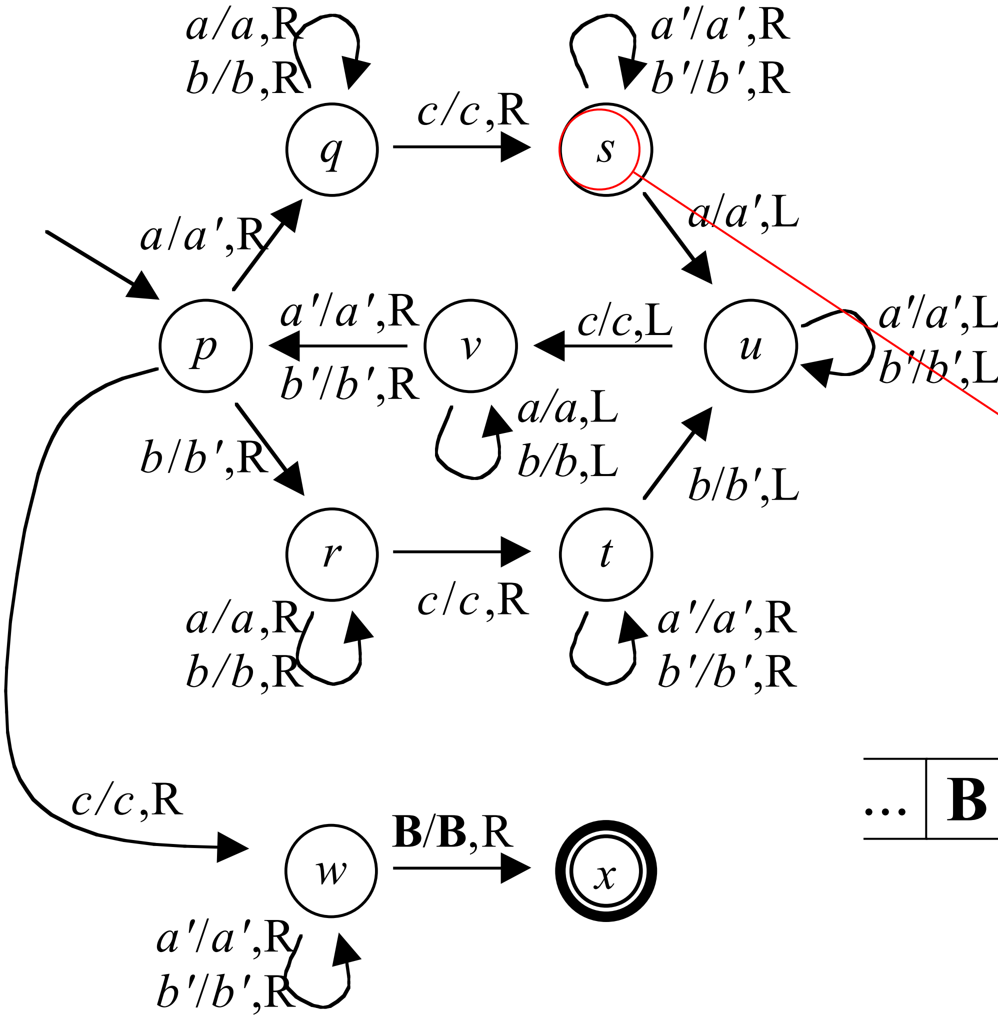


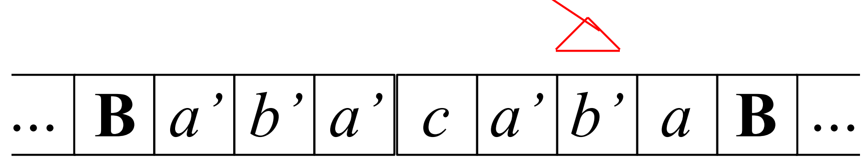
Example: *abacaba*



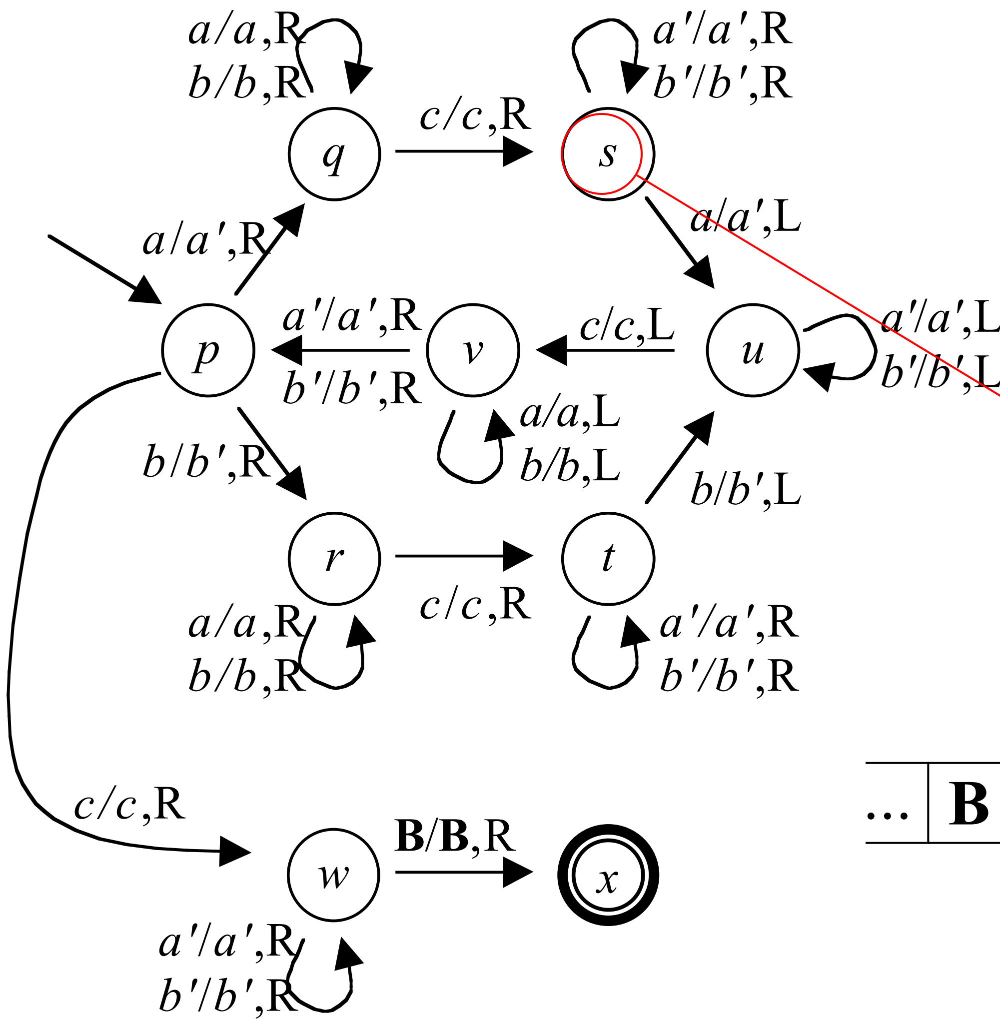


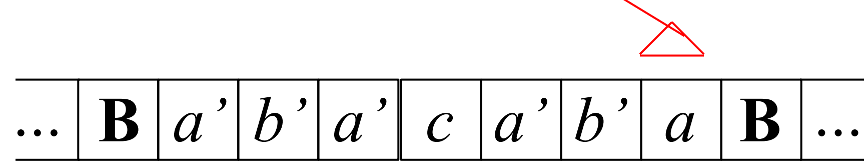
Example: *abacaba*



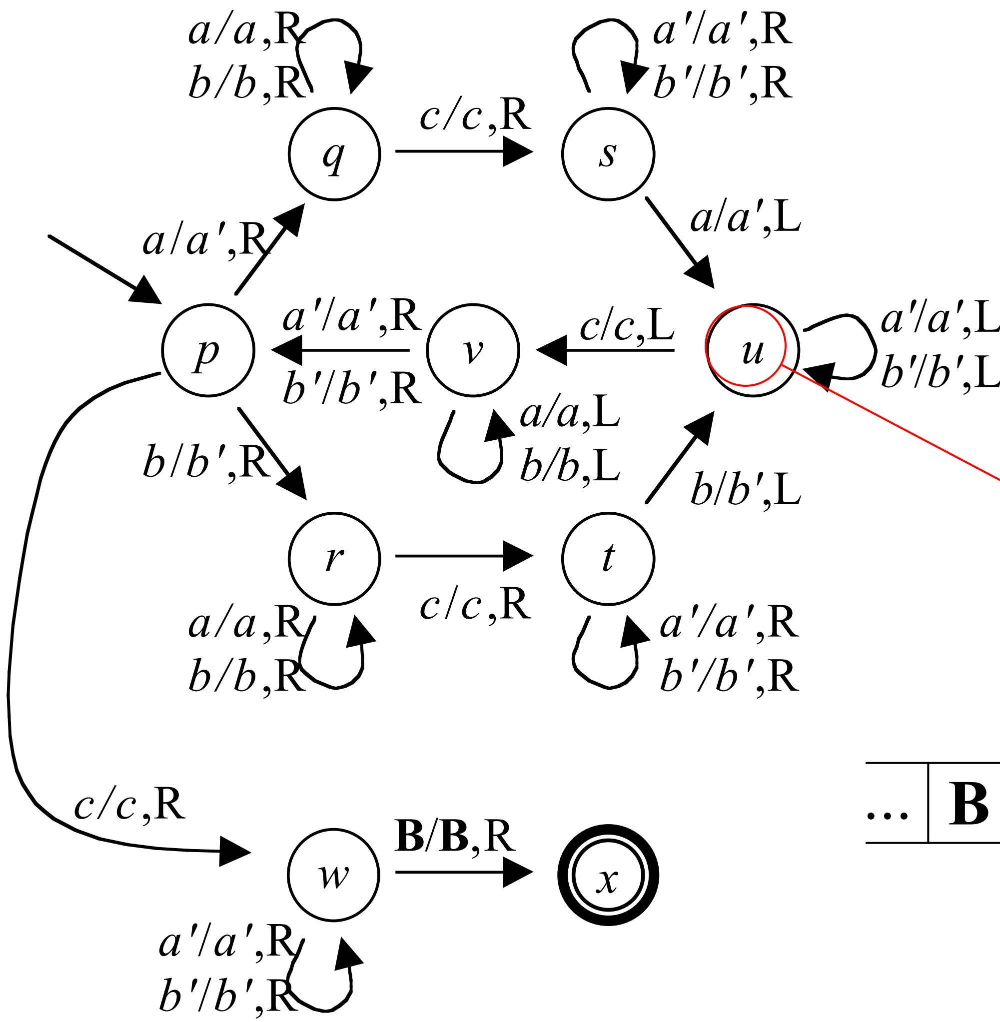


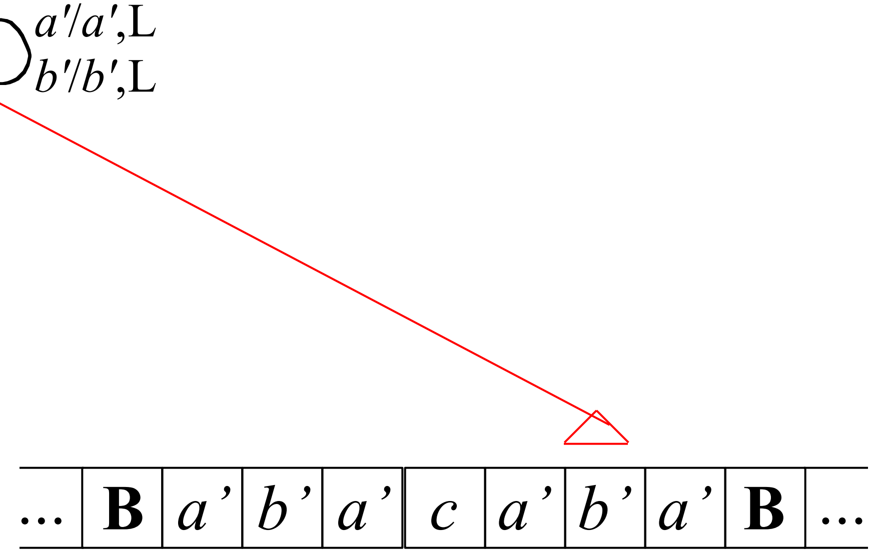
Example: *abacaba*



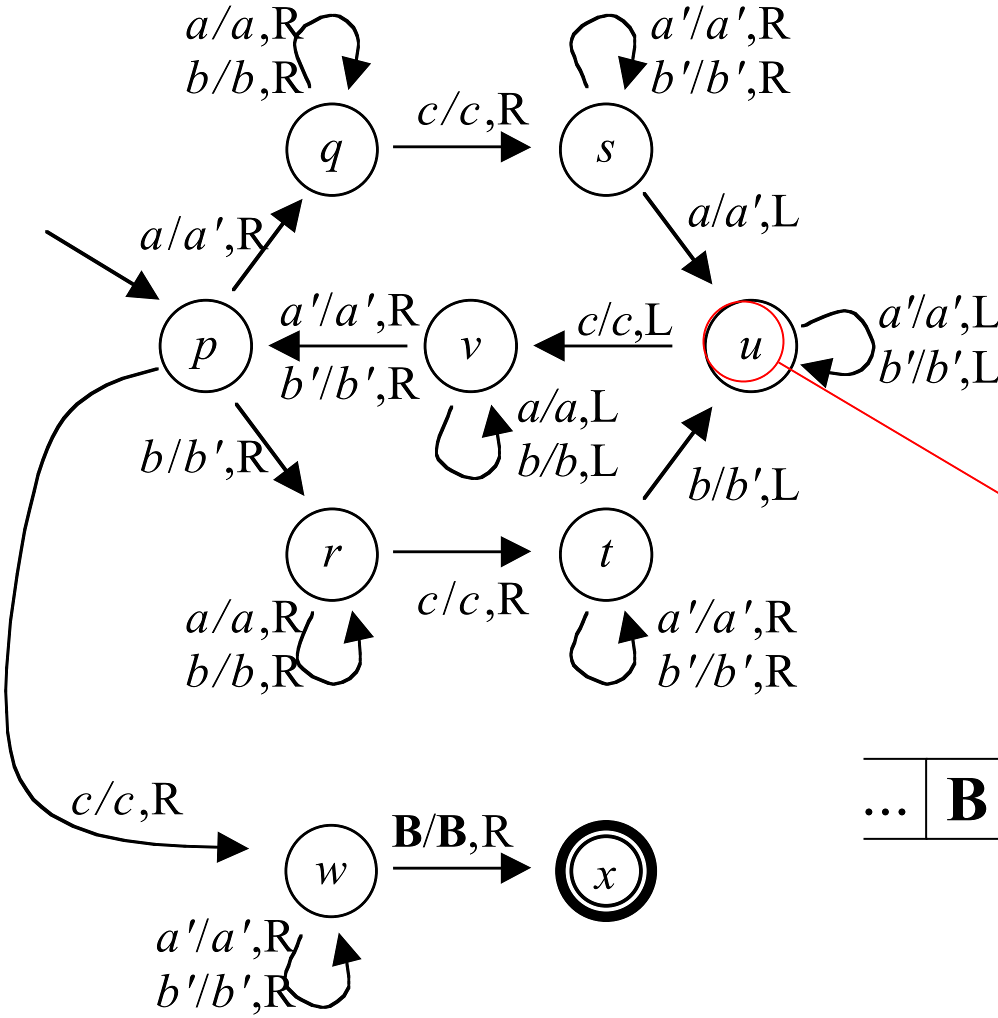


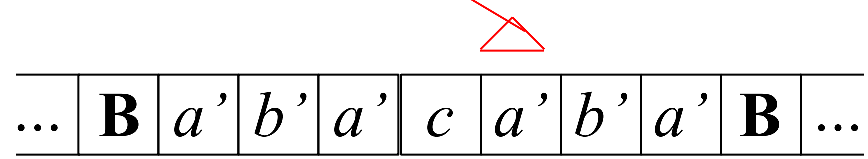
Example: *abacaba*



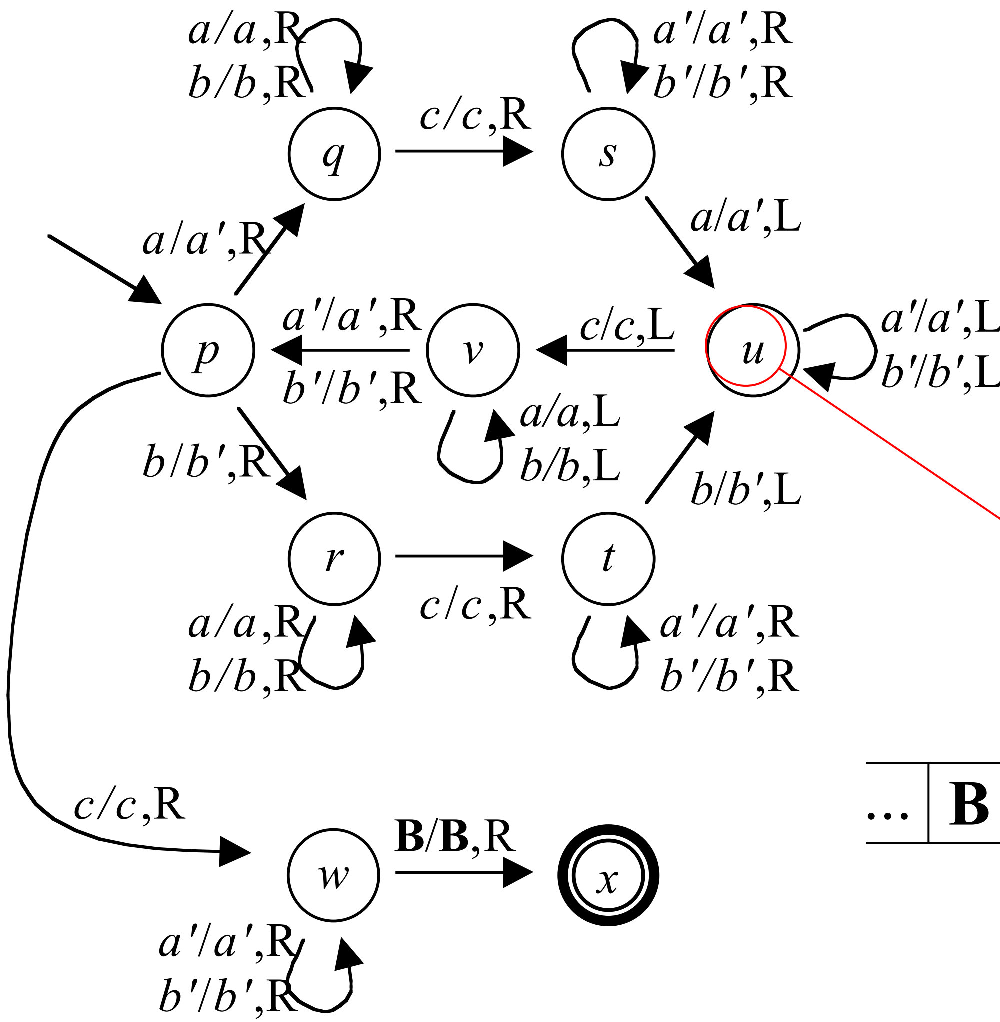


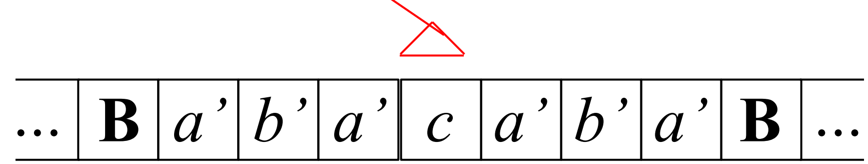
Example: *abacaba*



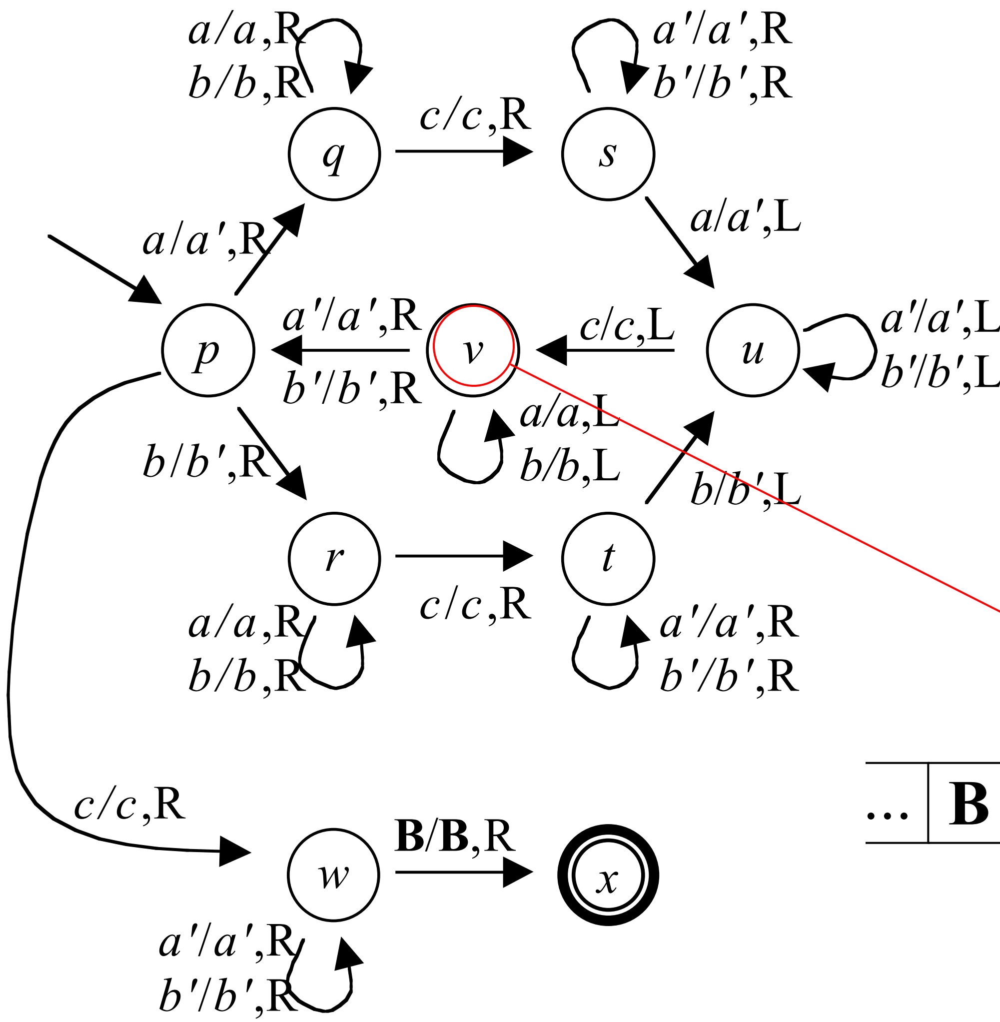


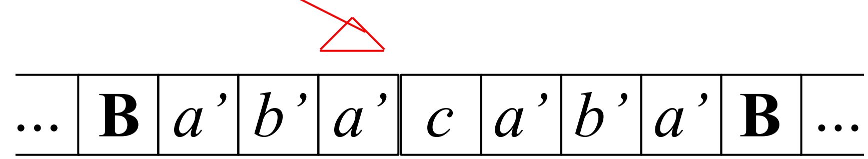
Example: *abacaba*



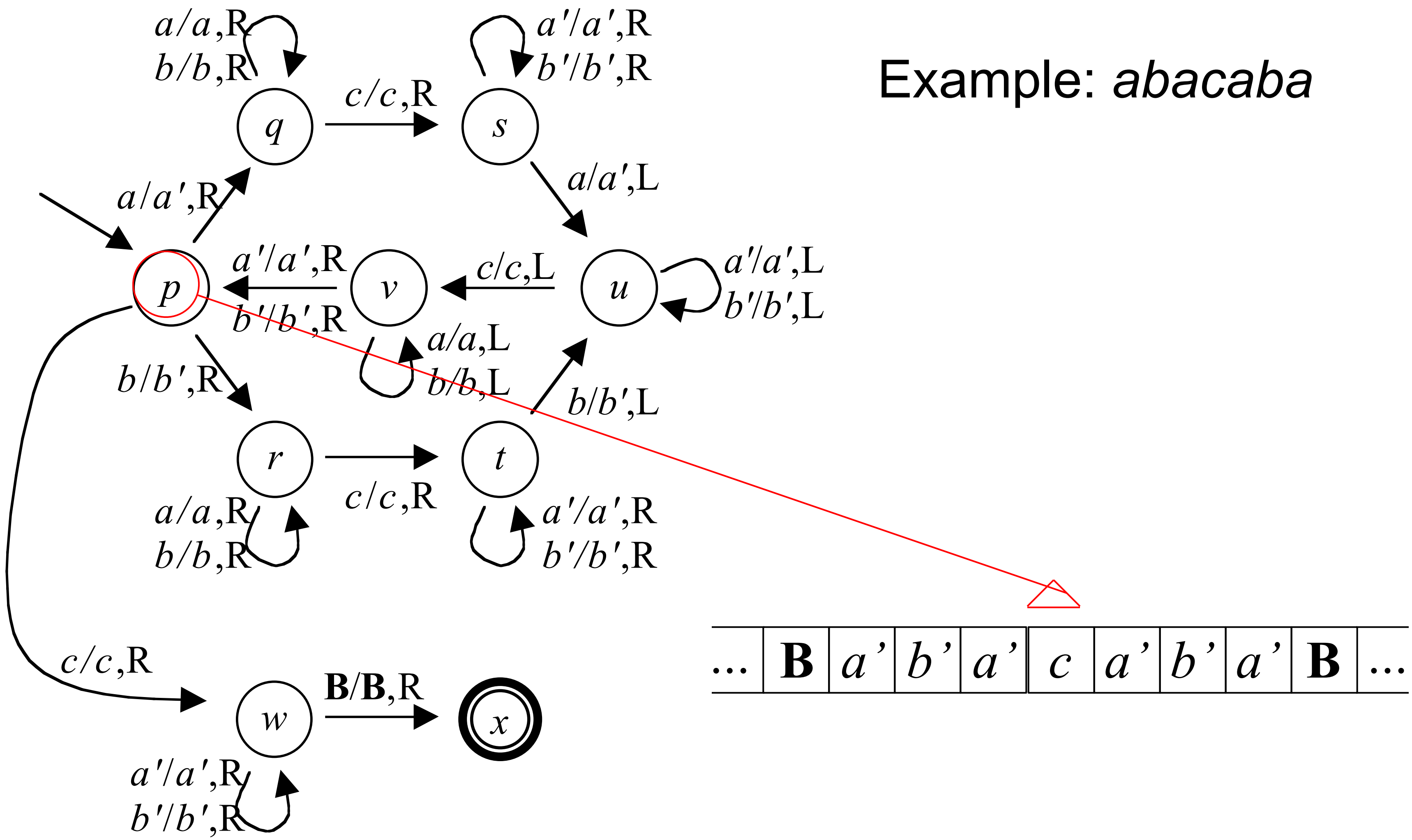


Example: *abacaba*

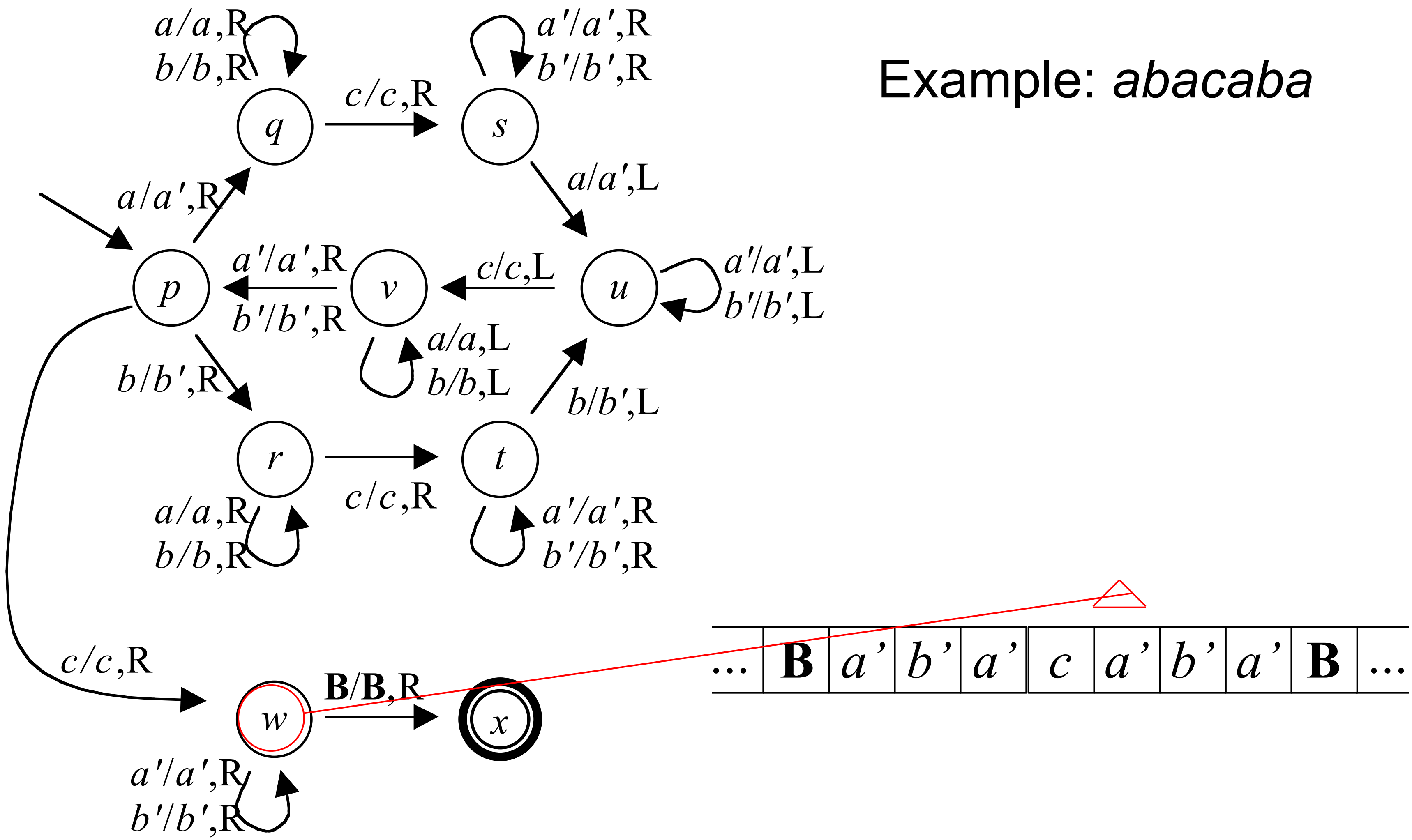




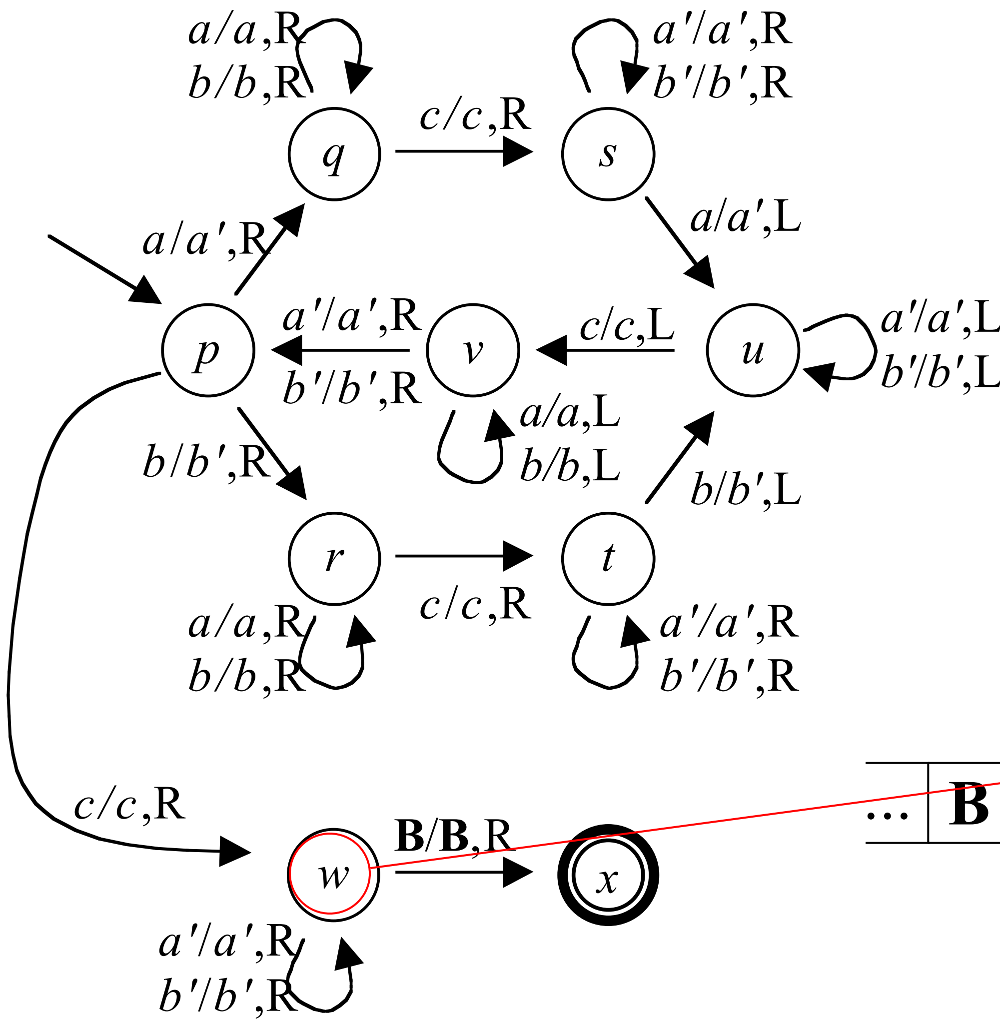
Example: *abacaba*

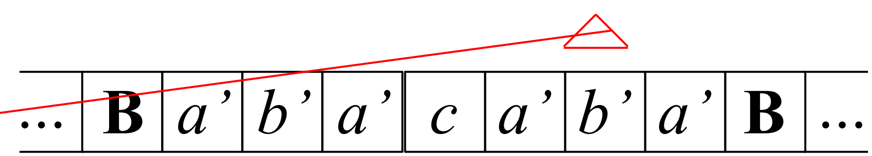


Example: *abacaba*

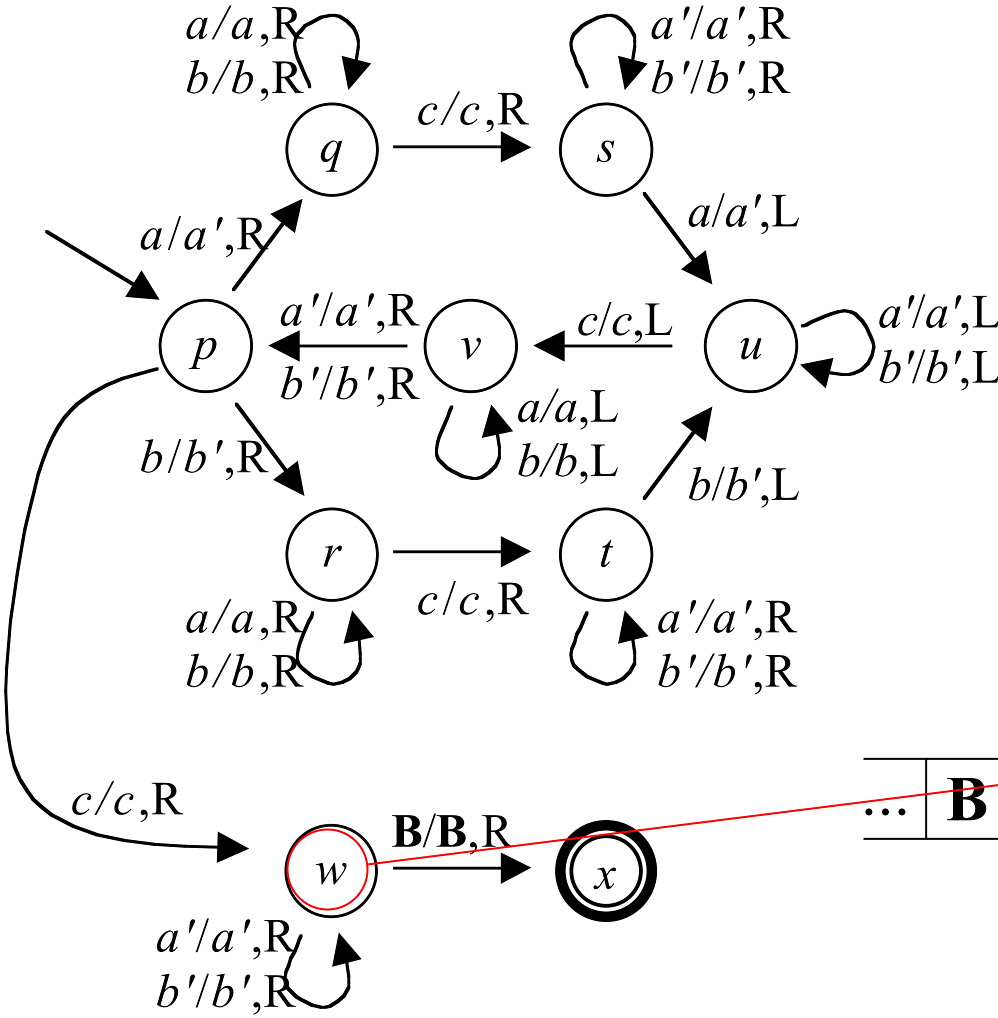


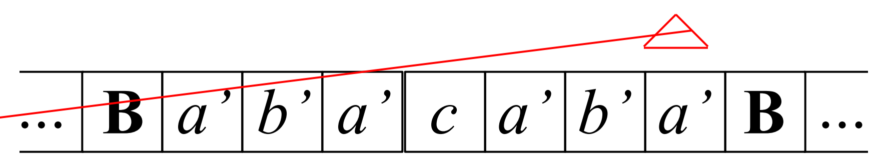
Example: *abacaba*



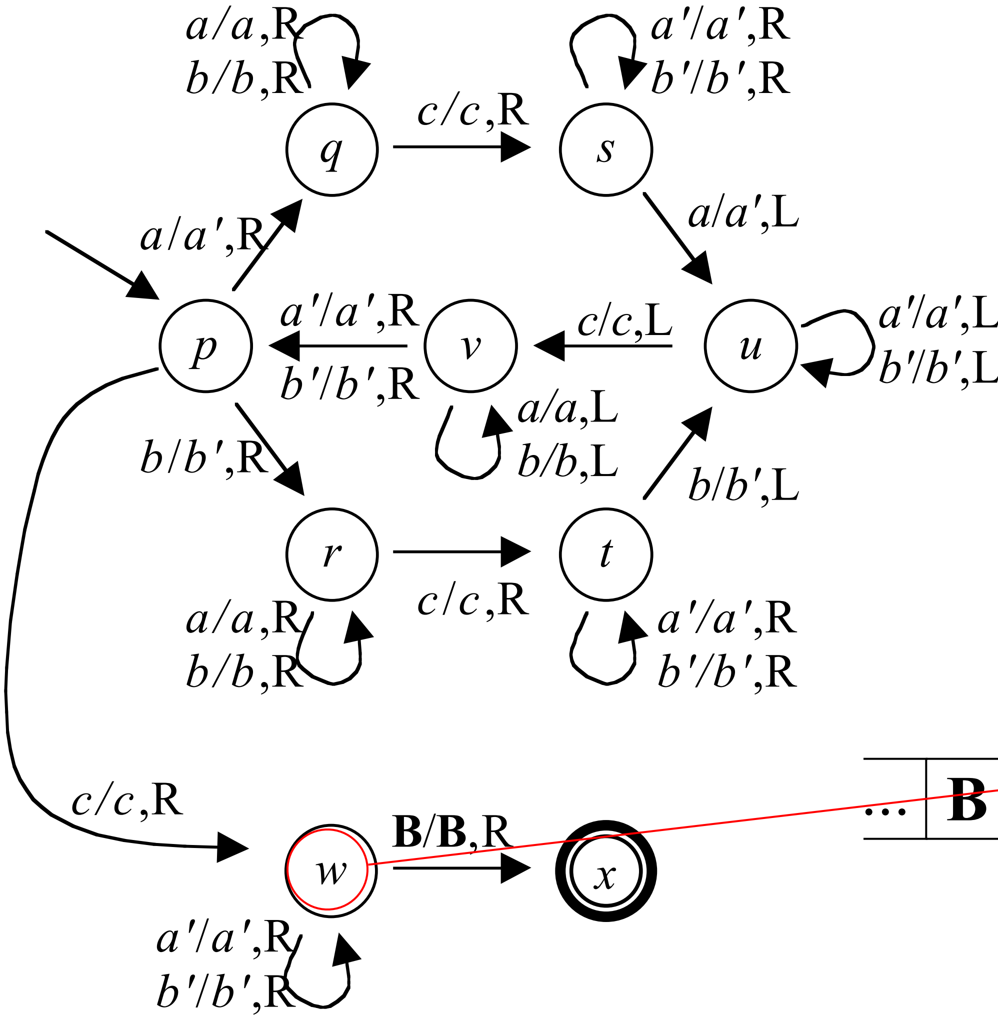


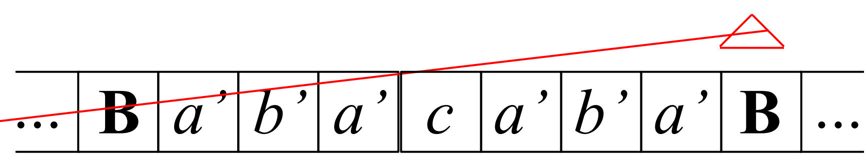
Example: *abacaba*



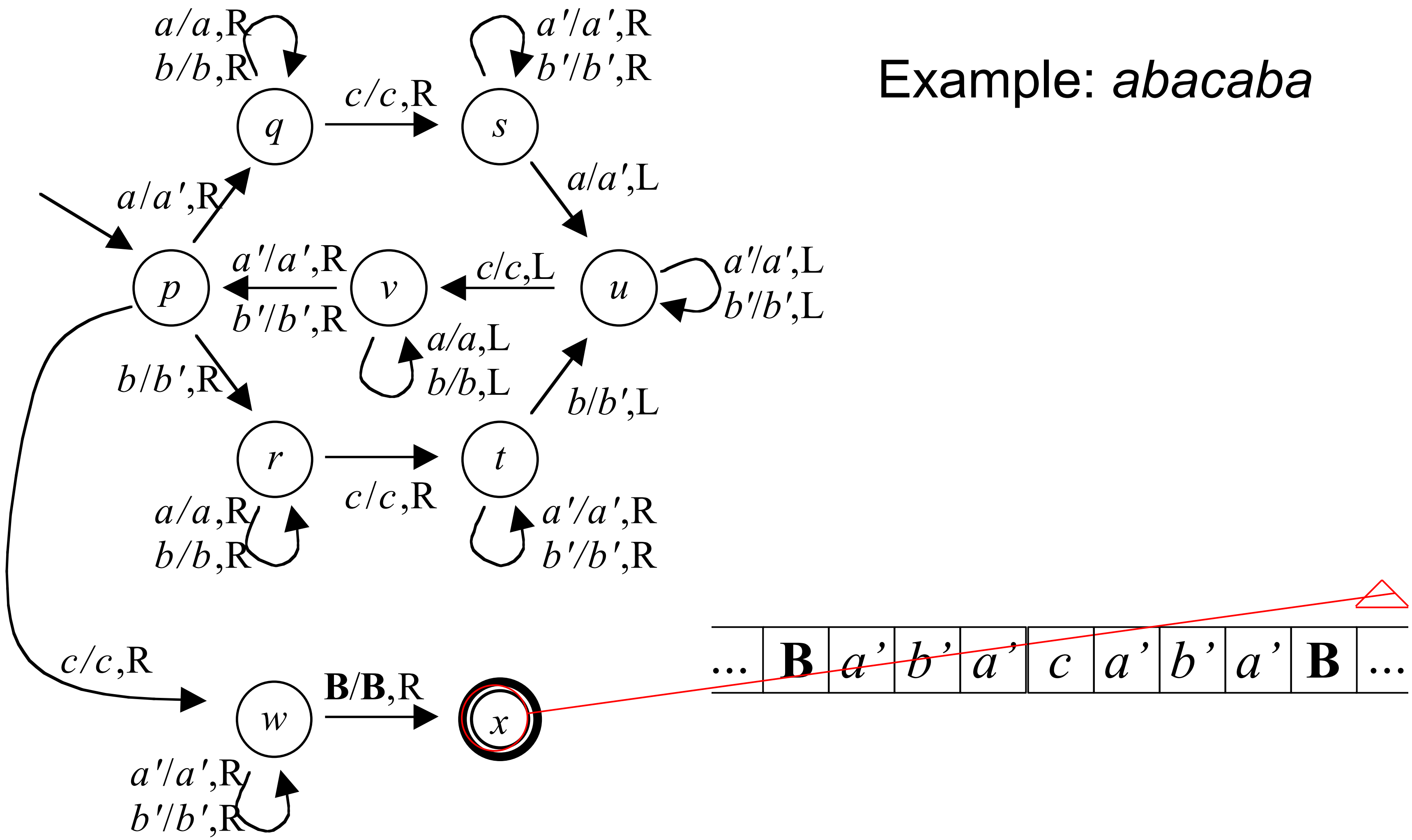


Example: *abacaba*





Example: *abacaba*



Two General Techniques

- Marking:
  - A cell can be marked by overwriting the symbol with a marked version of that symbol
  - Our input alphabet was {*a*,*b*,*c*}, but the tape alphabet was {*a*,*b*,*c*,*a'*,*b'*,**B**}
- Remembering
  - A TM can use states to record any finite information
  - Ours remembered whether *a* or *b* was seen in the first half, using two paths of states

Outline

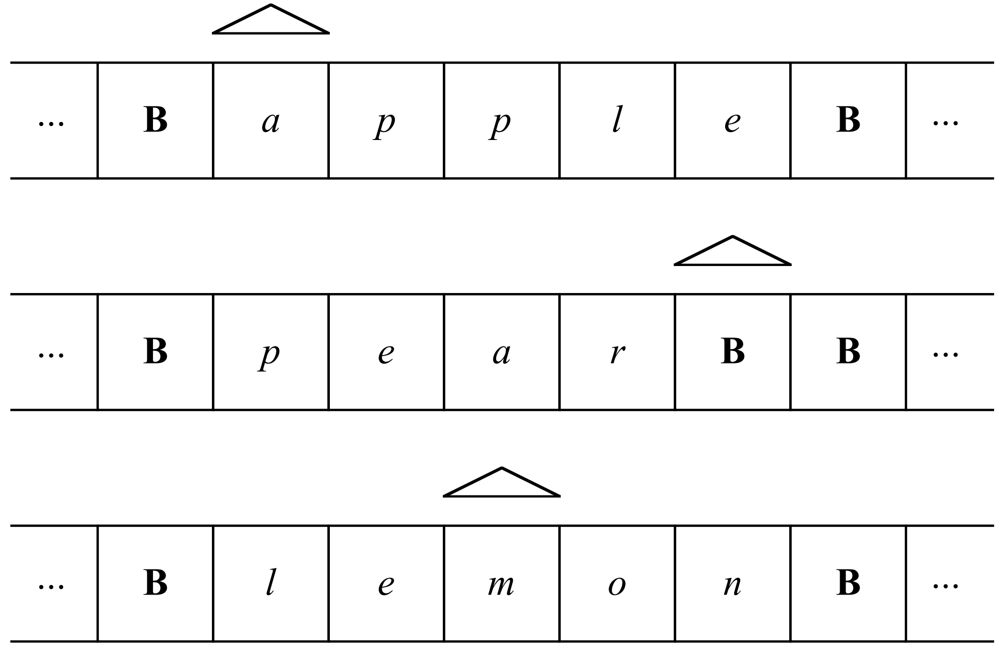
- 16.1 Turing Machine Basics
- 16.2 Simple TMs
- 16.3 A TM for $\{a^n b^n c^n\}$
- 16.4 The 7-Tuple
- 16.5 The Languages Defined By A TM
- 16.6 To Halt Or Not To Halt
- 16.7 A TM for $\{xcx \mid x \in \{a,b\}^*\}$
- 16.8 Three Tapes
- 16.9 Simulating DFAs
- 16.10 Simulating Other Automata

A 3-Tape TM

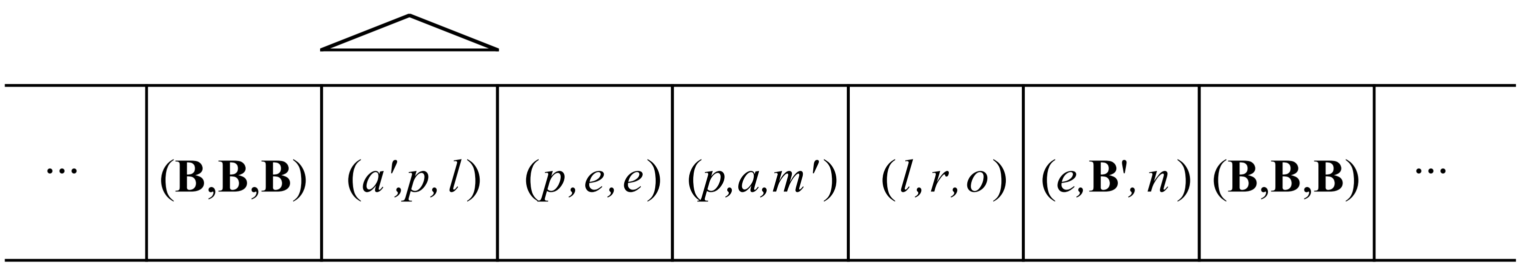
- Simulations are easier if we have a 3-tape TM
- Like the basic model, but with three tapes, each with an independent read/write head
- Basic model: $\delta(q,X) = (p,Y,D)$
  - Two inputs: current state and symbol at head
  - Three outputs: new state, symbol to write, and direction
- 3-tape model: $\delta(q,X_1,X_2,X_3) = (p,Y_1,D_1,Y_2,D_2,Y_2,D_3)$
  - Separate read, write, and direction for each of the three heads
- Otherwise, the same

Same Power

- The 3-tape model is easier to program, but no more powerful than the basic model
- For any 3-tape TM we can construct an equivalent basic TM
- We can encode all the information from the 3 tapes (with their head positions) in a single tape, using an enlarged alphabet…



- Image the three tapes side by side, each with its current head position
- Encode all this information on one tape, using triples as symbols in the enlarged tape alphabet:



New Tape Alphabet

- Each new symbol is a triple
- Marked elements in the triple indicate head positions
- For example, (a’,p,l) is a symbol of the new tape alphabet, used to show
  - a in this position on the first tape
  - p in this position on the second tape
  - l in this position on the third tape
  - First head at this position, other two elsewhere

Alphabet Construction

- To make the expanded tape alphabet:
  - First double the alphabet by the adding a marked version of each symbol
    - Γ = {*a*,**B**} would become Γ_2 = {*a*,*a*',**B**,**B**'}
  - Then cube that alphabet by forming 3-tuples
    - Γ_2 = {*a*,*a*',*b*,*b*'} would become $\Gamma_3 = (\Gamma_2 \times \Gamma_2 \times \Gamma_2)$ = {(*a*,*a*,*a*), (*a*,*a*,*a*'), (*a*,*a*,**B**), (*a*,*a*,**B**'), (*a*,*a*',*a*), (*a*,*a*',*a*'), ...}
  - The result is the tape alphabet for the 1-tape TM

1-Tape TM Construction

- Given any 3-tape M_3, construct a 1-tape M_1
  - Use the alphabet of triples, with (**B**,**B**,**B**) as blank
  - To simulate a move of M_3, M_1 makes two passes:
    - A left-to-right pass, collecting the three input symbols. Keep track (using state) of M_3's state and input symbols. Now if M_3 accepts, halt and accept; if M_3 rejects, halt and reject.
    - A right-to-left pass, carrying out M_3's actions at each of its three head positions (writing and moving marks). Leave the head at the leftmost mark and go to step 1.
- Far more states, symbols, and moves than M_3
- But for any input string, M_1's outcome matches M_3's

Theorem 16.8

For any given partition of a Σ^* into three subsets L, R, and F, there is a three-tape TM M_3 with $L(M_3) = L$, $R(M_3) = R$, and $F(M_3) = F$, if and only if there is a one-tape TM M_1 with $L(M_1) = L$, $R(M_1) = R$, and $F(M_1) = F$.

- Proof sketch:
  - Given any 3-tape TM we can construct a 1-tape TM that simulates it, as just outlined
  - Given any 1-tape TM we can construct a 3-tape TM that simulates it, simply by not using two of its tapes

Outline

- 16.1 Turing Machine Basics
- 16.2 Simple TMs
- 16.3 A TM for $\{a^n b^n c^n\}$
- 16.4 The 7-Tuple
- 16.5 The Languages Defined By A TM
- 16.6 To Halt Or Not To Halt
- 16.7 A TM for $\{xcx \mid x \in \{a,b\}^*\}$
- 16.8 Three Tapes
- 16.9 Simulating DFAs
- 16.10 Simulating Other Automata

Automata As Input

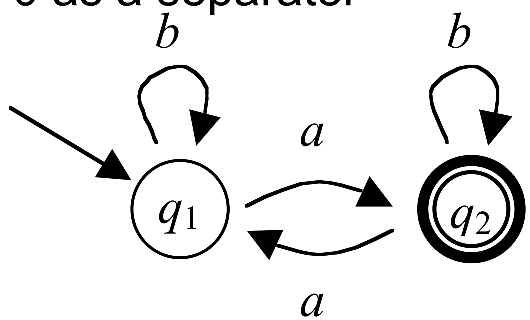
- Our goal is to make TMs that can simulate other automata, given as input
- TMs can only take strings as input, so we need a way to encode automata as strings
- We'll start with the simplest: DFAs…

DFAs Encoded Using {0,1}

- The DFA's alphabet and strings:
  - Number Σ arbitrarily as $\Sigma = \{\sigma_1, \sigma_2, ...\}$
  - Use the string 1^i to represent symbol σ_i
  - Use 0 as a separator for strings
  - For example, if $\Sigma = \{a, b\}$, let $a = \sigma_1$ and $b = \sigma_2$; then *abba* is represented by 101101101
- The DFA's states:
  - Number $Q = \{q_1, q_2, ...\}$, making q_1 the start state and numbering the others arbitrarily
  - Use the string 1^i to represent symbol q_i

DFA Encoding, Continued

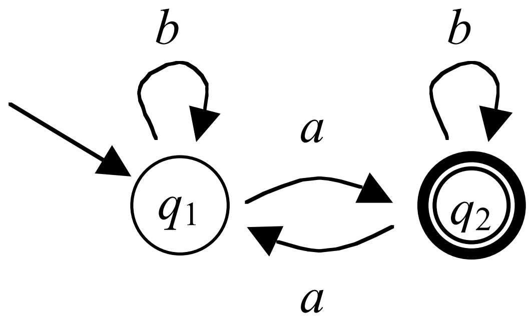
- The DFA's transition function:
  - Encode each transition $\delta(q_i,\sigma_j) = q_k$ as a string $1^i01^j01^k$
  - Encode the entire transition function as a list of such transitions, in any order, using 0 as a separator
  - For example,



  - Numbering a as σ_1 and b as σ_2, δ is

    $\delta(q_1,\sigma_1) = q_2$ $\delta(q_1,\sigma_2) = q_1$
    $\delta(q_2,\sigma_1) = q_1$ $\delta(q_2,\sigma_2) = q_2$
  - That is encoded as:
    101011 0 101101 0 110101 0 11011011

DFA Encoding, Continued



- The DFA's set of accepting states:
  - We already encode each state q_i as 1^i
  - Use a list of state codes, separated by 0s
- Finally, the complete DFA:
  - Transition-function string, 00, accepting-state string:
    101011 0 101101 0 110101 0 11011011 00 11

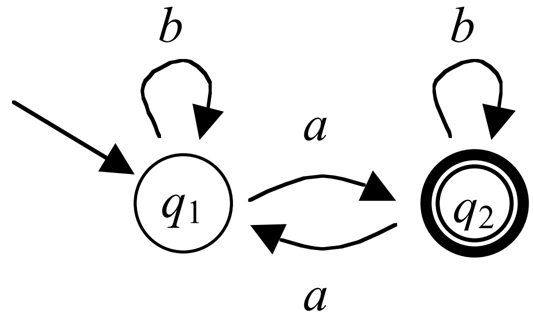
Simulating a DFA

- We have a way to represent a DFA as a string over {0,1}
- Now, we'll show how to construct a TM that simulates any given DFA
  - Given the encoded DFA as input, along with an encoded input string for it
  - Decide whether the given DFA accepts the given string
- We'll use a 3-tape TM…

3-Tape DFA Simulator

- First tape holds the DFA being simulated
- Second tape holds the DFA's input string
- Third tape hold the DFA's current state q_i, encoded as 1^i as usual

Example:



- Initial configuration, in the start state, on input *abab*:

… | **B** 101011 0 101101 0 110101 0 11011011 00 11 **B** | …

… | **B** 1 0 11 0 1 0 11 **B** | …

… | **B** 1 **B** | …

- Each simulated move performs one state transition and erases one encoded input symbol…

… **B** 101011 0 101101 0 110101 0 11011011 00 11 **B** …

… **B** 1 0 11 0 1 0 11 **B** …

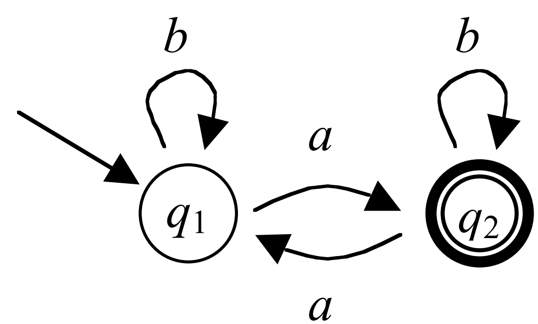
… **B** 1 **B** …

First move on *abab*: read *a*, go to state q_2

… **B** 101011 0 101101 0 110101 0 11011011 00 11 **B** …

… **B B B** 11 0 1 0 11 **B** …

… **B** 11 **B** …



Strategy

- Step 1: handle termination:
  - If the second tape is not empty, go to step 2
  - If it is empty, the DFA is done; search the list of accepting states (tape 1) for a match with the final state (tape 3)
  - If found, halt and accept; if not, halt and reject
- Step 2: look up move:
  - Search tape 1 for the move $1^i01^j01^k$ that applies now, where 1^i matches the current state (tape 3) and 1^j matches the current input symbol (tape 2)
- Step 3: execute move:
  - Replace the 1^i on the tape 3 with 1^k
  - Write **B** over the 1^j (and any subsequent 0) on tape 2
  - Go to step 1

An Easy Simulation

- That was no challenge for our 3-tape TM
- Used only a fixed, finite portion of each tape
- There is (by Theorem 16.8) a 1-tape TM with the same behavior
- One detail we're skipping: what should the TM do with ill-formed inputs?
  - If we specified behavior for ill-formed inputs, there would have to be an extra initial pass to verify the proper encoding of a DFA and its input

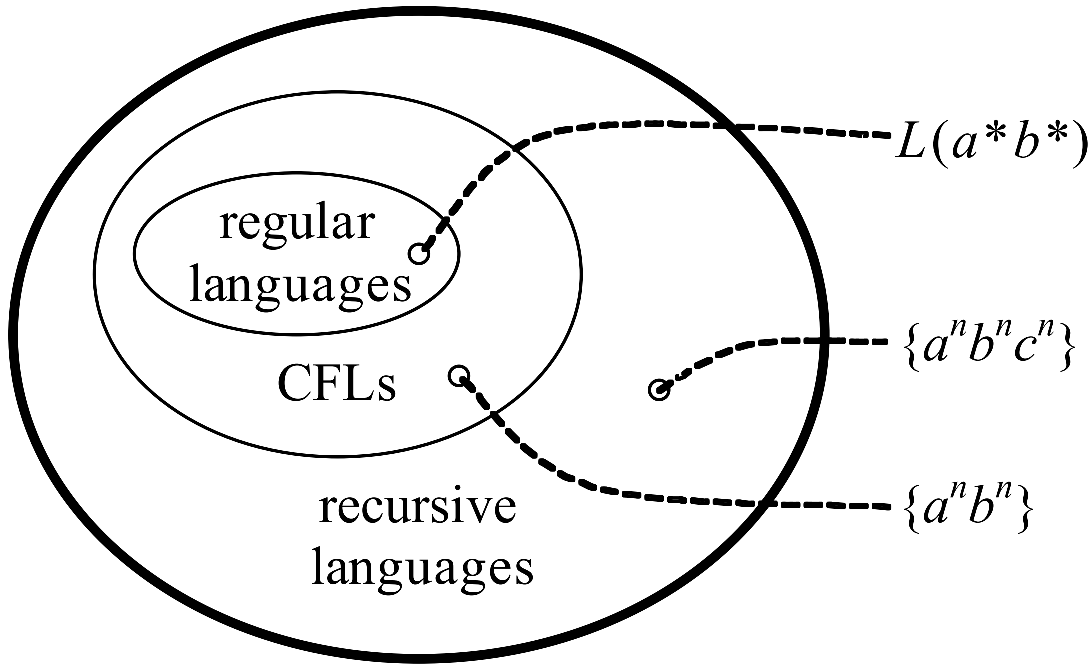
Outline

- 16.1 Turing Machine Basics
- 16.2 Simple TMs
- 16.3 A TM for $\{a^n b^n c^n\}$
- 16.4 The 7-Tuple
- 16.5 The Languages Defined By A TM
- 16.6 To Halt Or Not To Halt
- 16.7 A TM for $\{xcx \mid x \in \{a,b\}^*\}$
- 16.8 Three Tapes
- 16.9 Simulating DFAs
- 16.10 Simulating Other Automata

Simulating Other Automata

- We can use the same 3-tape technique to simulate all our other automata
- Trickier for nondeterministic models (NFAs and stack machines): our deterministic TM must search all sequences of moves
- Relatively straightforward for deterministic automata

Language Categories



- Proofs of these inclusions can be constructed by having total TMs simulate other automata

Universal Turing Machines

- A *universal Turing machine* is any Turing machine that takes an encoded Turing machine and an encoded input string, and decides whether the given Turing machine accepts the given string
- It's like an interpreter: a program that takes another program as input and carries out the instructions of that input program
- A universal TM is, in effect, a TM interpreter

A Universal TM Outline

- Design a TM encoding using {0,1}:
  - Most of it is the transition function, like our DFA encoding
- Familiar 3-tape layout:
  - Tape 1: the encoded TM as input
  - Tape 2: that TM's tape (input and working space)
  - Tape 3: that TM's current state
- Simulation:
  - Look up the appropriate transition (on tape 1)
  - Do the necessary write and move (on tape 2)
  - Do the state change (on tape 3)
  - Repeat until accepting state or no next move

Constructions?

- All the “constructions” since 16.7 have been high-level outlines, not detailed constructions:
  - 3-tape to 1-tape conversion
  - DFA simulator
  - Universal TM
- In effect, we sketched proofs that the constructions were possible, without actually doing them
- Why not give more detail?

The Problem Of Detail

- It is hard to figure out what a TM does by inspection
  - Very hard for small TMs
  - Inhumanly difficult for large TMs
- To convince someone that a language can be recognized by a TM, it is not often useful to just show the TM that does it
- It is more convincing to give less detail -- to describe in outline how a TM might work
- Once you’re convinced a TM can be constructed for a given problem, there is no point in actually constructing it…

TM Puzzles

- …except for fun!
- People have actually constructed many universal TMs in full detail
- It is an interesting puzzle
- The challenge is to do it using the smallest number of states and the smallest alphabet
- An informal global competition for many years
- See Minsky, *Computation: Finite and Infinite Machines*
- He derives a small (but not record-holding) universal TM with $|Q| = 7$ and $|\Gamma| = 4$