Chapter Seventeen: Computability

Computability describe problems that range from the trivially simple to the hideously complicated, but all are algorithmically solvable given unbounded time and resources.

The shape of this border was first clearly perceived in the 1930s.

Recursive Languages - also known as *Turing Decidable* languages are considered languages that can be solved algorithmically.

Recursively Enumerable Languages - also known as *Turing Recognizable* languages include languages that cannot be solved algorithmically.

Outline

- 17.1 Turing-Computable Functions
- 17.2 TM Composition
- 17.3 TM Arithmetic
- 17.4 TM Random Access
- 17.5 Functions And Languages
- 17.6 The Church-Turing Thesis
- 17.7 TM and Java Are Interconvertible
- 17.8 Infinite Models, Finite Machines

Turing Computability

- Consider any total TM
- Ignore final state, and consider final *tape*
- Viewed this way, a total TM computes a function with string input and string output
- A function f: $\Sigma^* \rightarrow \Gamma^*$ is *Turing-computable* if and only if there is some total TM *M* such that for all $x \in \Sigma^*$, if *M* starts with *x* on the tape, *M* halts with $f(x)$ on the tape

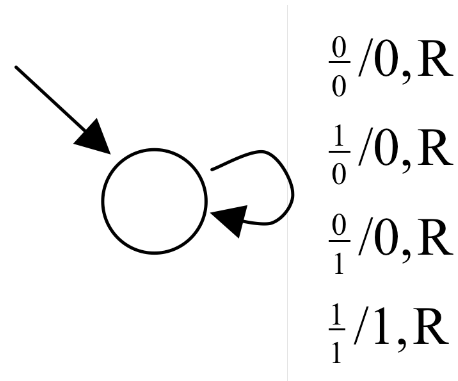
Bitwise *and*

- Like Java's `x&y`
- But allow arbitrarily long strings of bits
- First, assume "stacked" inputs:

$$\Sigma = \left\{ \frac{0}{0}, \frac{0}{1}, \frac{1}{0}, \frac{1}{1} \right\}$$

- So, for example, $and\left(\frac{0}{0}\frac{1}{1}\frac{1}{0}\frac{0}{1}\frac{1}{1}\frac{1}{0}\right) = 010010$
- The *and* function is Turing computable…

TM Computing *and*



- One left-to-right pass, overwriting each stacked pair with the bitwise and of that pair
- Halts at the end (no transition on **B**)
- *L*(*M*) = {}, but we're not interested in the language it defines
- It computes our *and* function

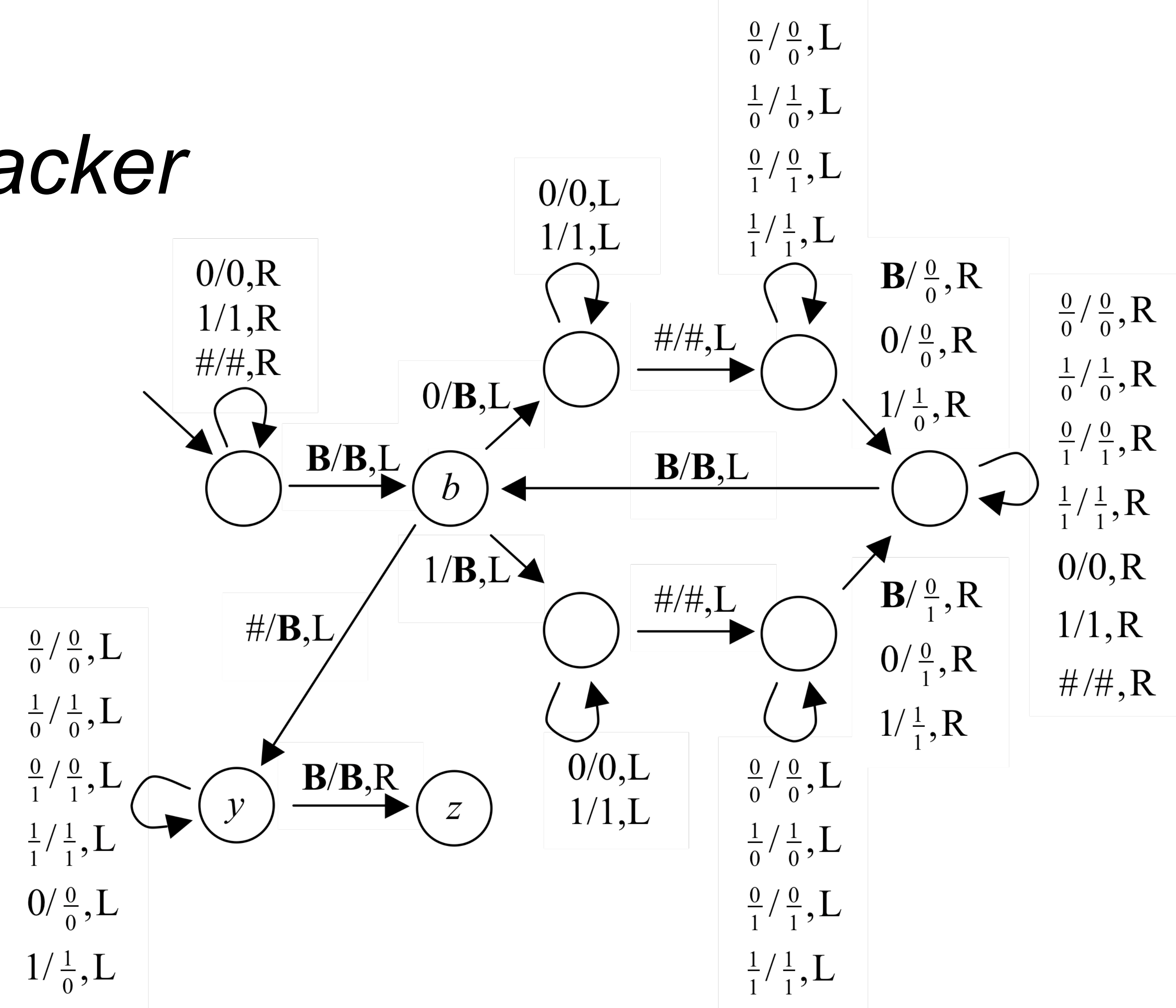
Outline

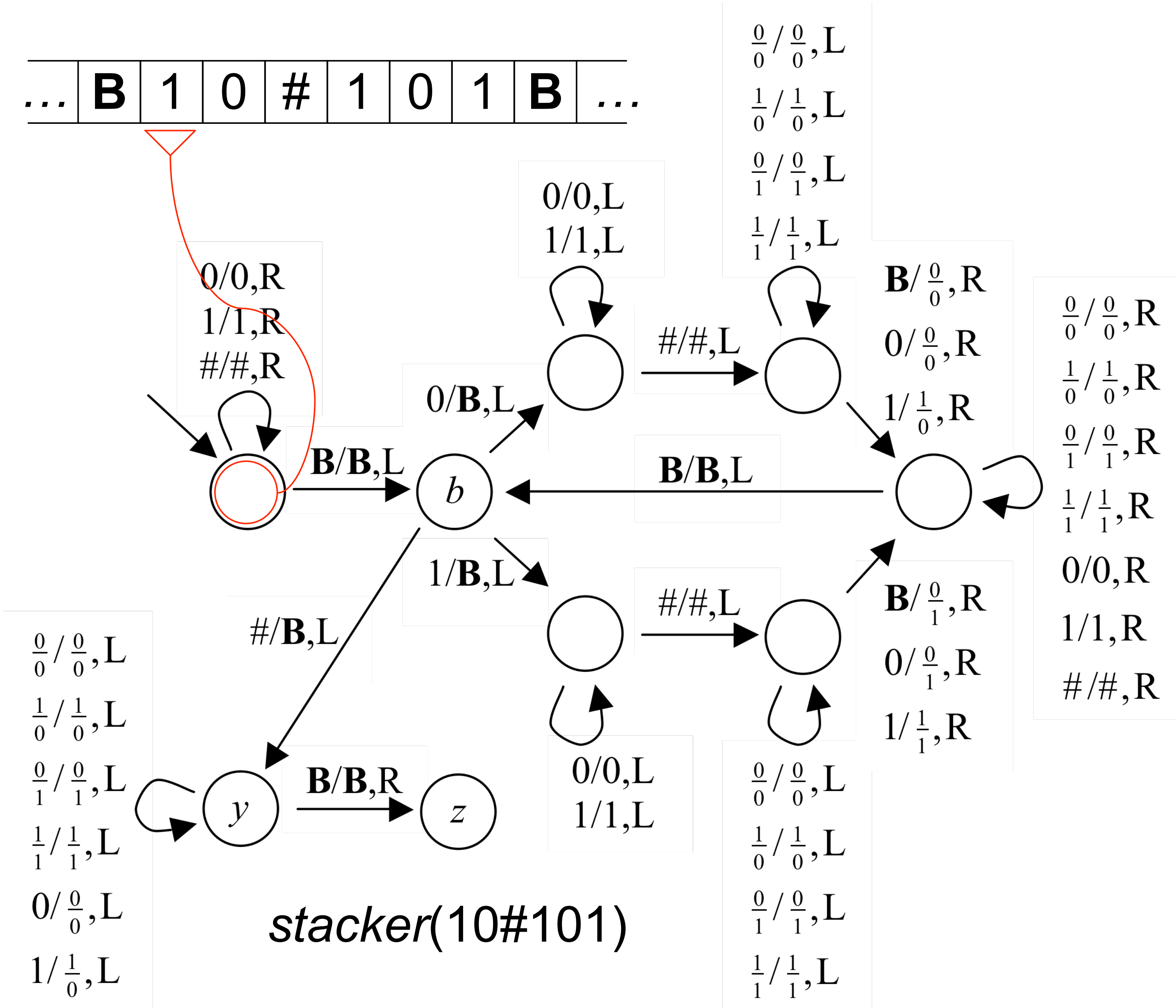
- 17.1 Turing-Computable Functions
- 17.2 TM Composition
- 17.3 TM Arithmetic
- 17.4 TM Random Access
- 17.5 Functions And Languages
- 17.6 The Church-Turing Thesis
- 17.7 TM and Java Are Interconvertible
- 17.8 Infinite Models, Finite Machines

The *stacker* function

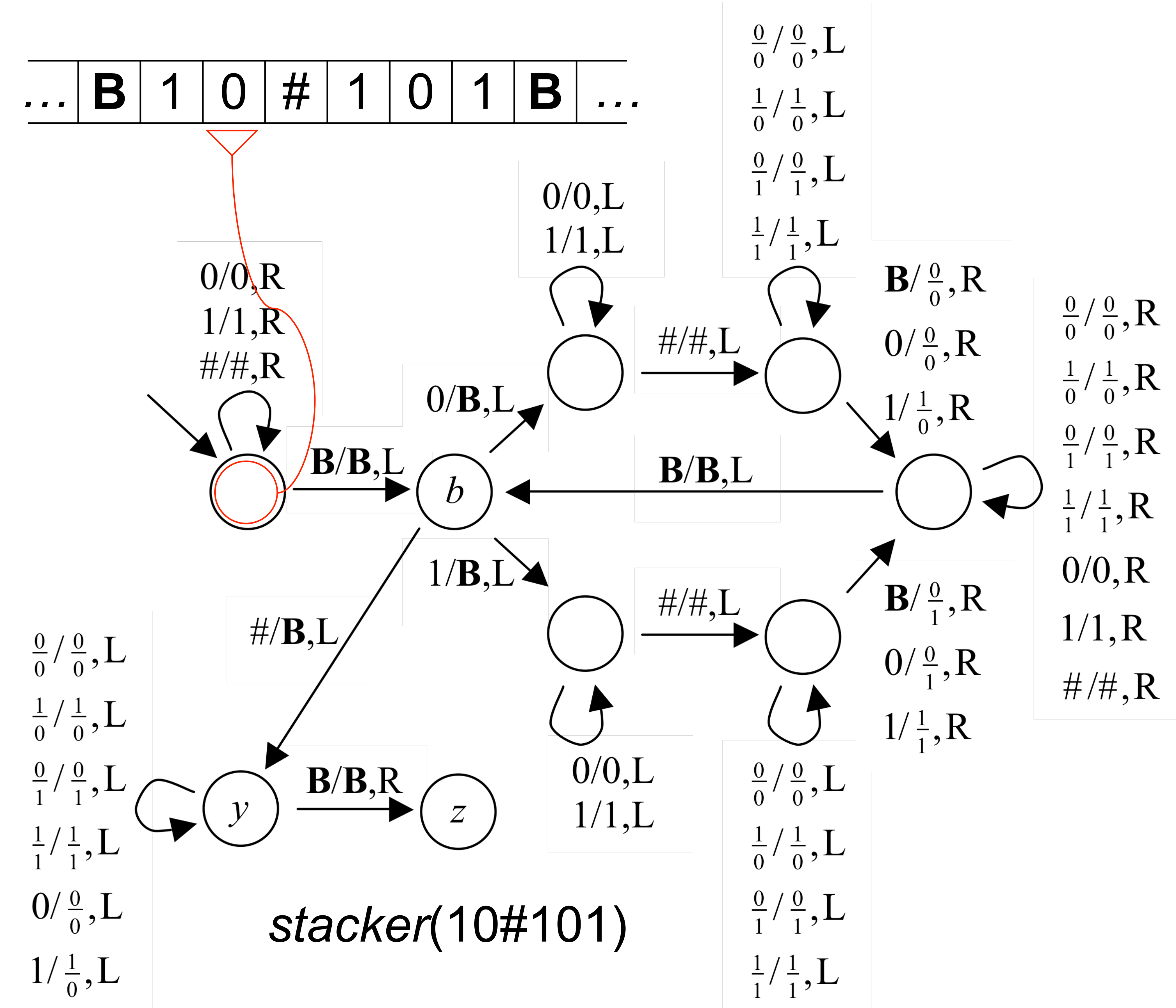
- Specifying stacked inputs made *and* very easy
- What if the two inputs are linear: two plain binary numbers with a separator between?
- Simple: we first convert to stacked form
- Define *stacker* to be the function that takes a string with two binary inputs separated by #, and returns the equivalent stacked form
- For example, $stacker(011011\#010110) = \frac{0}{0}\frac{1}{1}\frac{1}{0}\frac{0}{1}\frac{1}{1}\frac{1}{0}$
- (If one is shorter than the other, 0-extend it on the left)
- This *stacker* function is Turing computable…

stacker

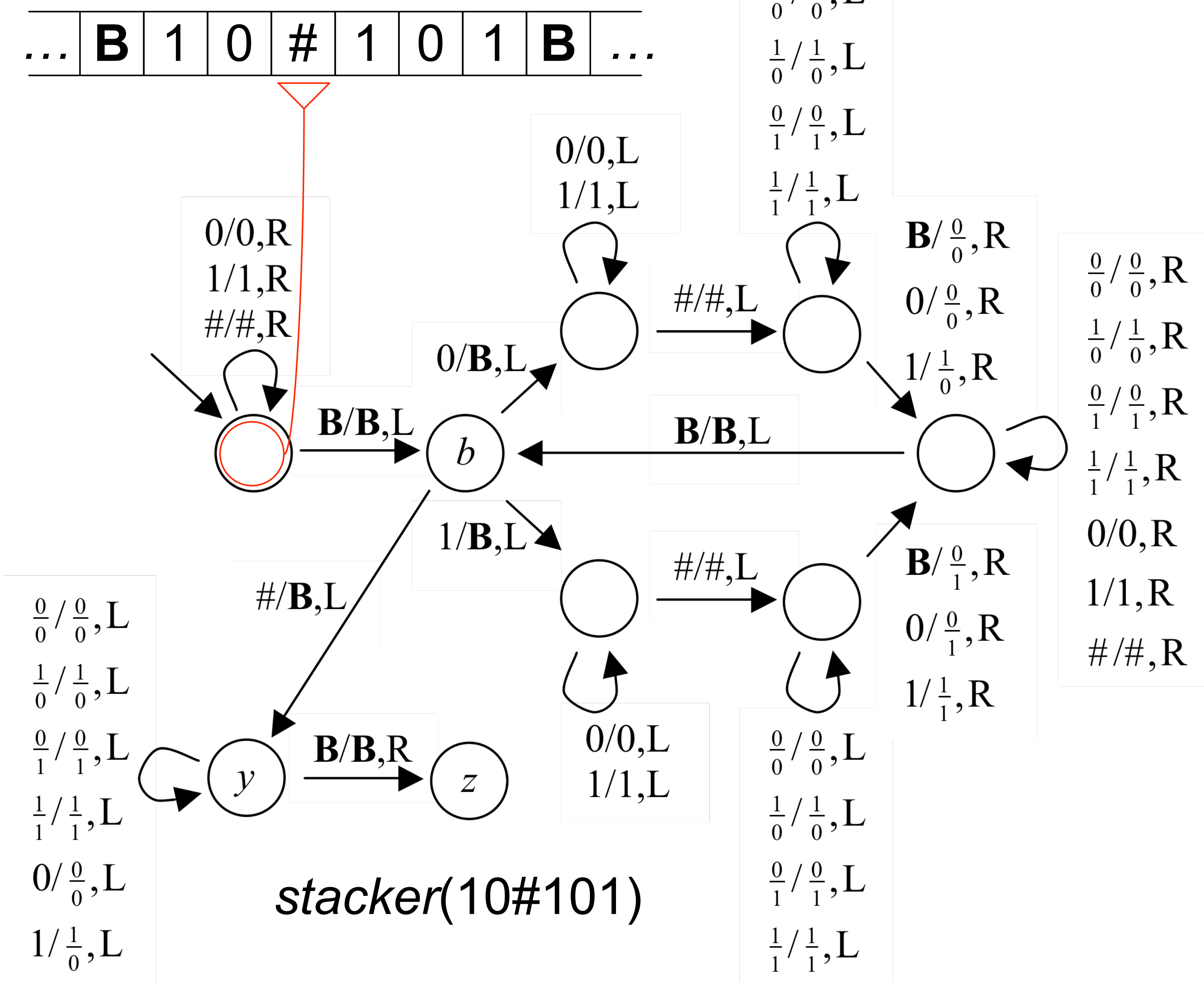




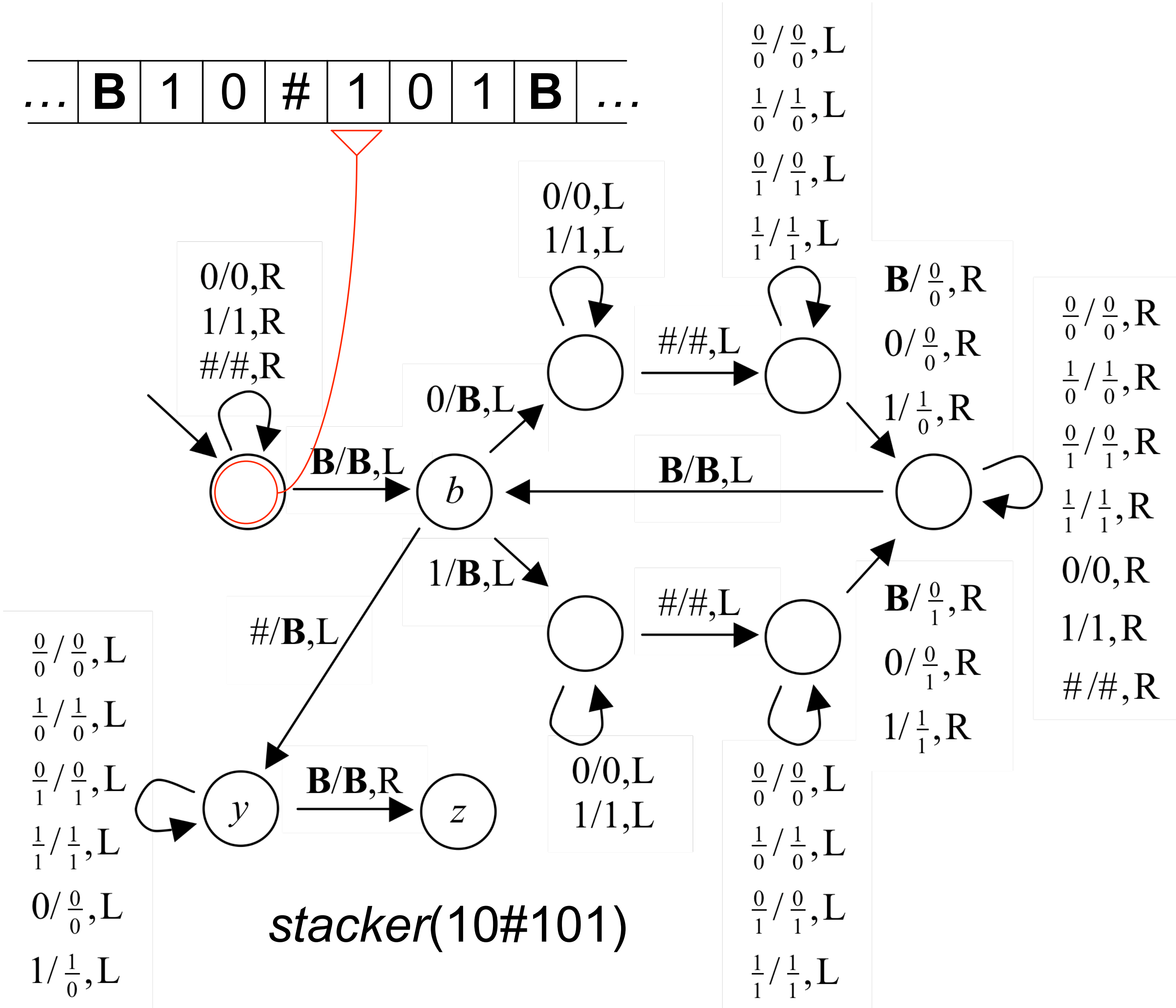
stacker(10#101)



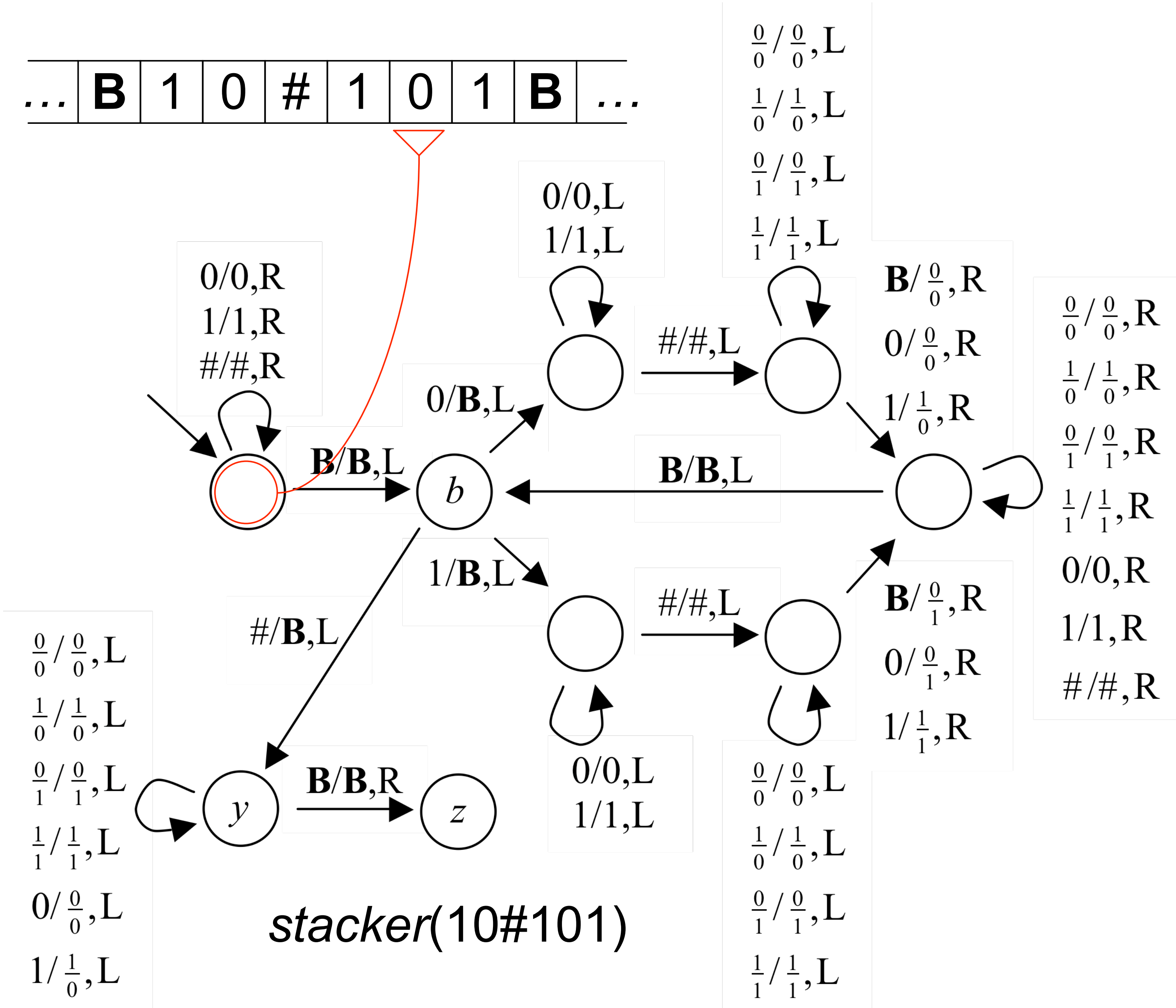
stacker(10#101)



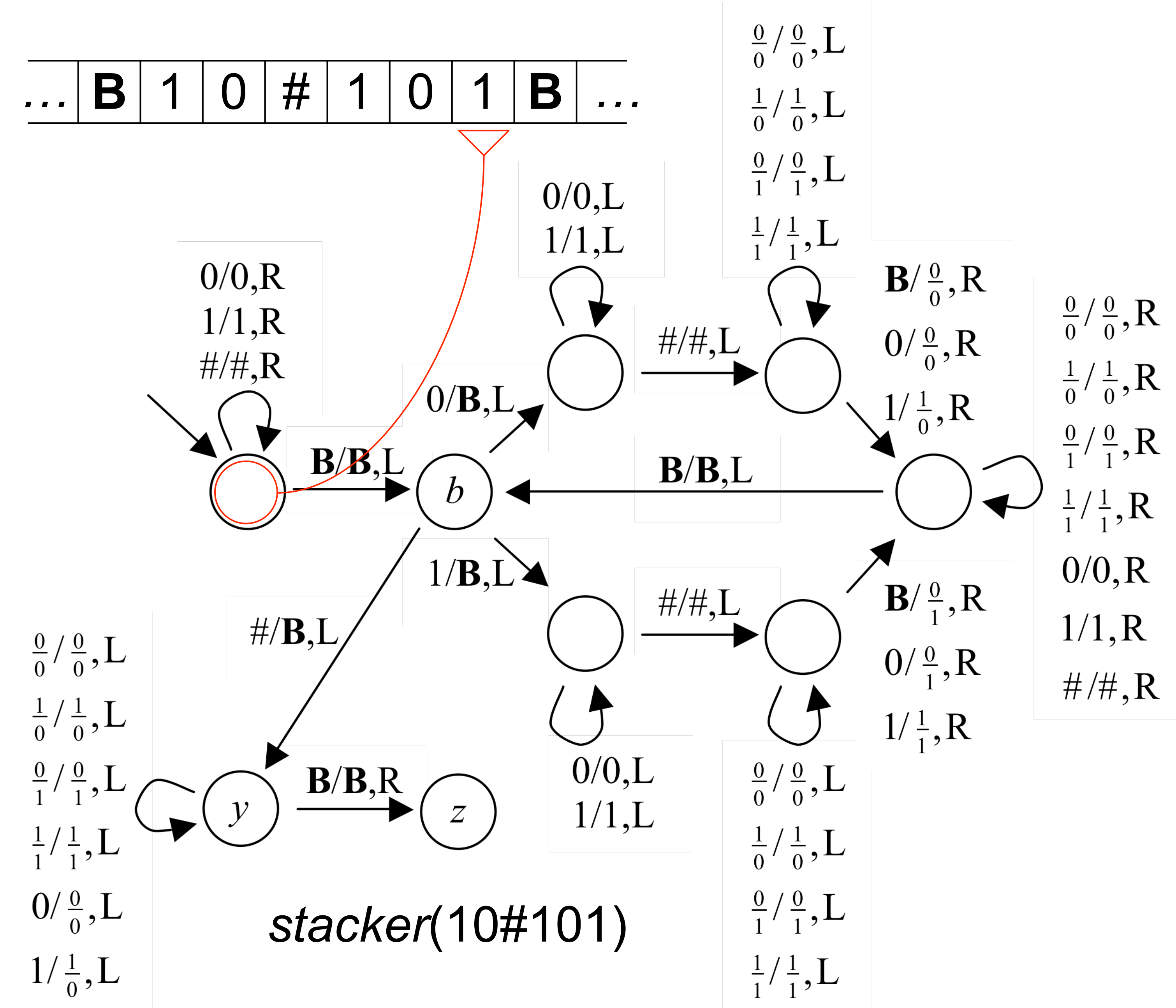
stacker(10#101)



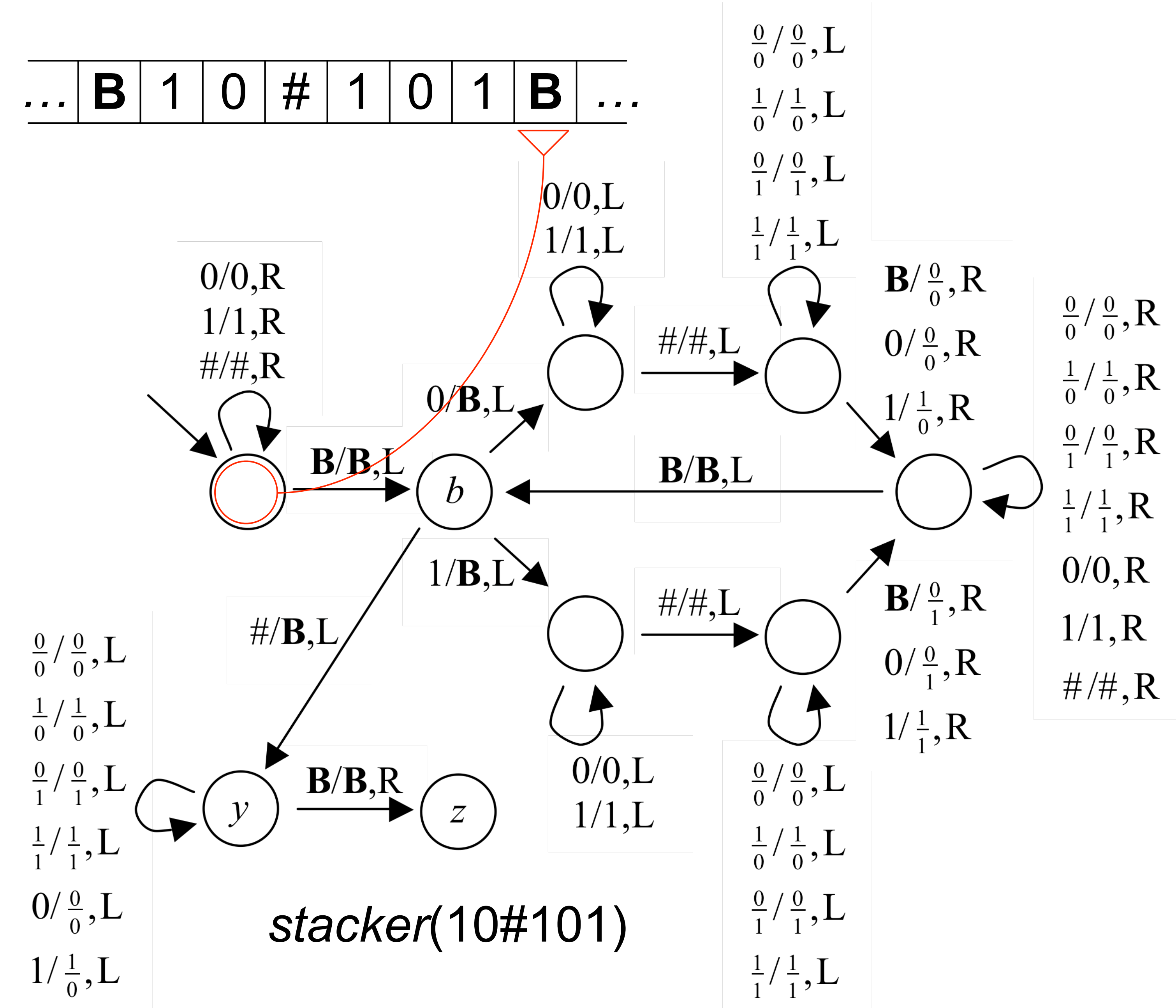
stacker(10#101)



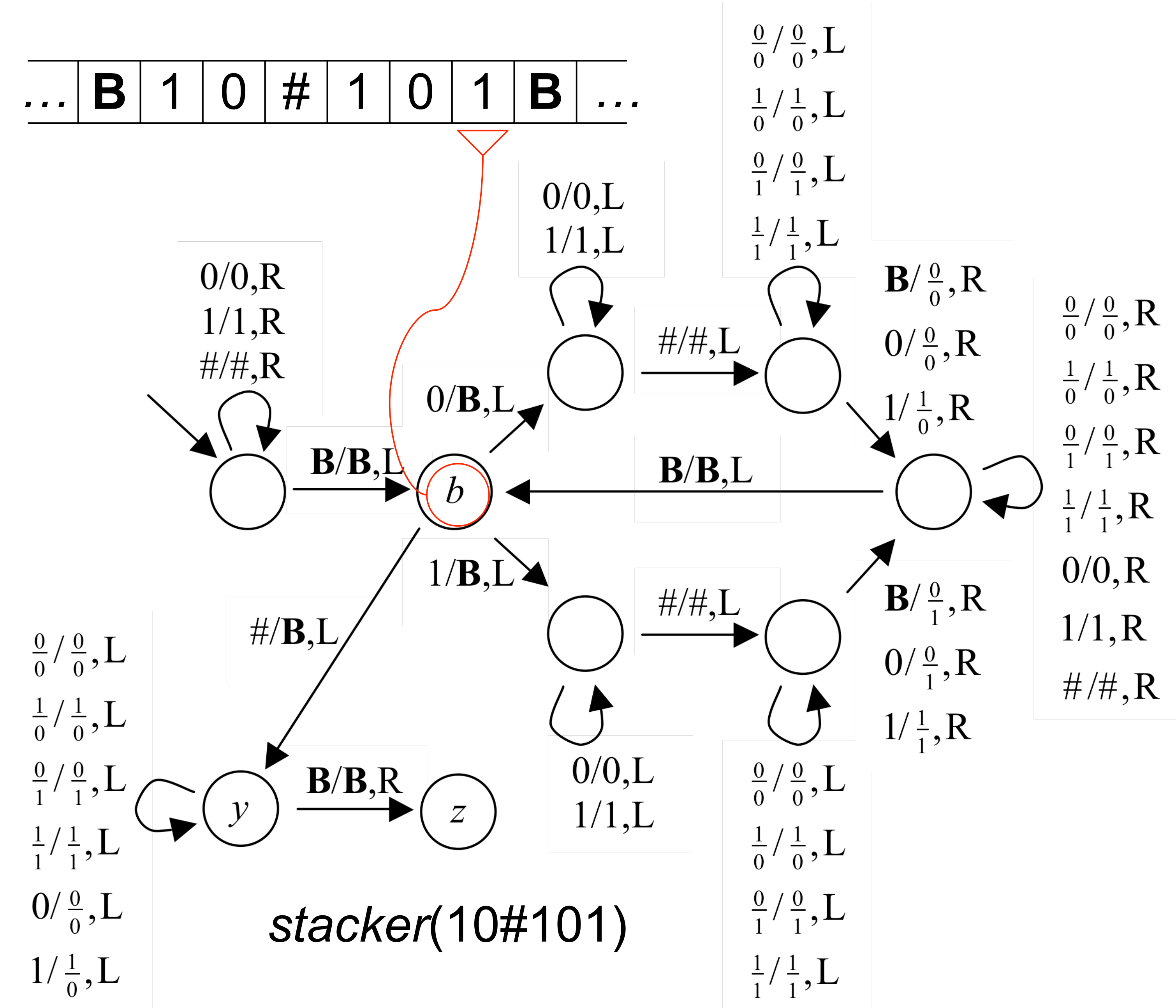
stacker(10#101)



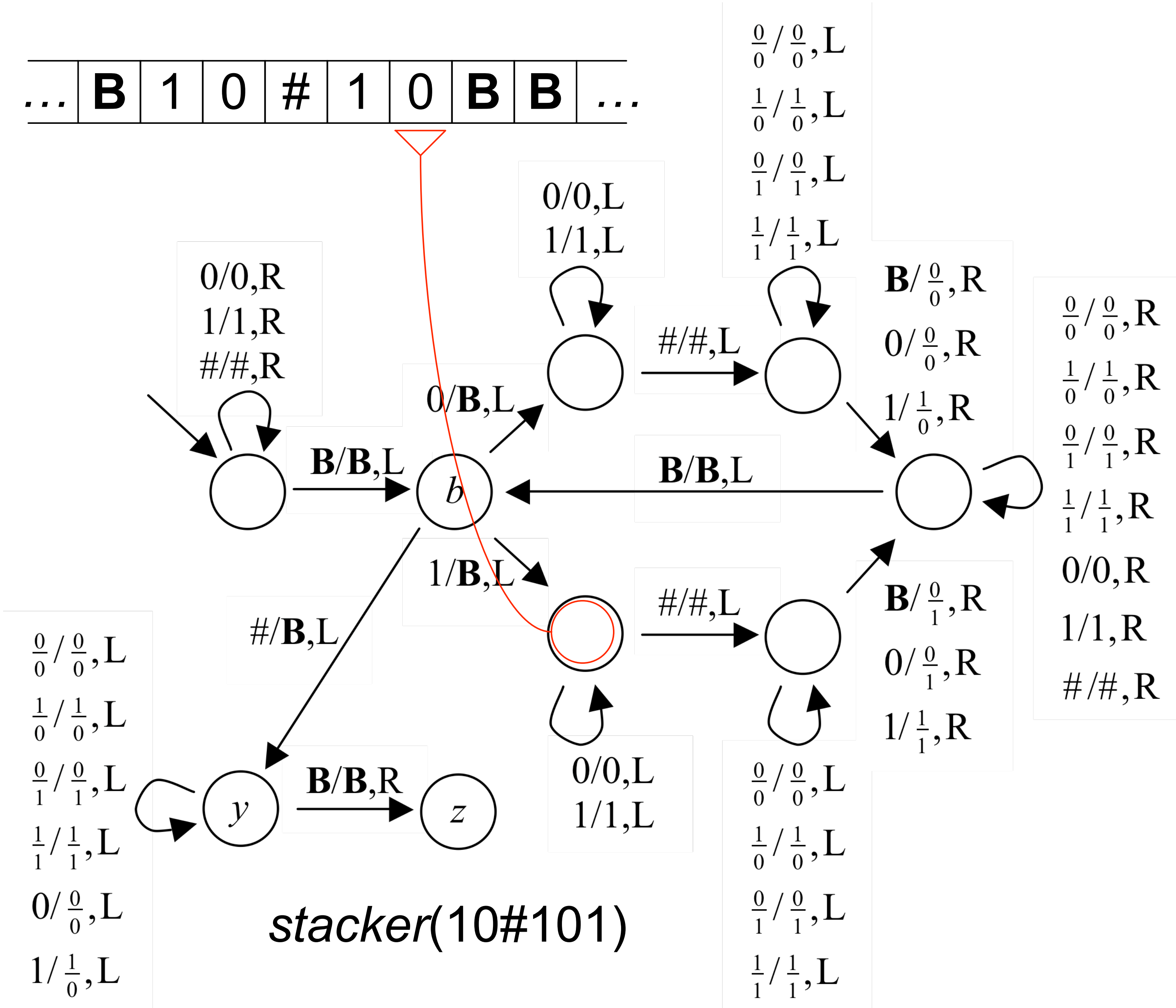
stacker(10#101)



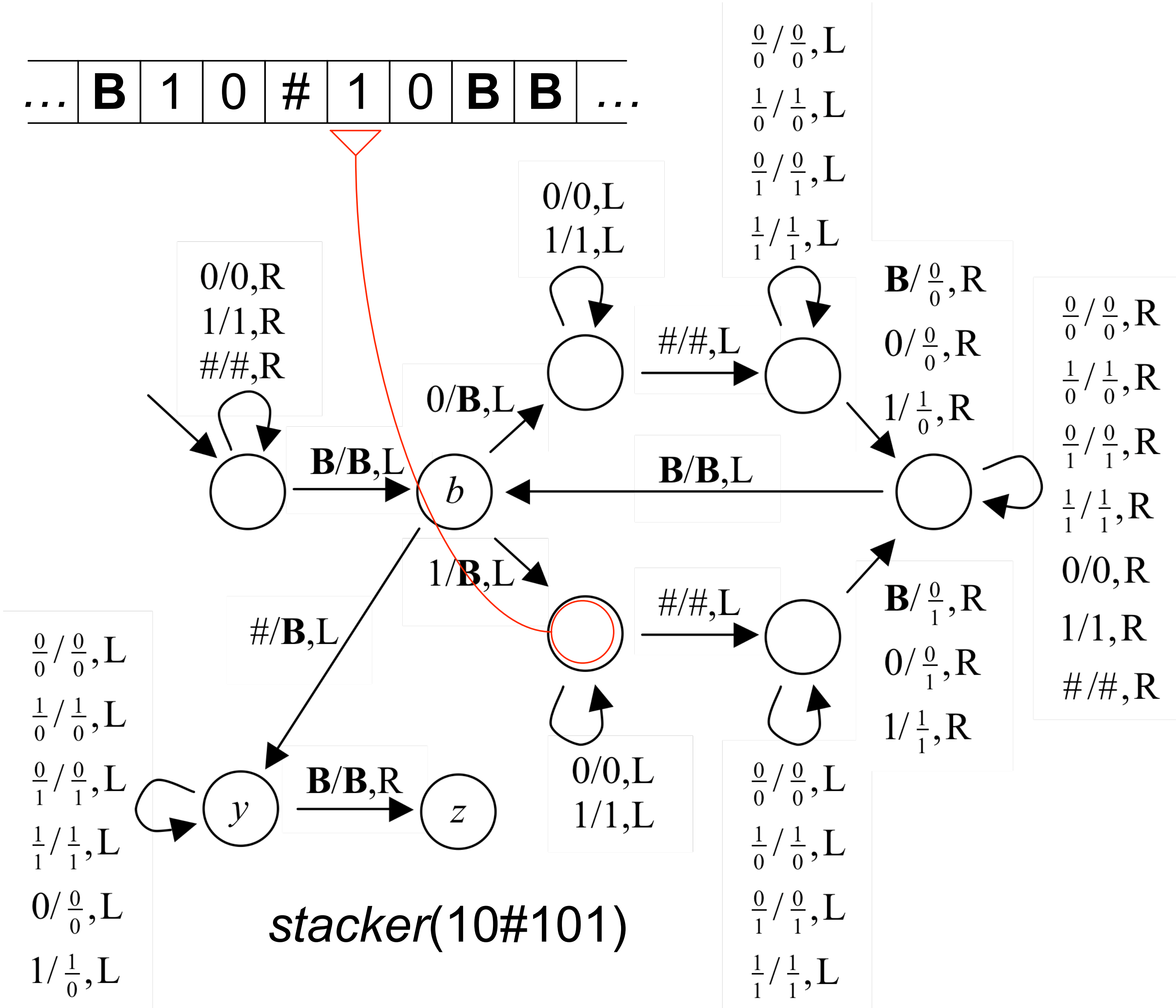
stacker(10#101)



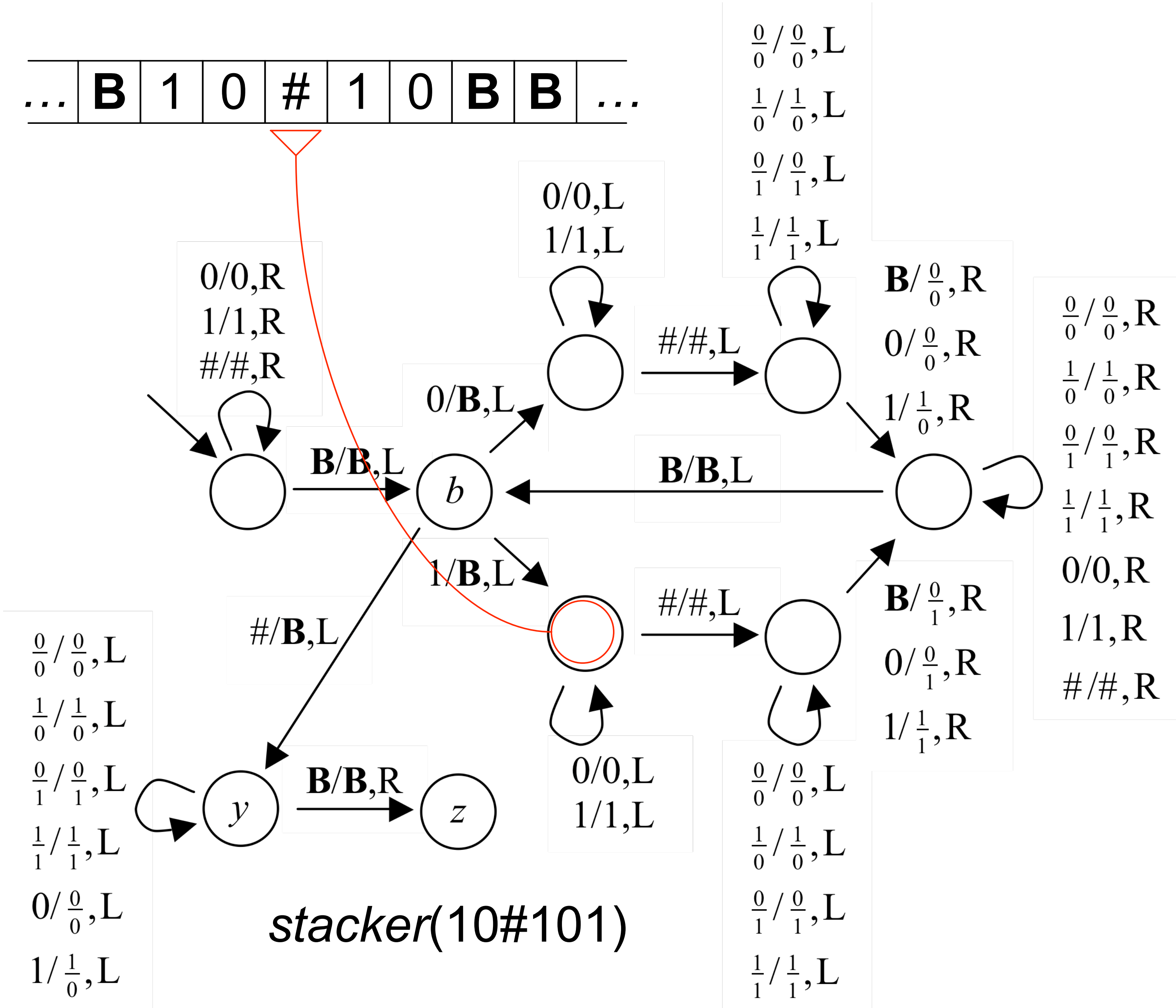
stacker(10#101)



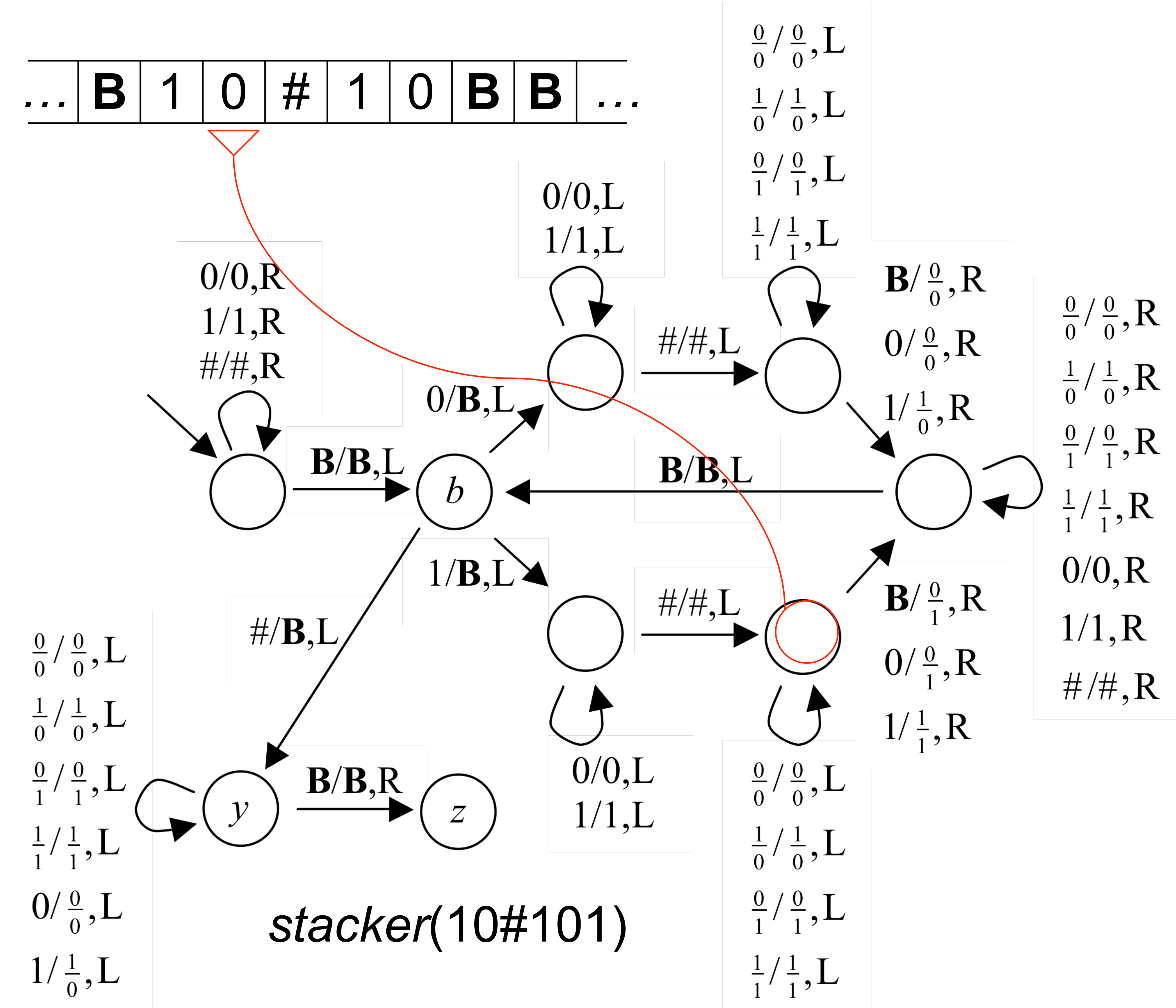
stacker(10#101)



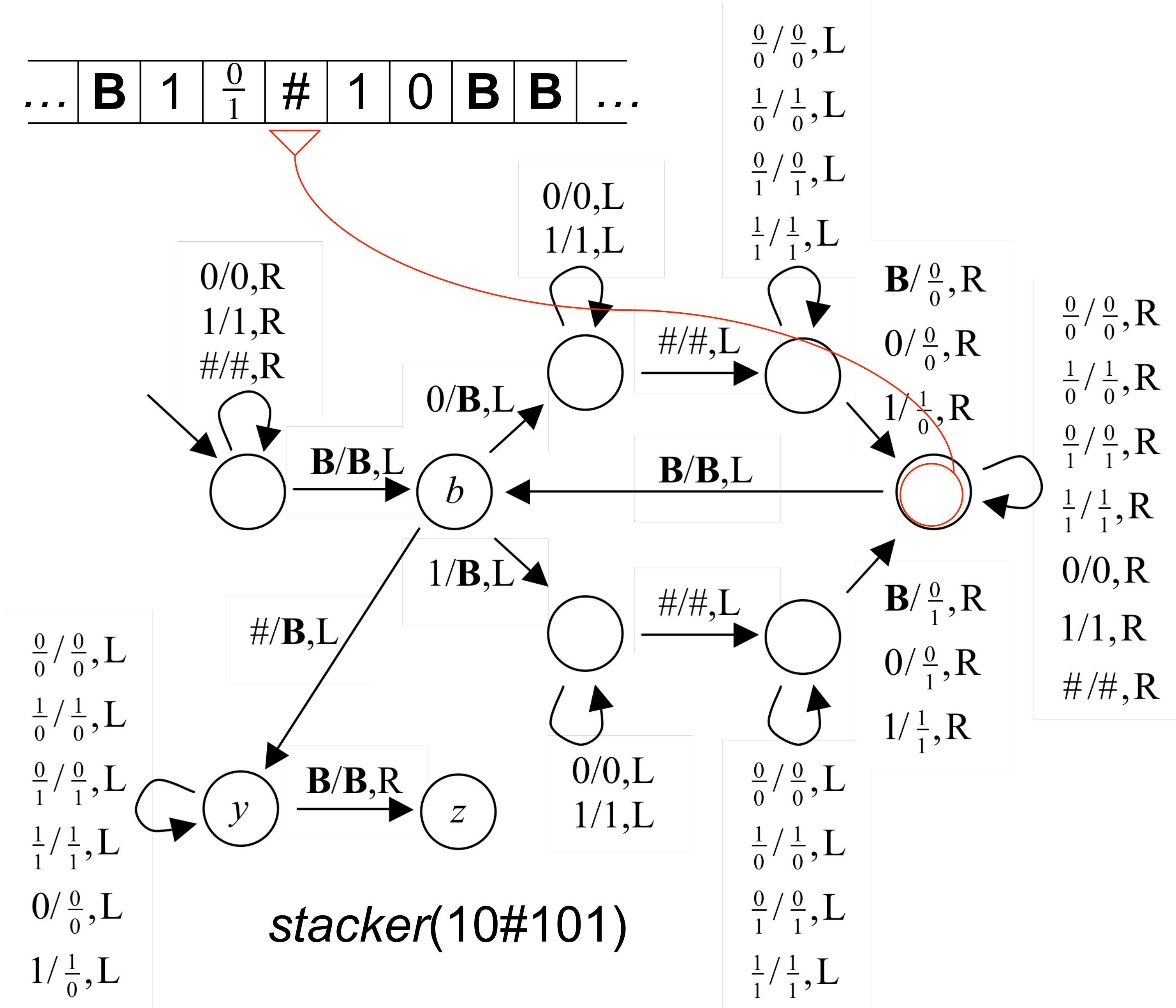
stacker(10#101)



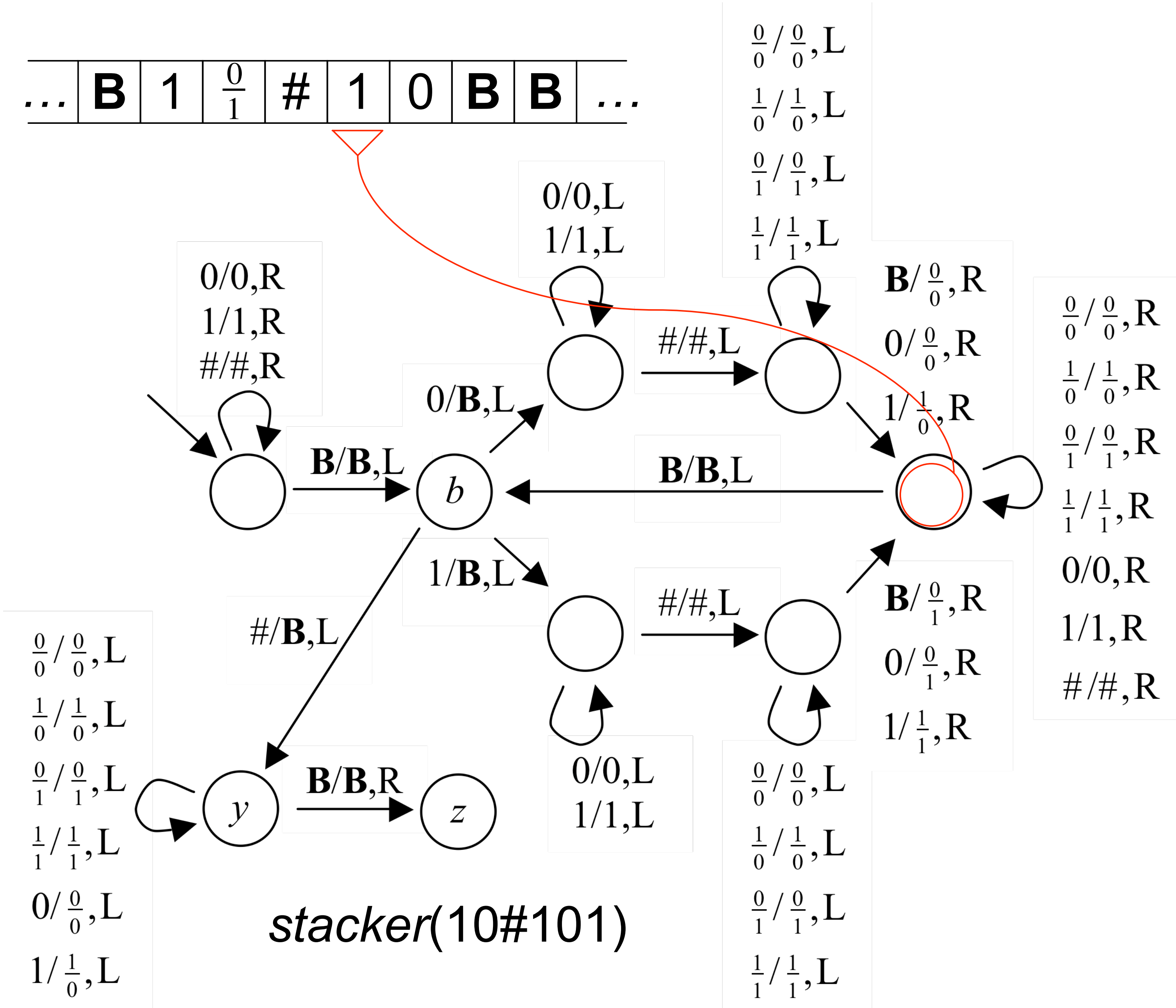
stacker(10#101)



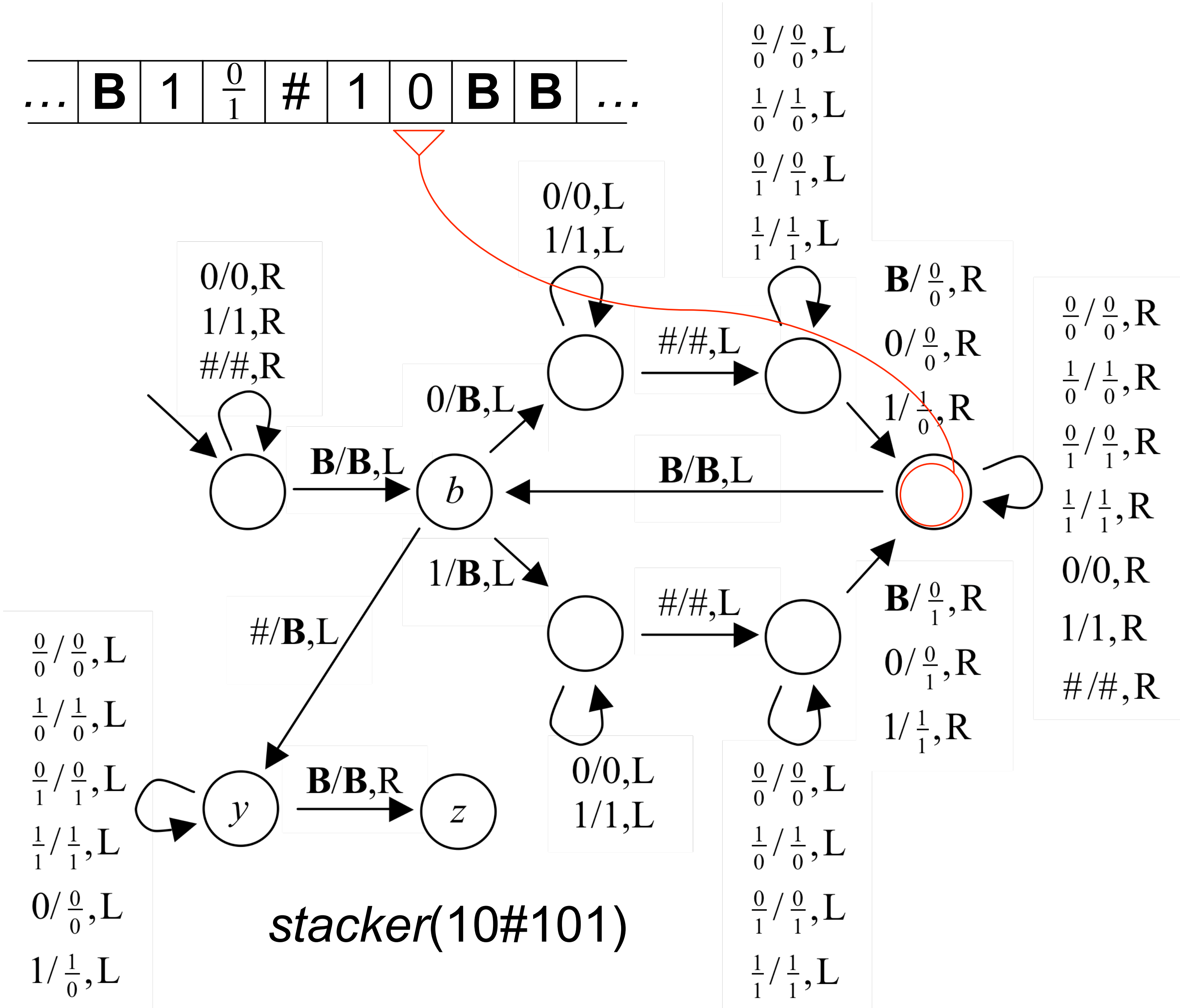
stacker(10#101)



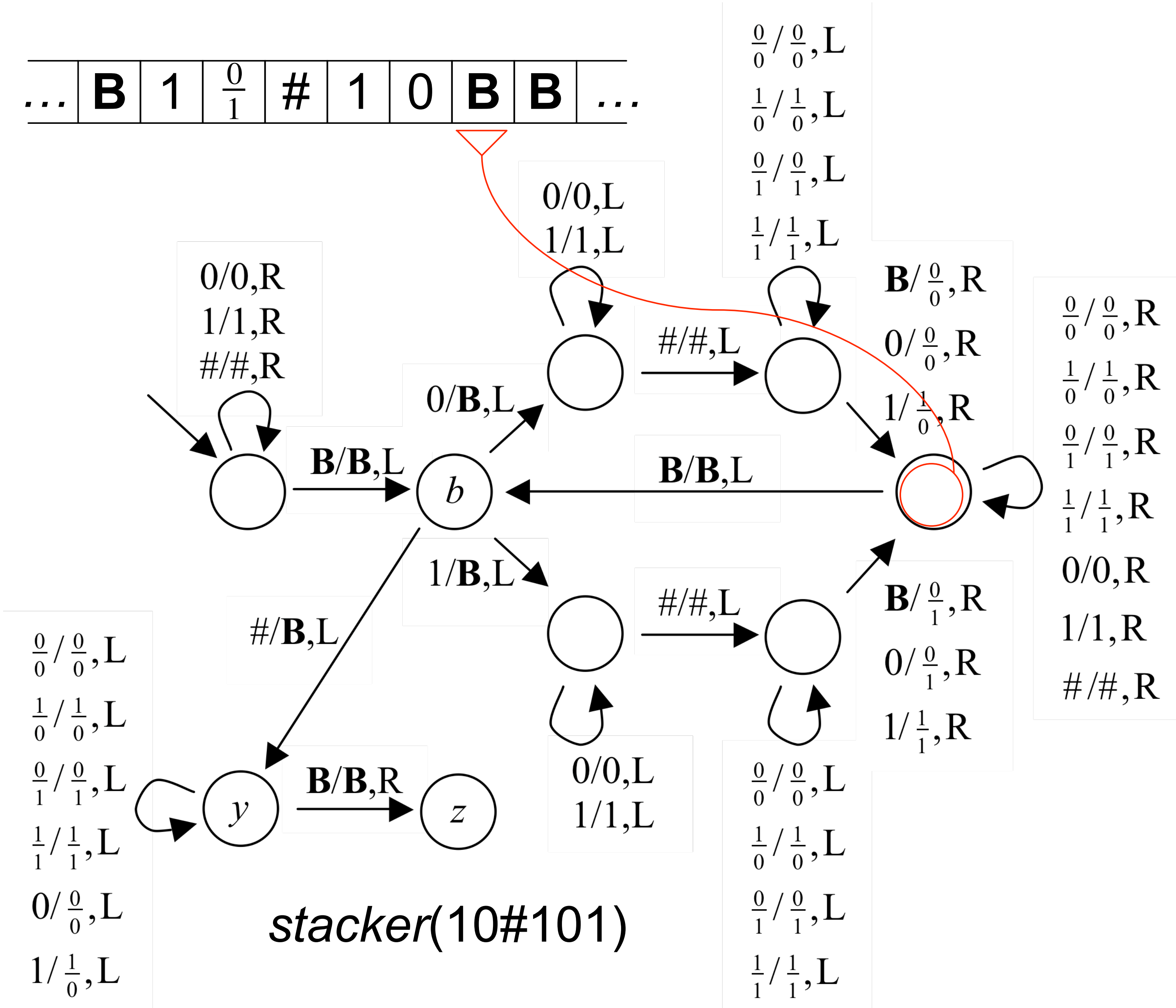
stacker(10#101)



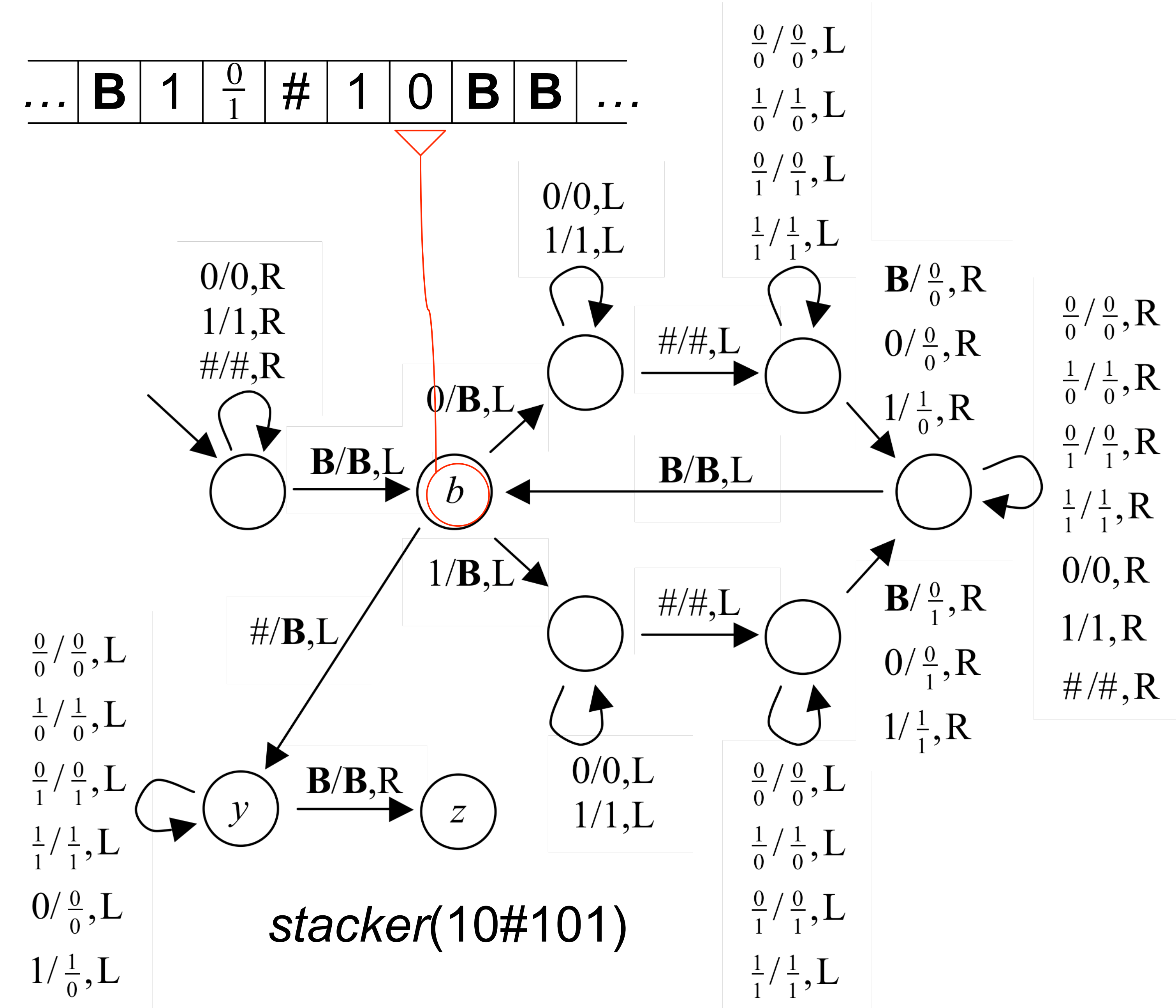
stacker(10#101)



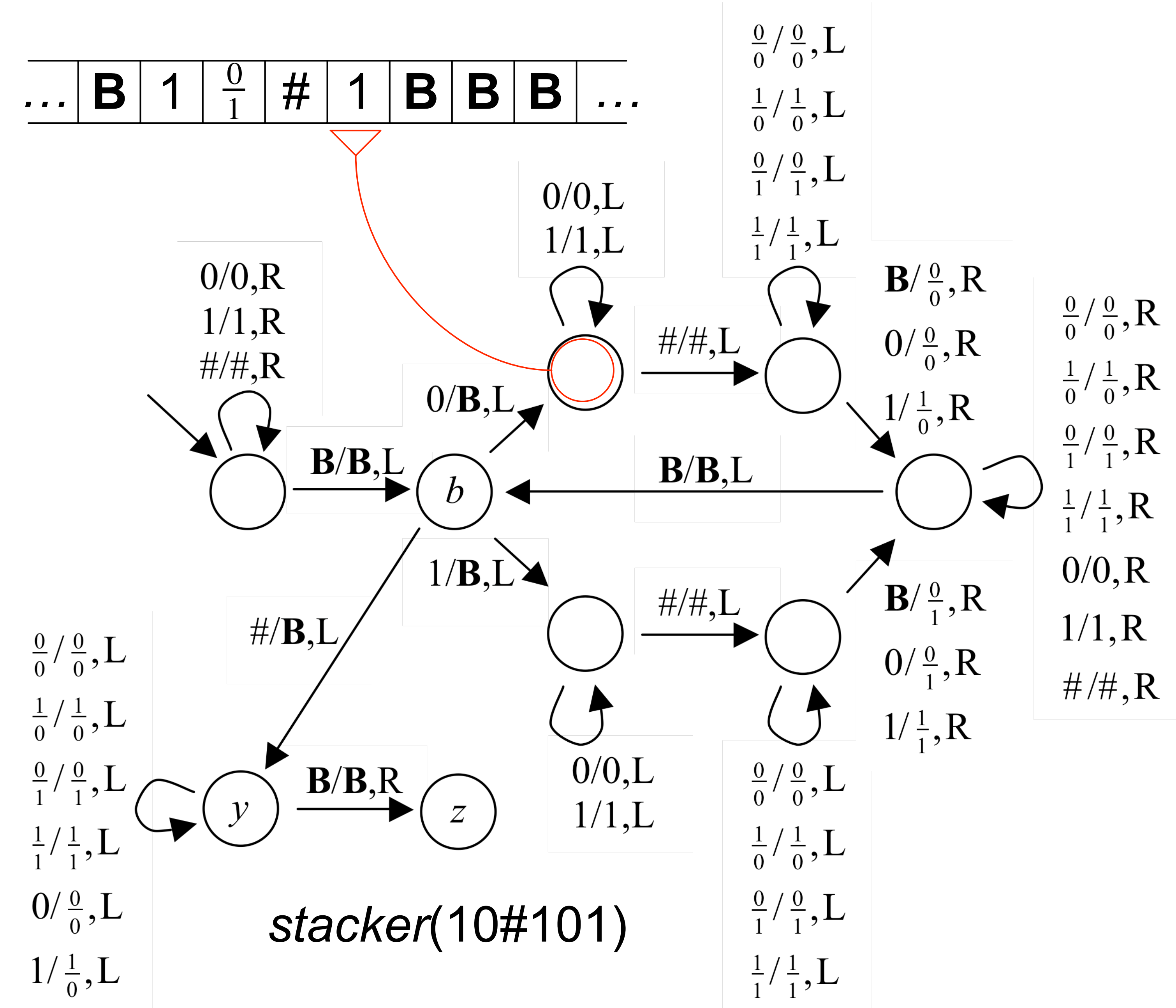
stacker(10#101)



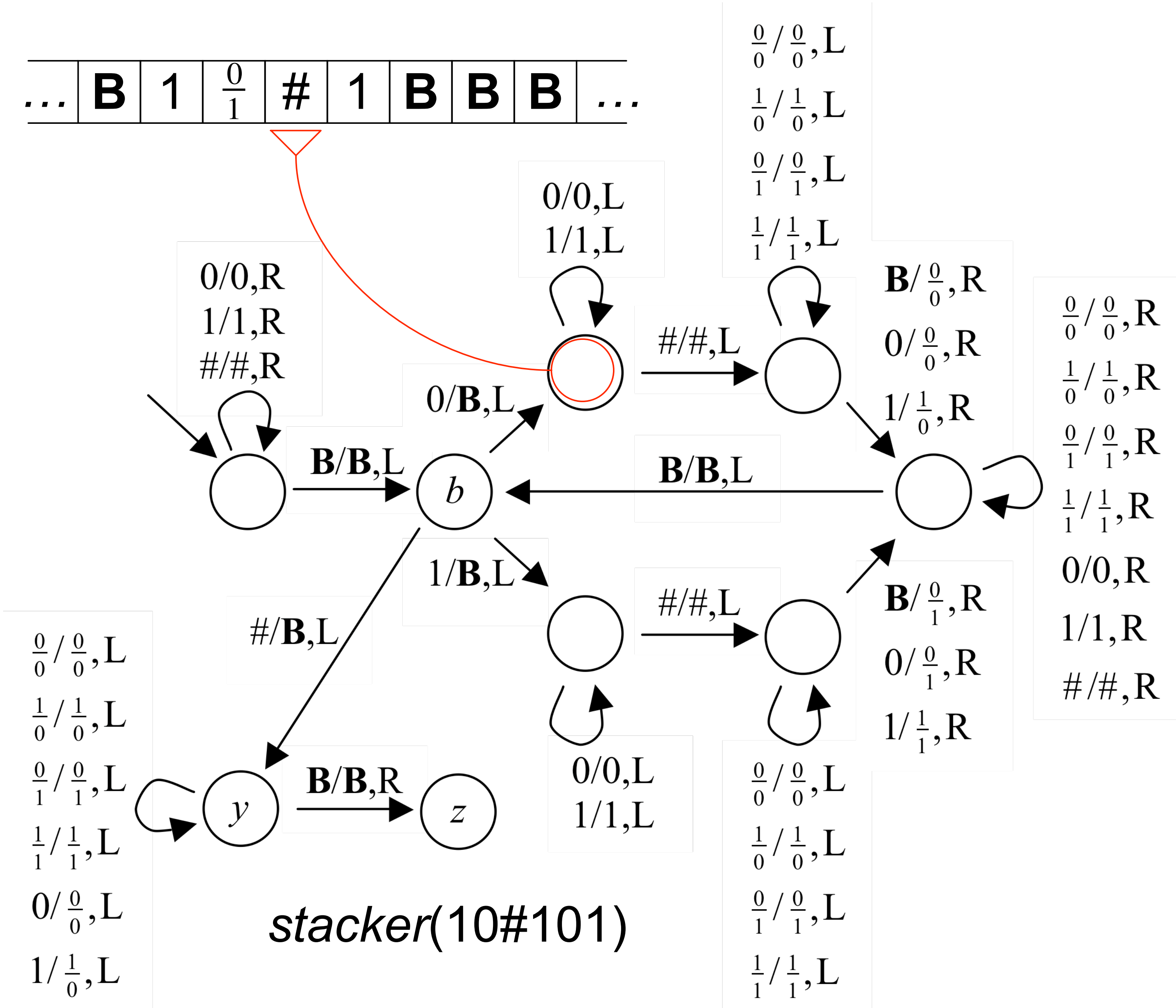
stacker(10#101)



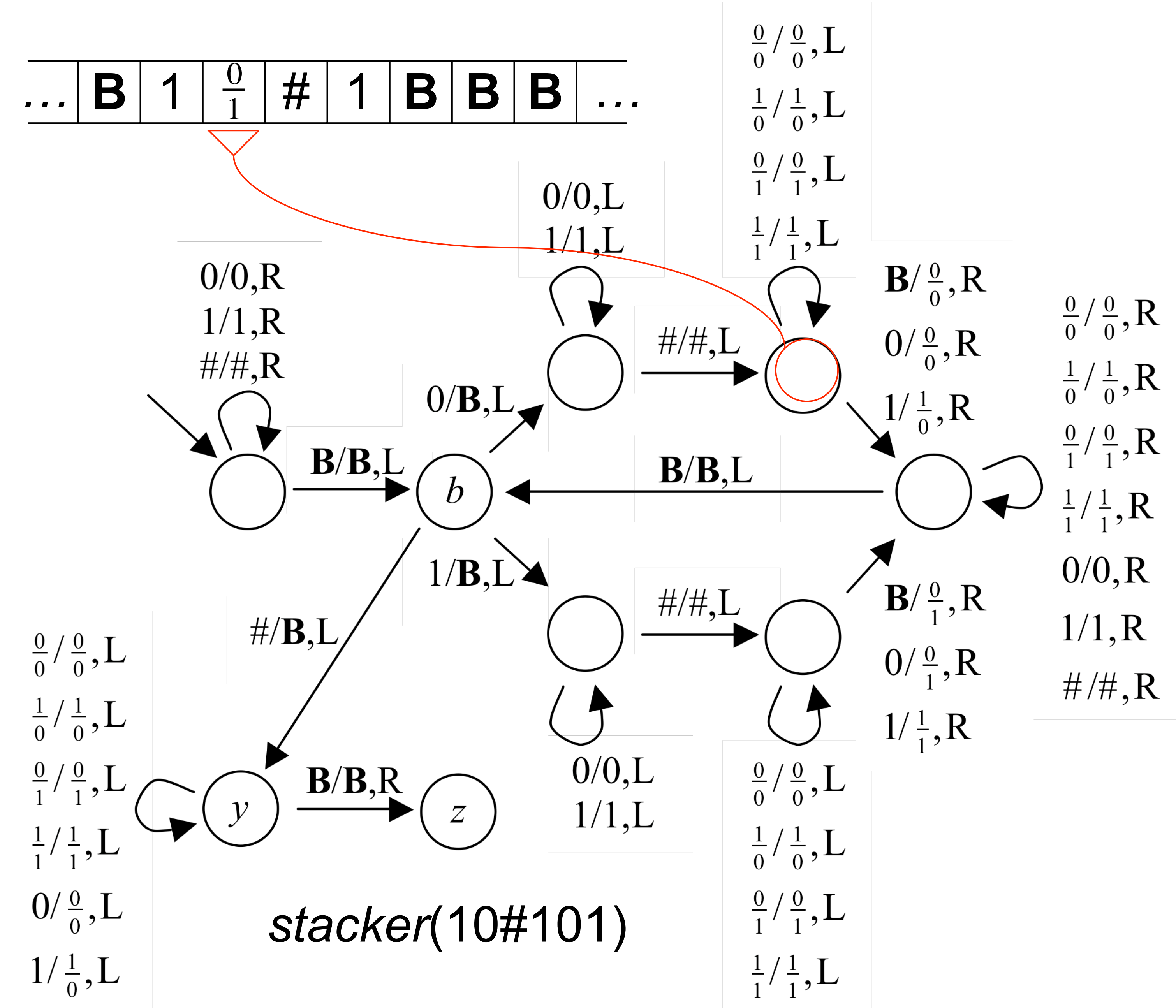
stacker(10#101)



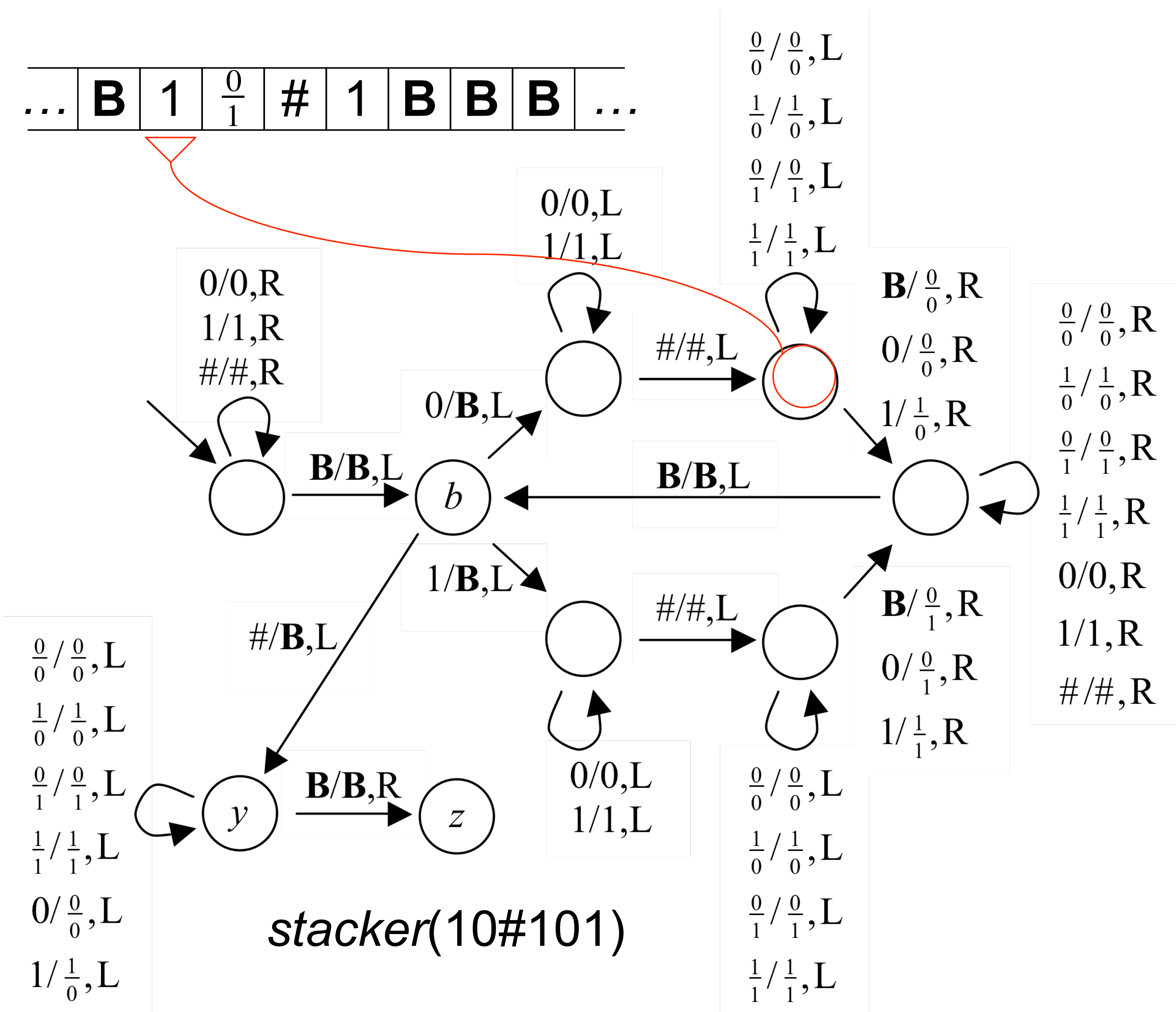
stacker(10#101)



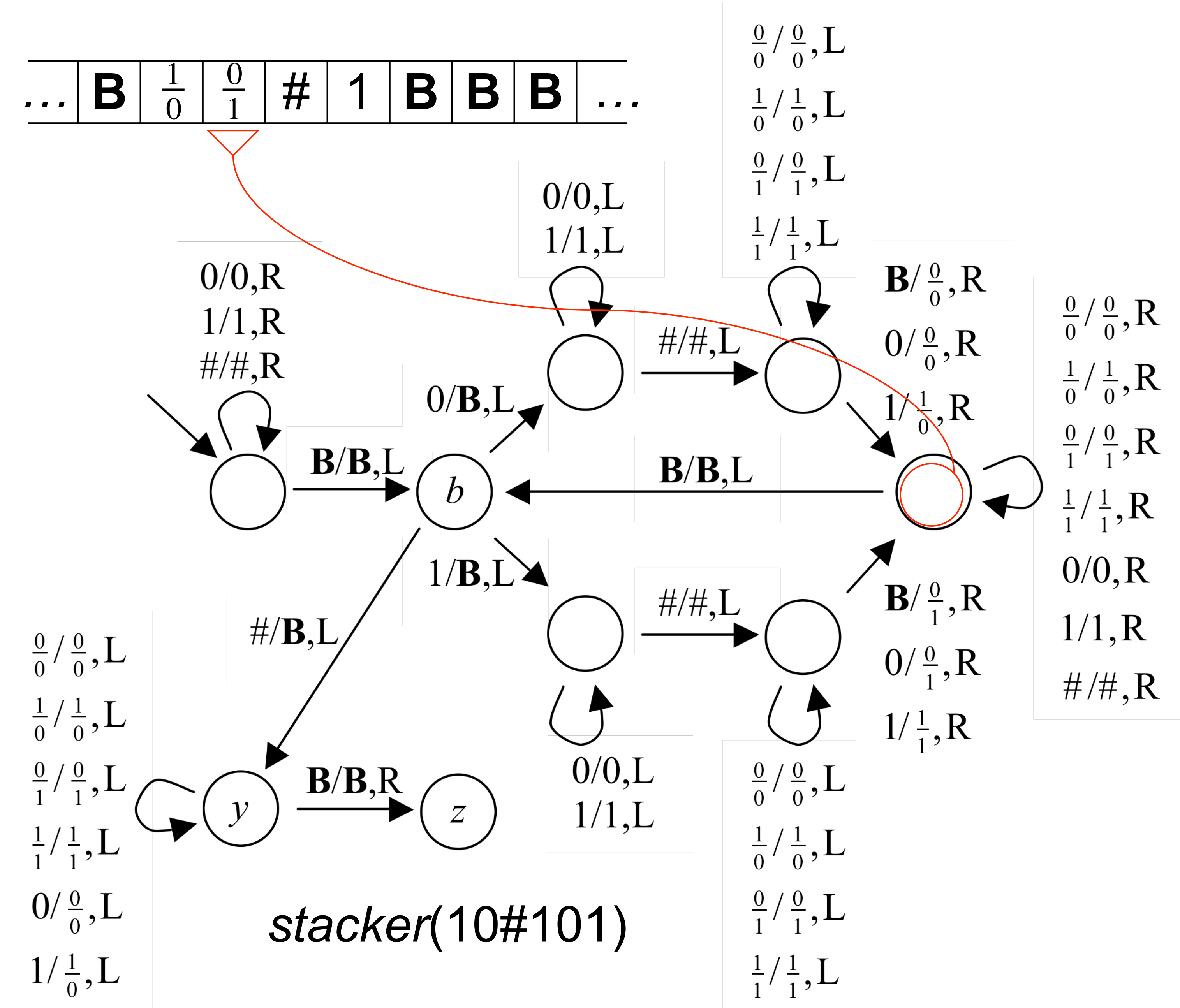
stacker(10#101)



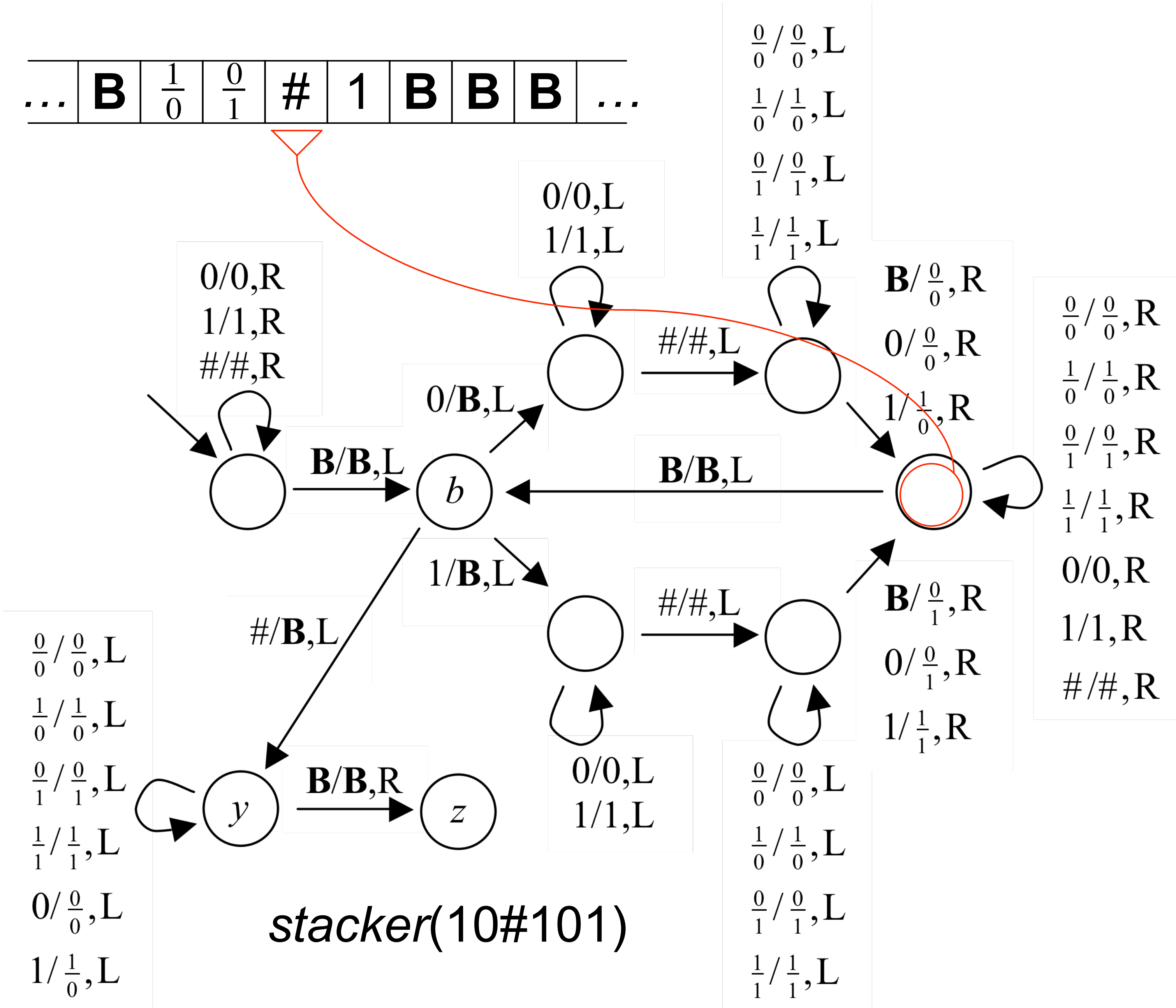
stacker(10#101)



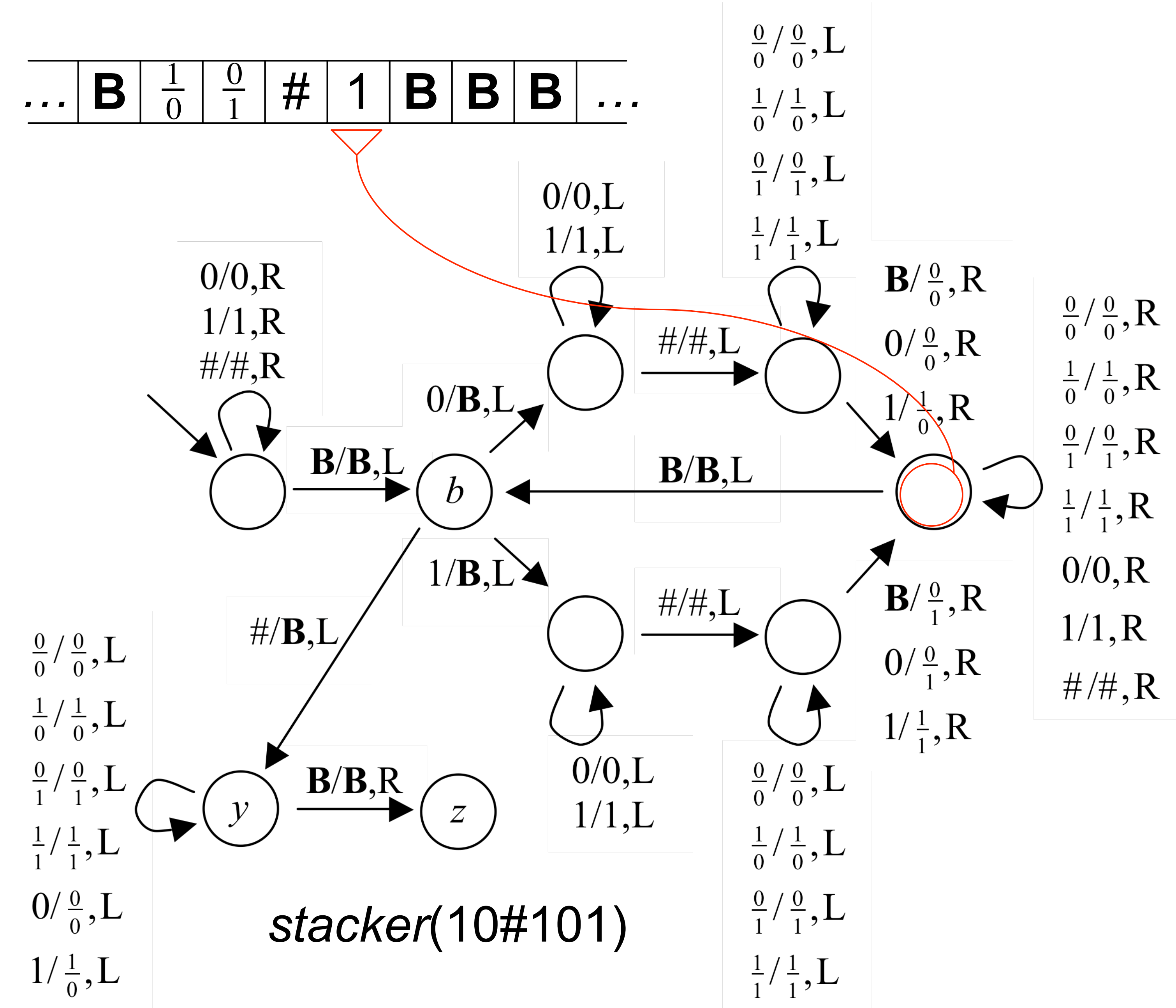
stacker(10#101)



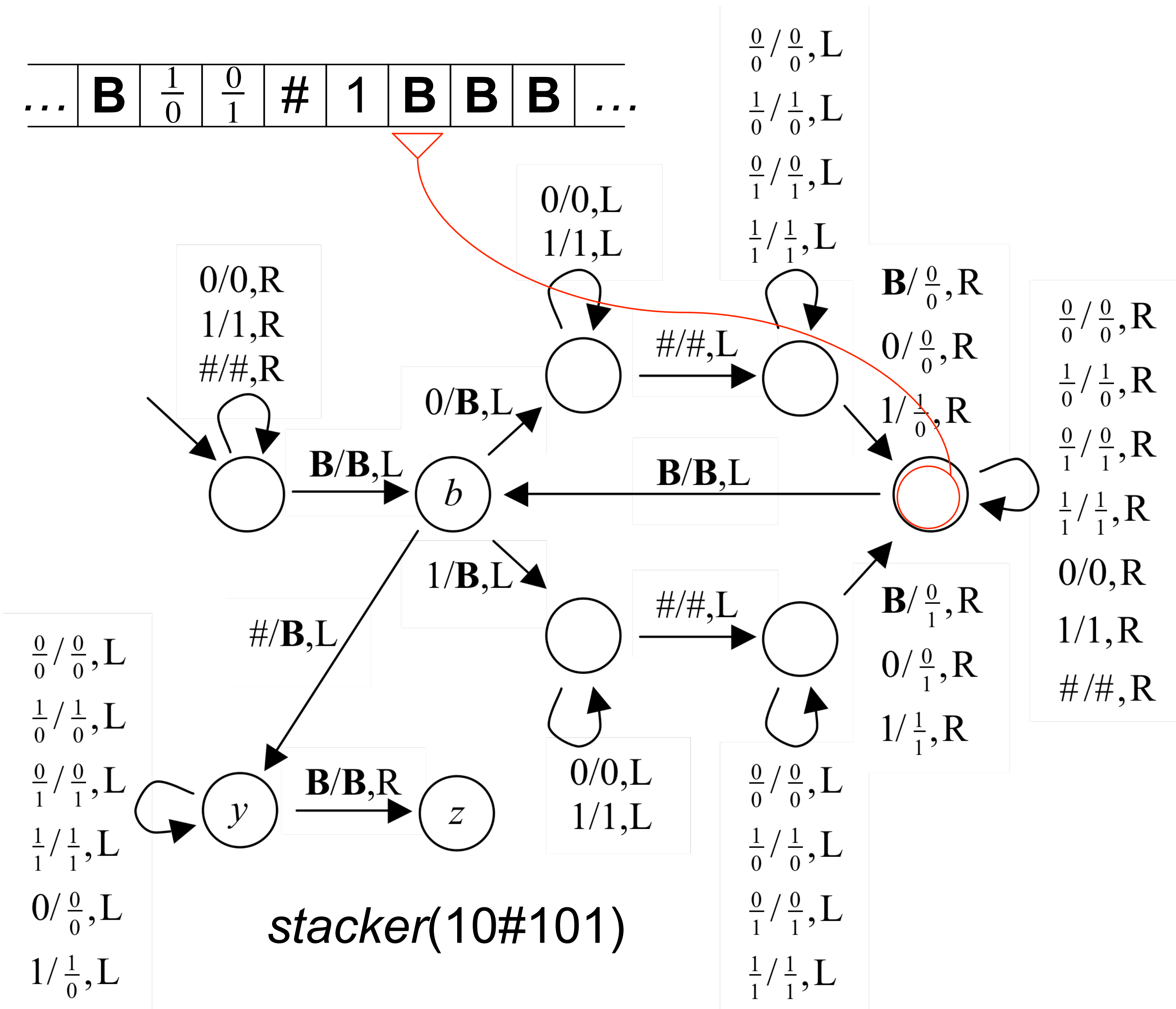
stacker(10#101)



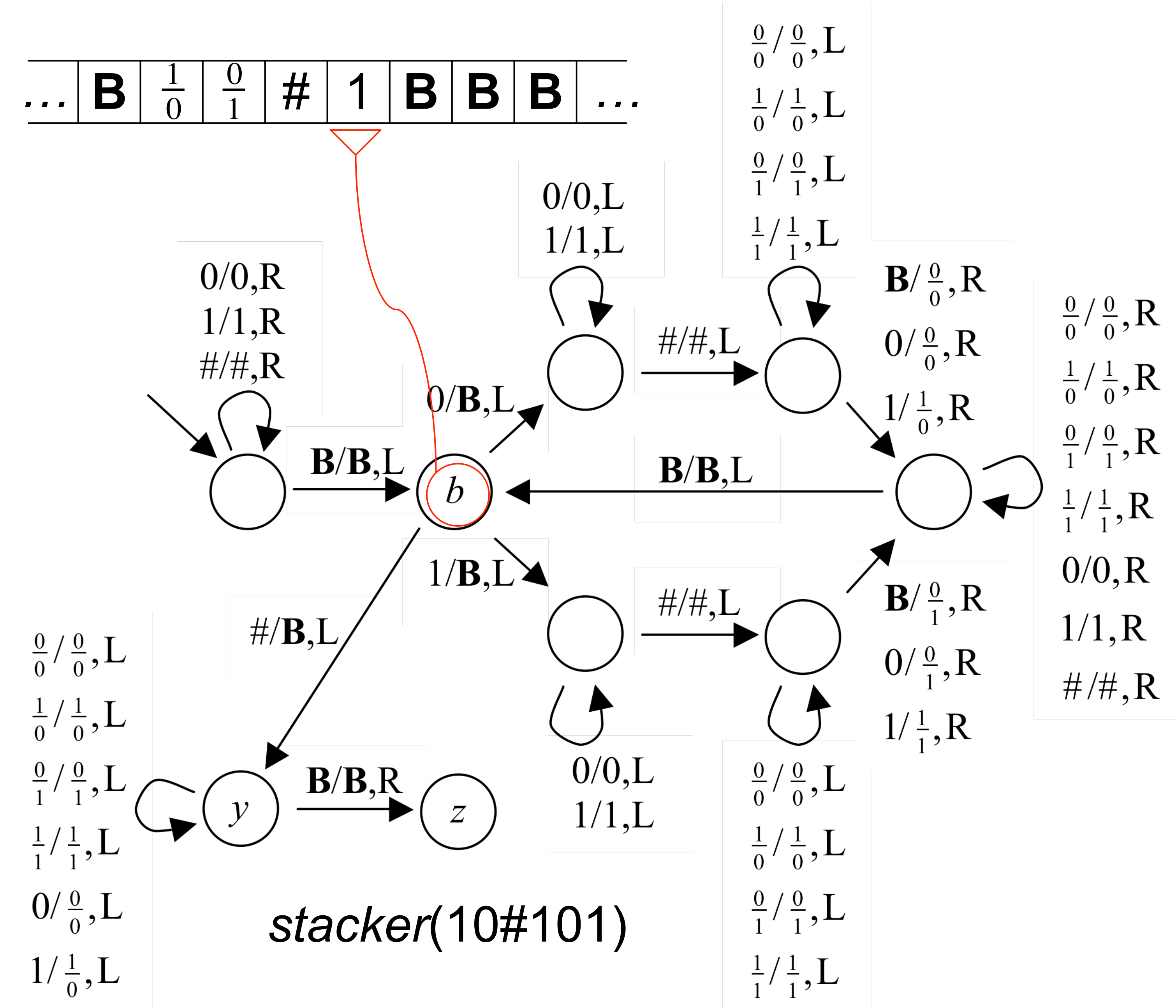
stacker(10#101)



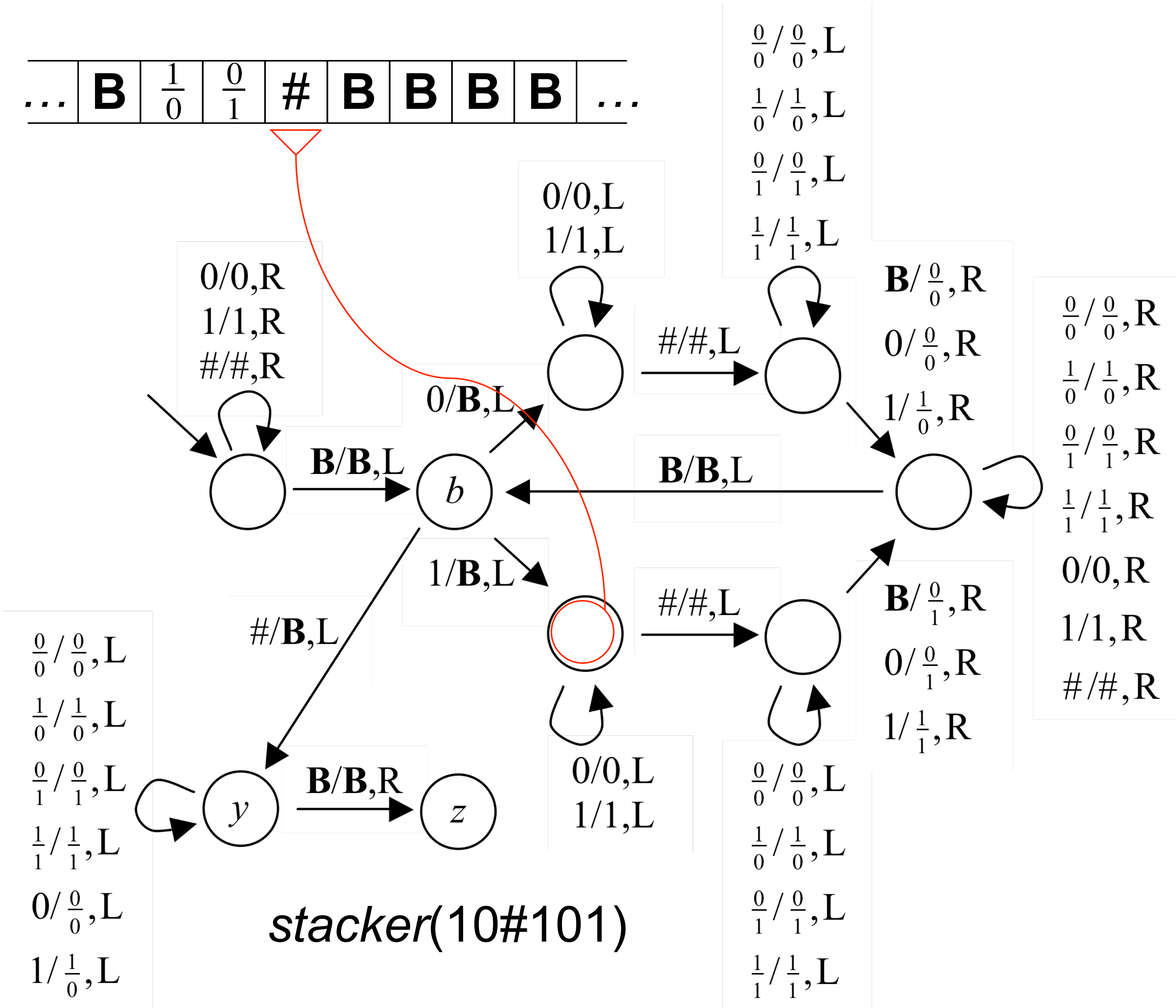
stacker(10#101)



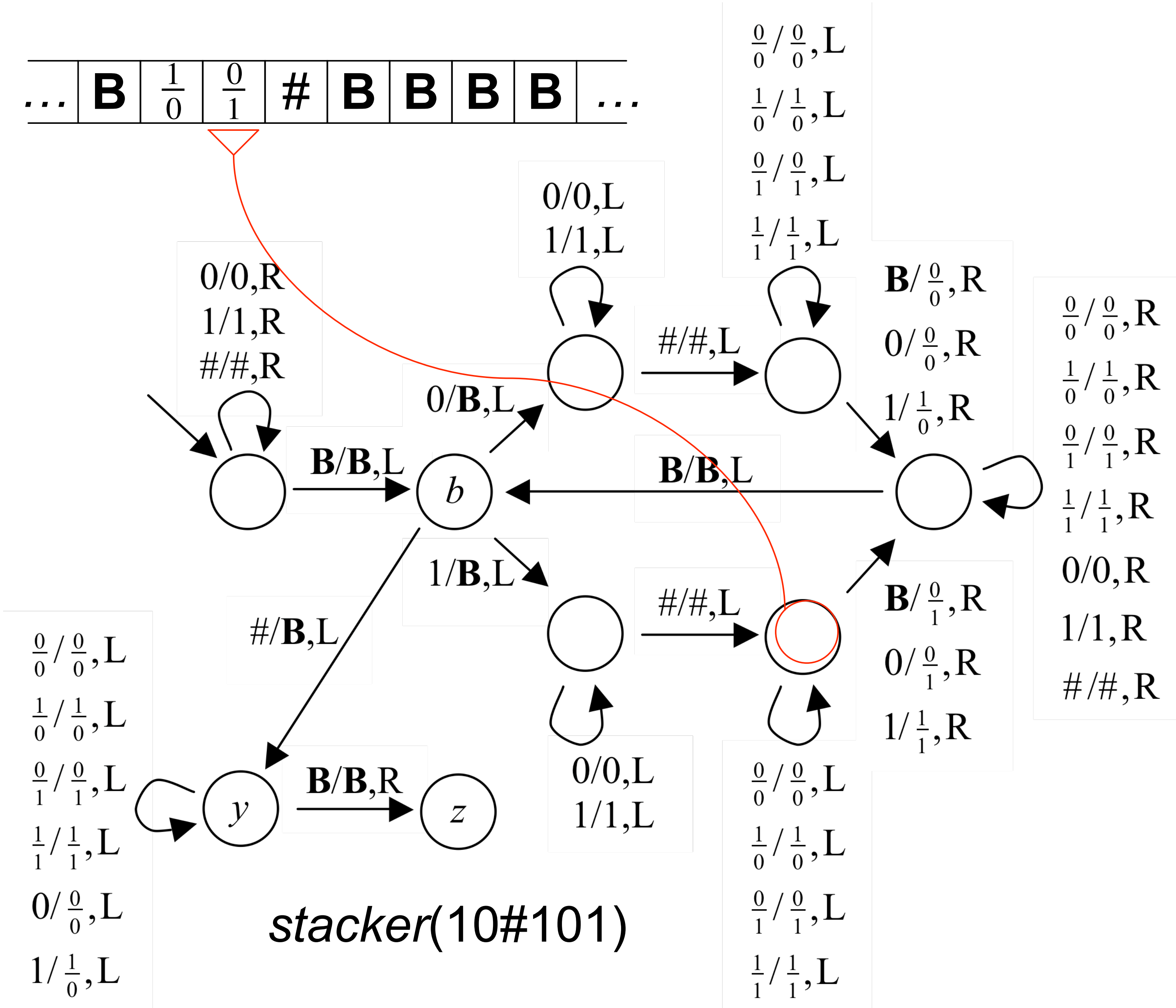
stacker(10#101)



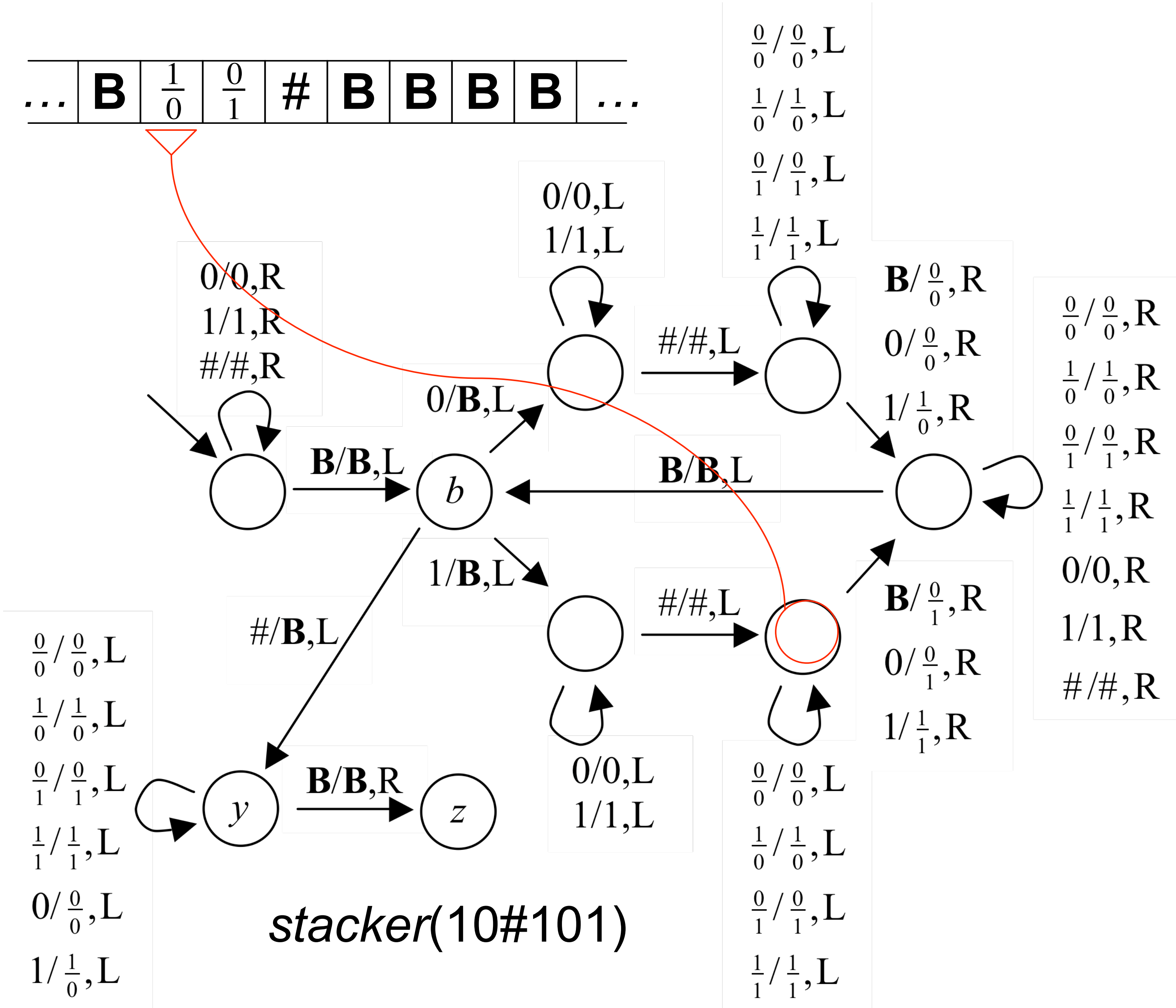
stacker(10#101)



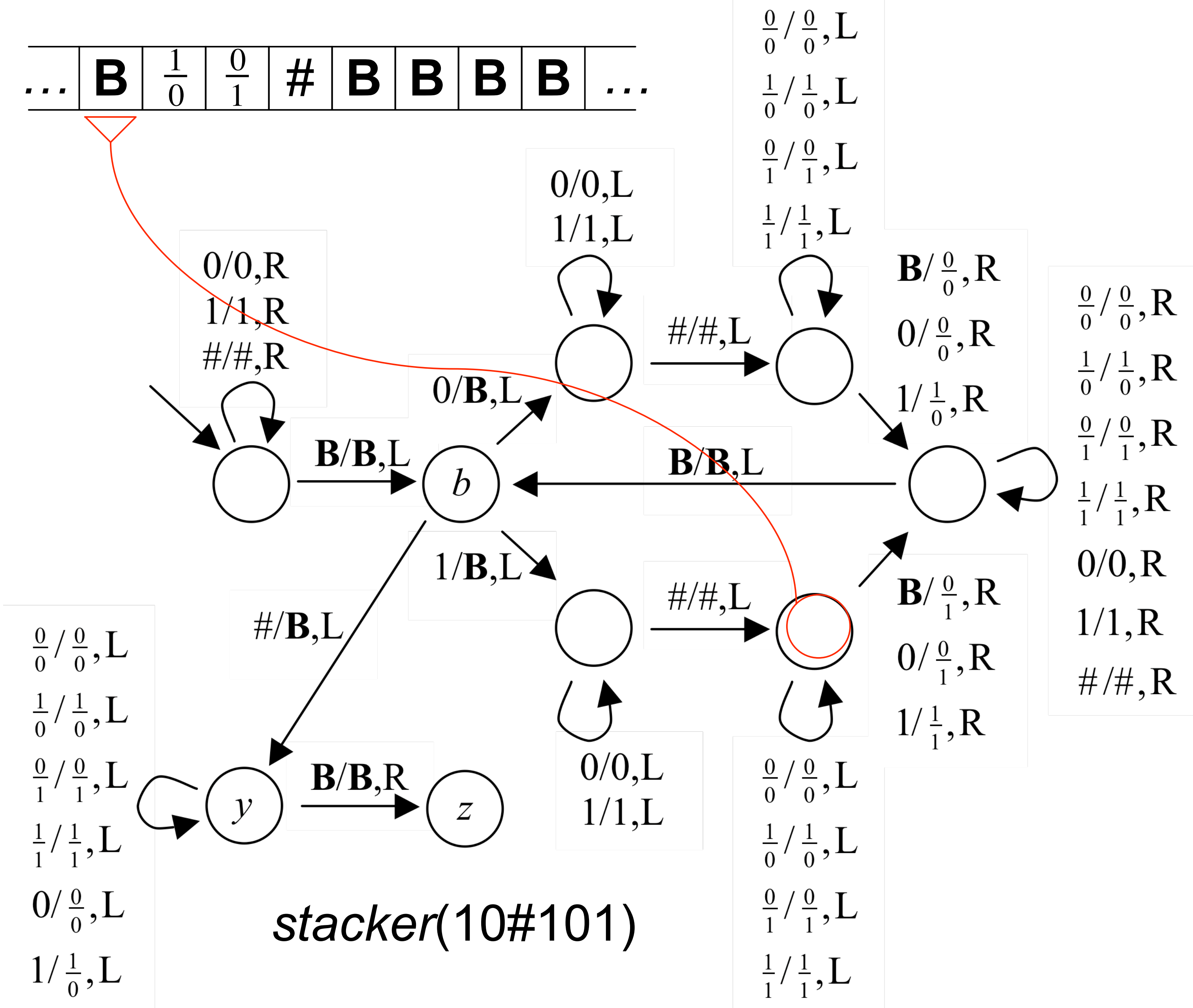
stacker(10#101)



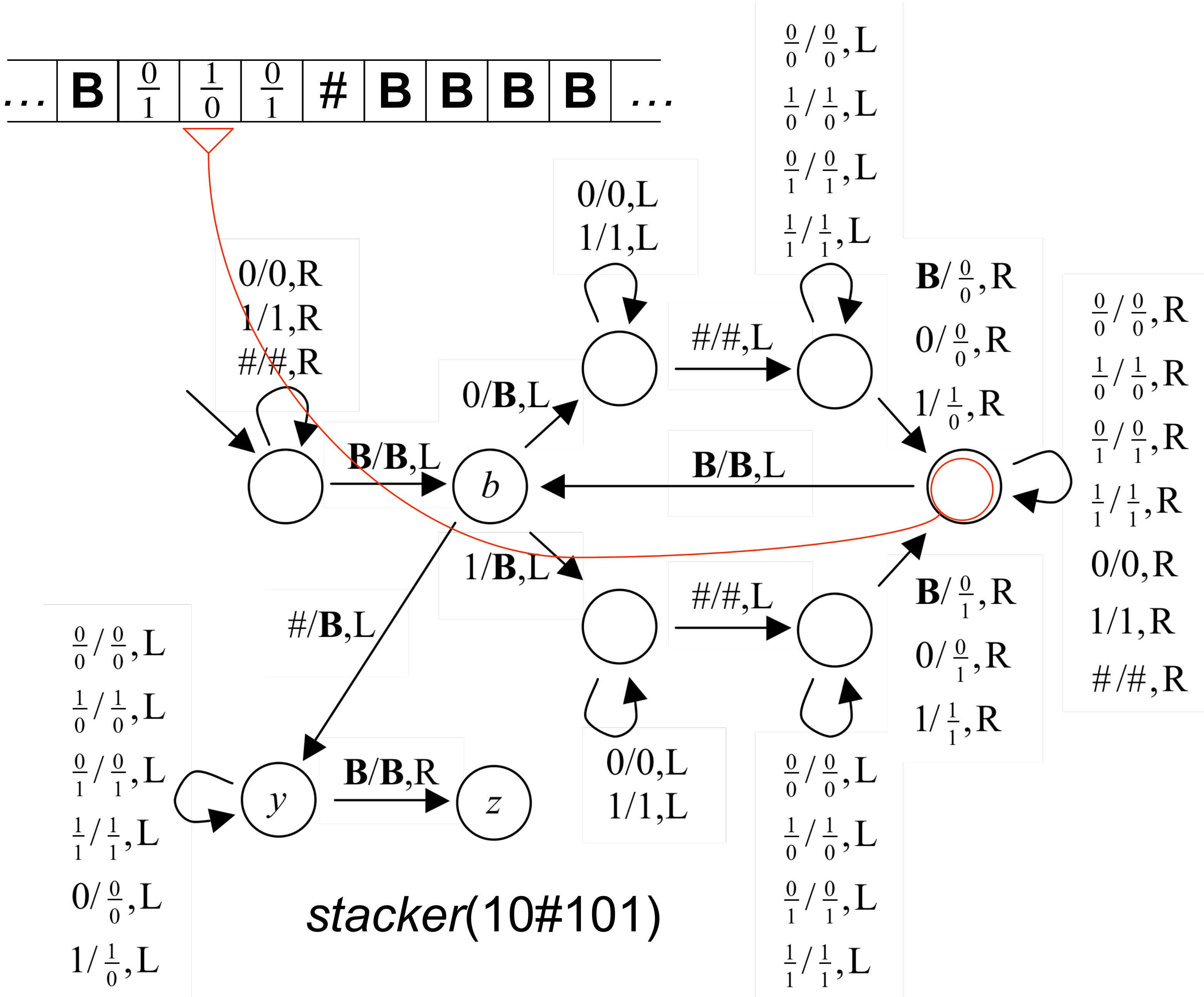
stacker(10#101)



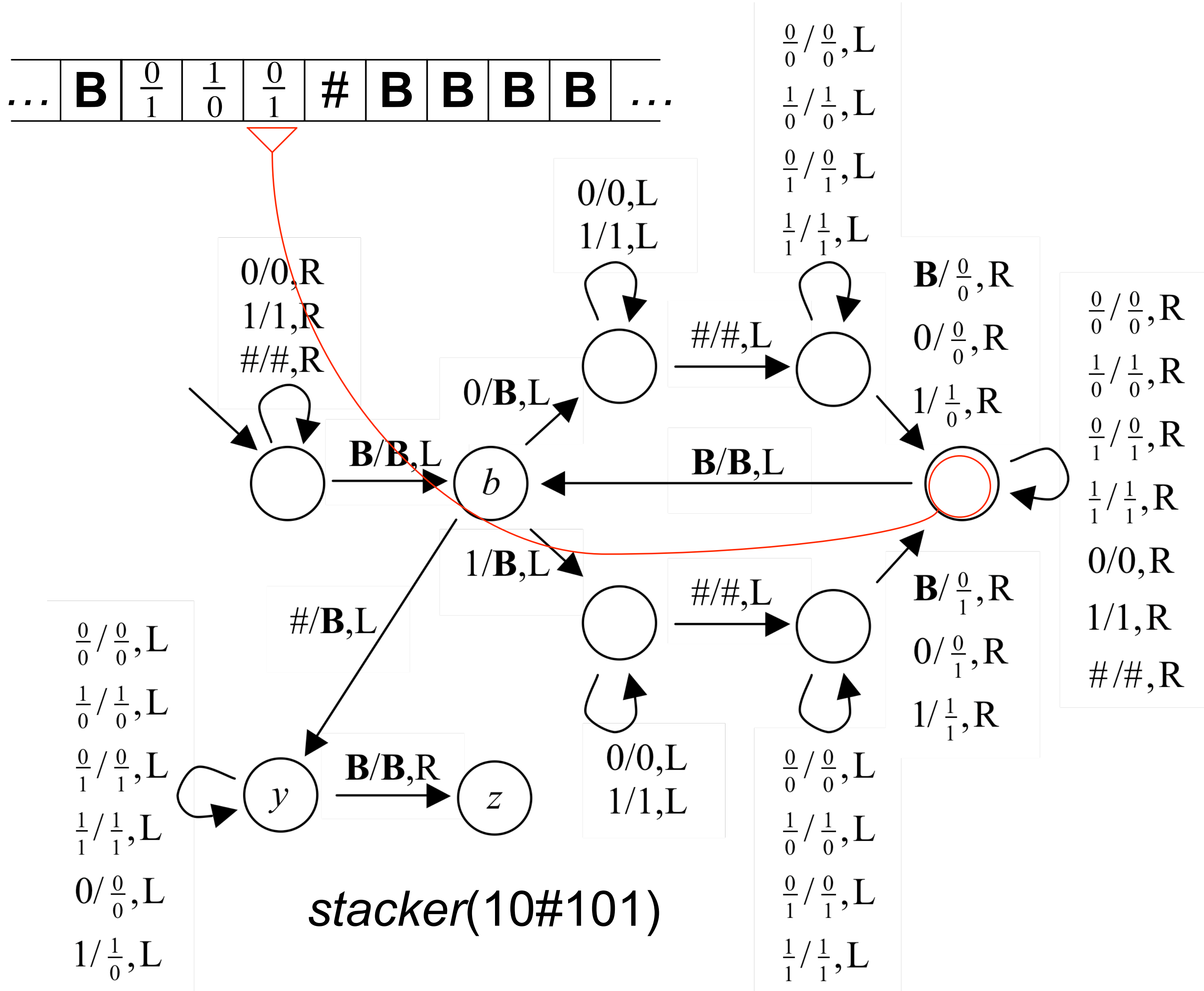
stacker(10#101)



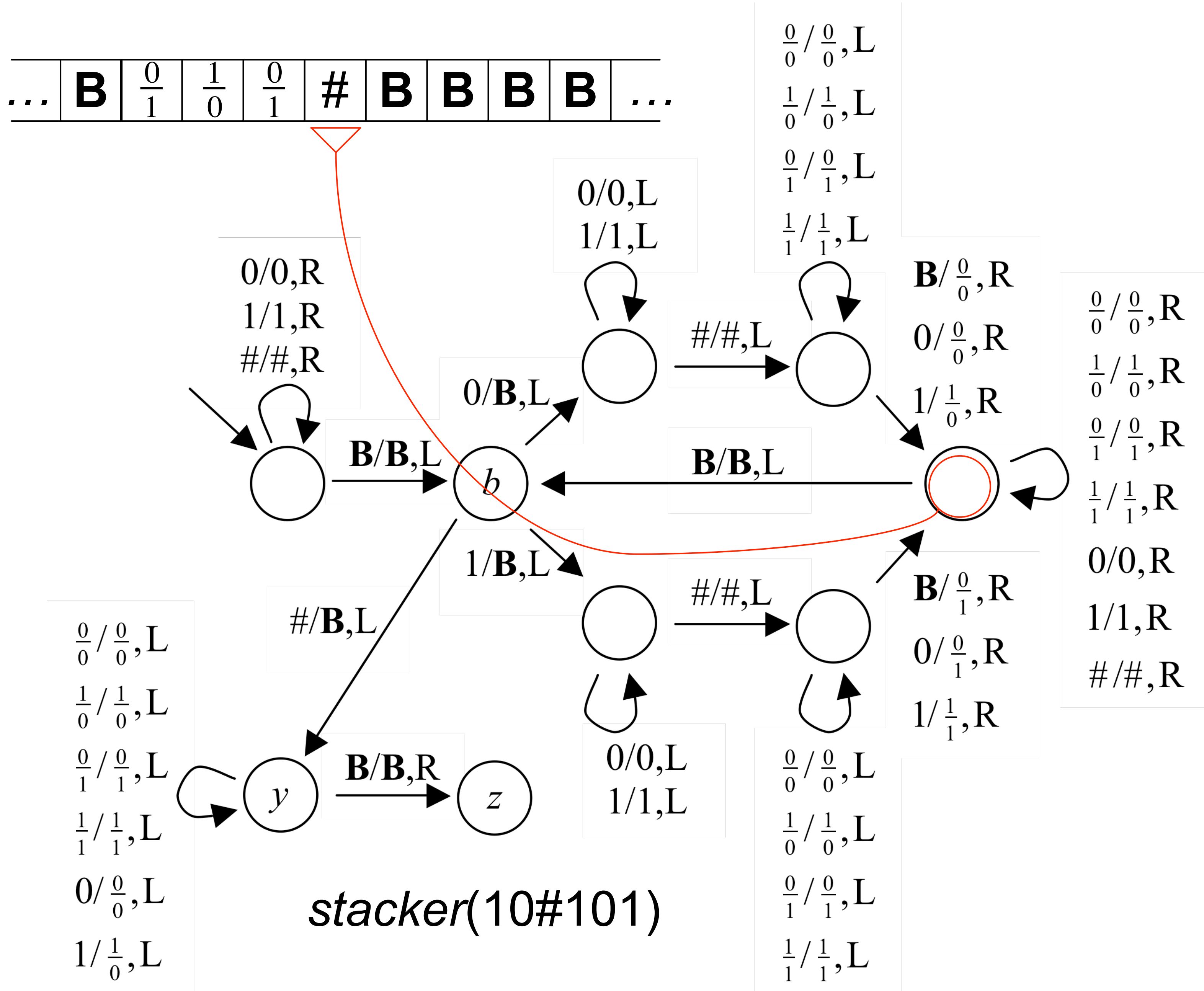
stacker(10#101)



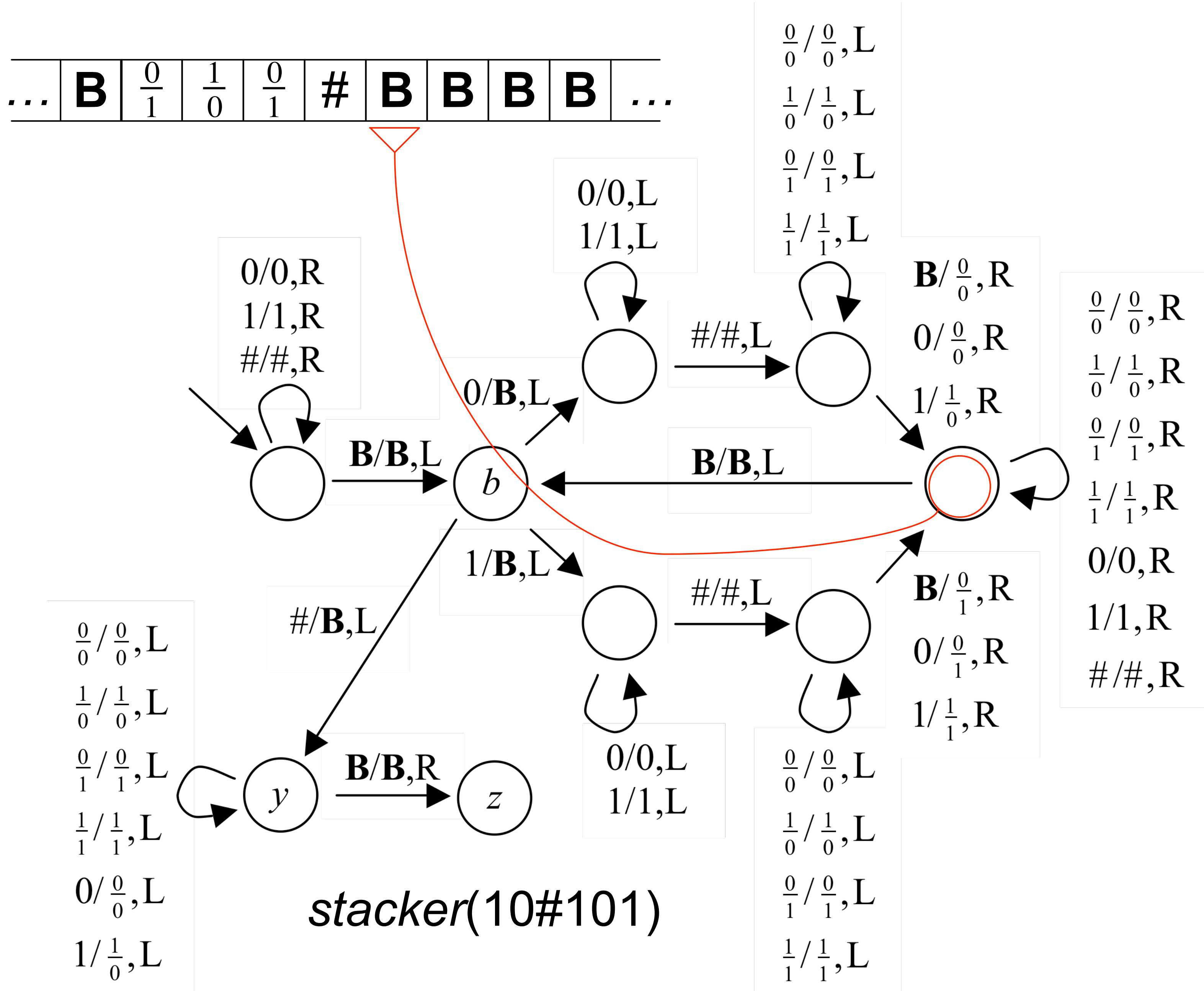
stacker(10#101)



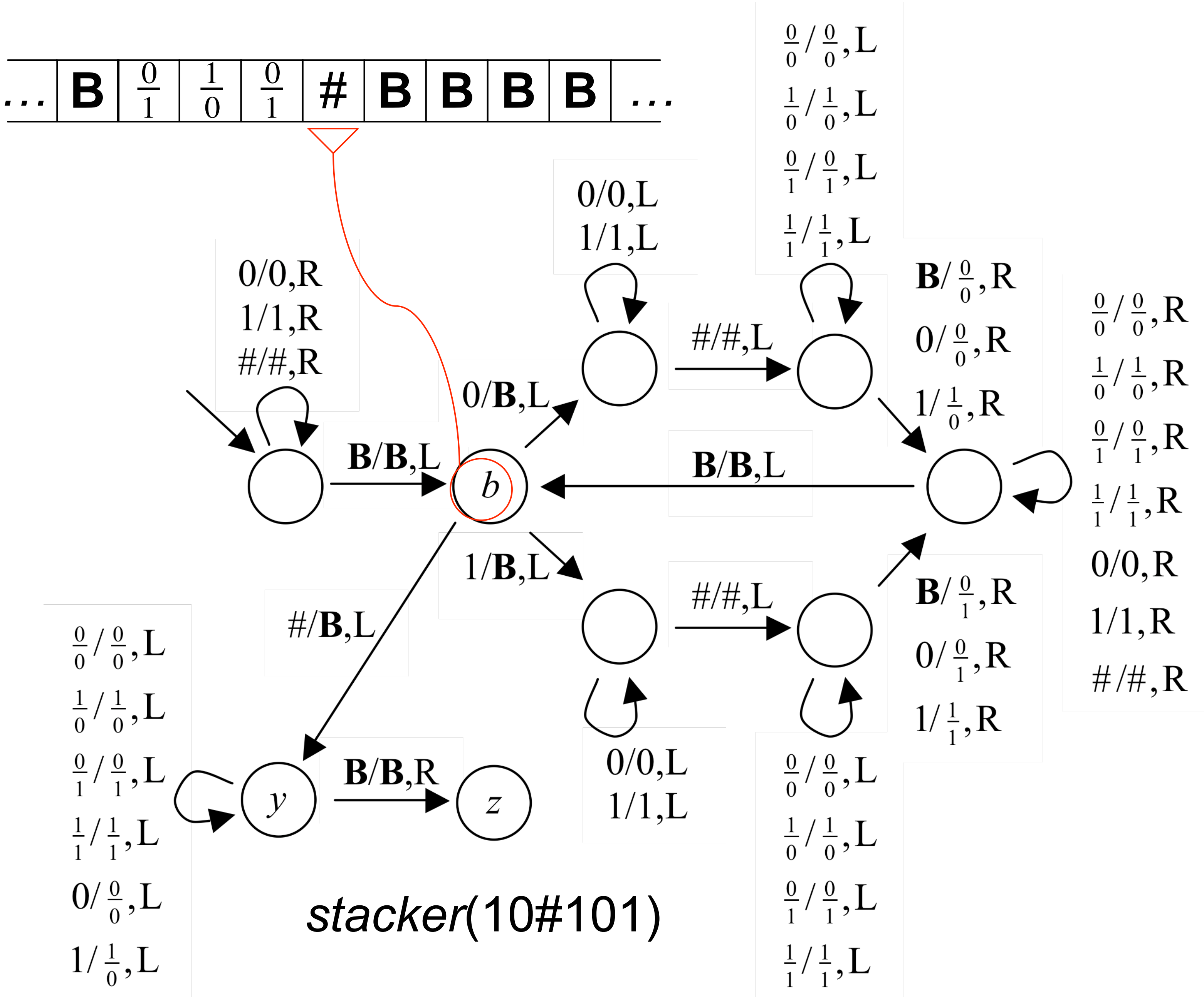
stacker(10#101)



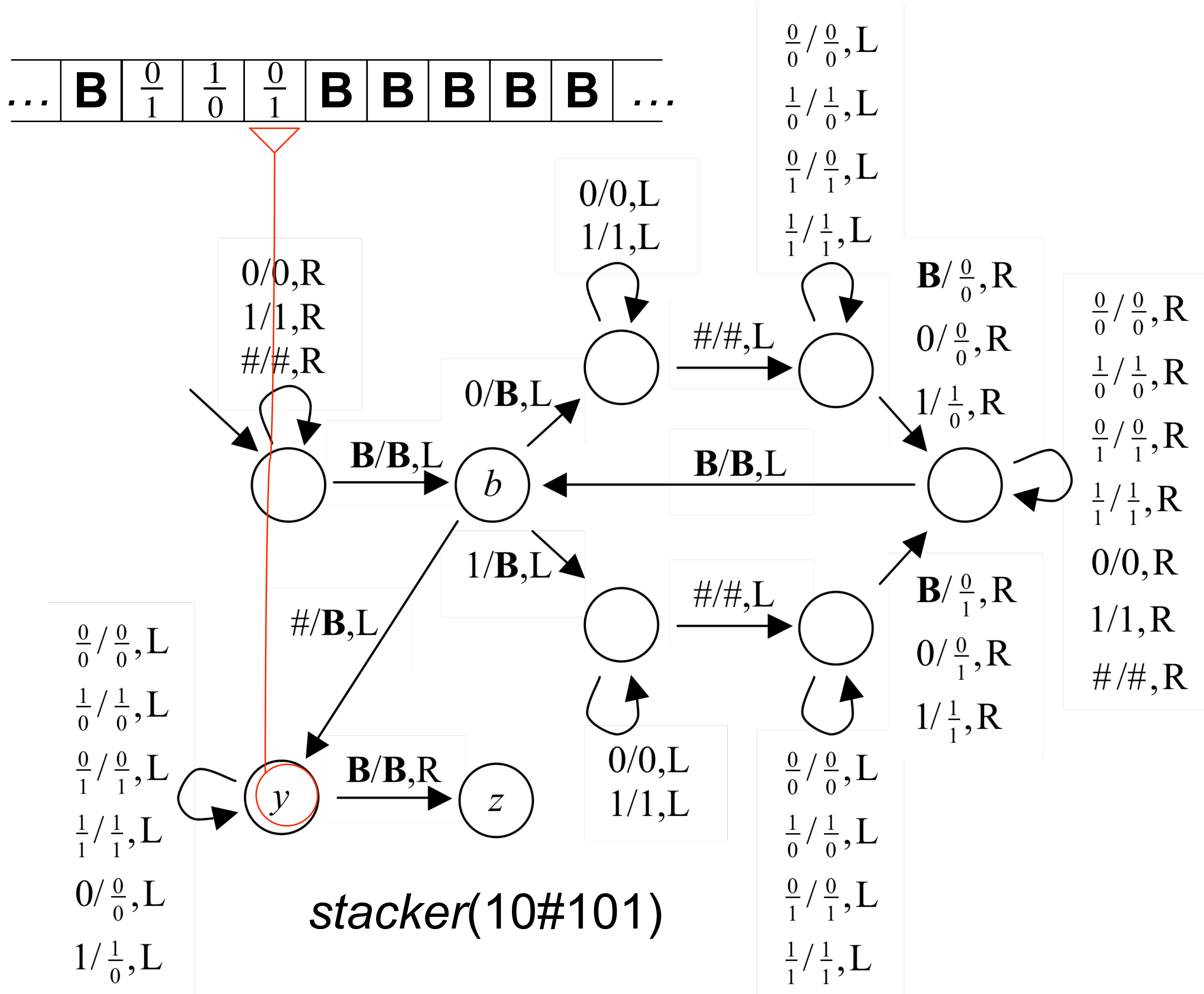
stacker(10#101)



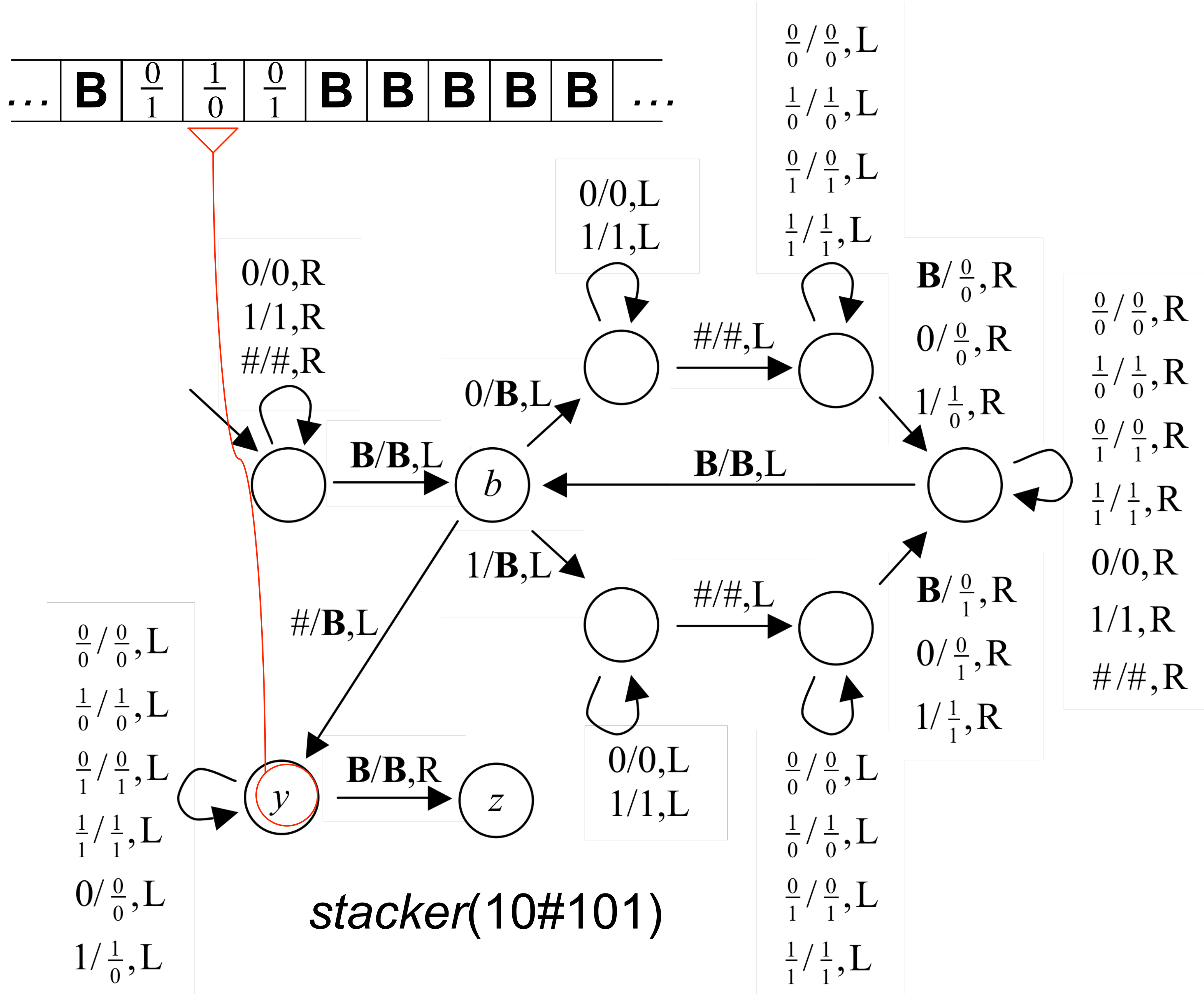
stacker(10#101)



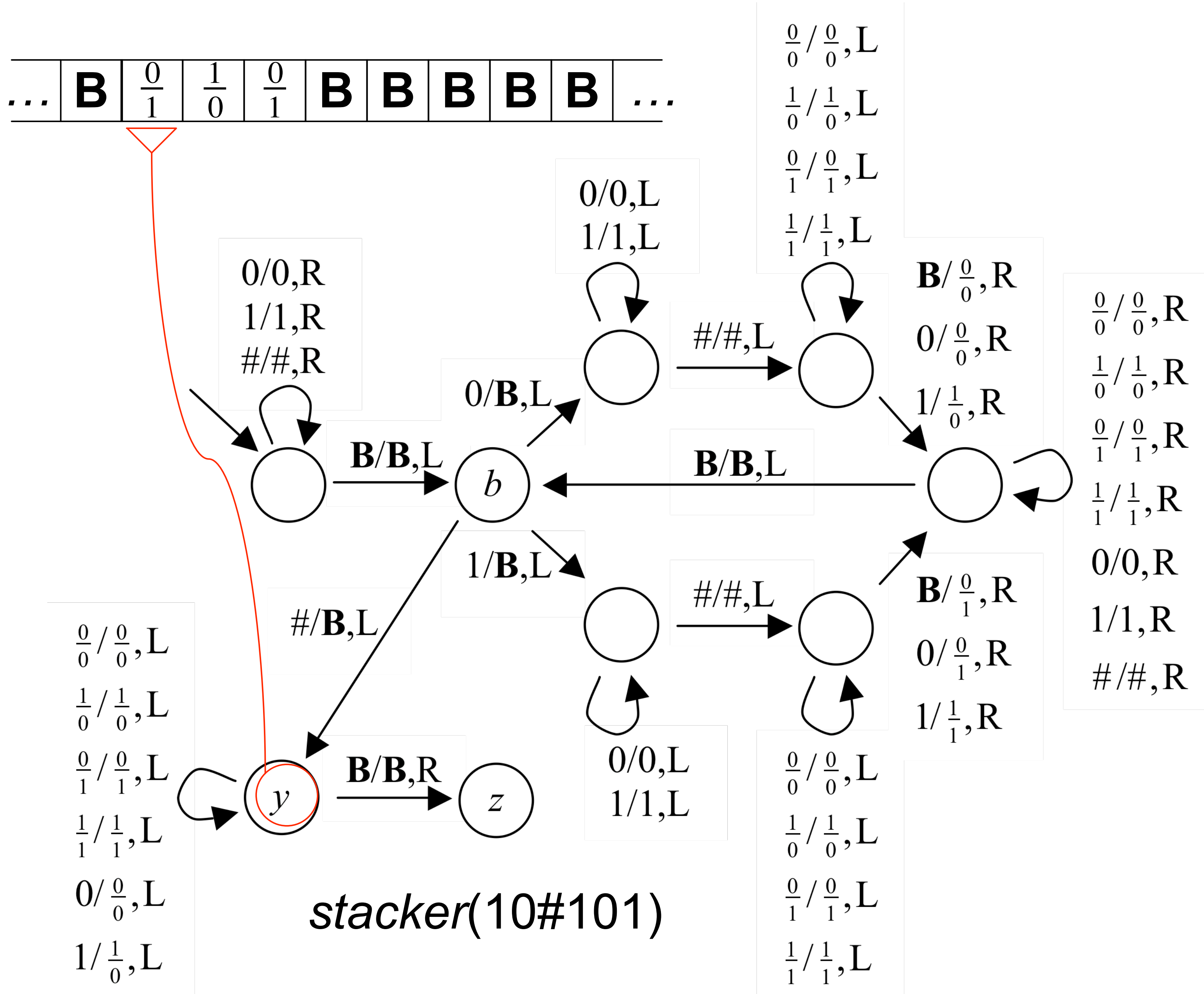
stacker(10#101)



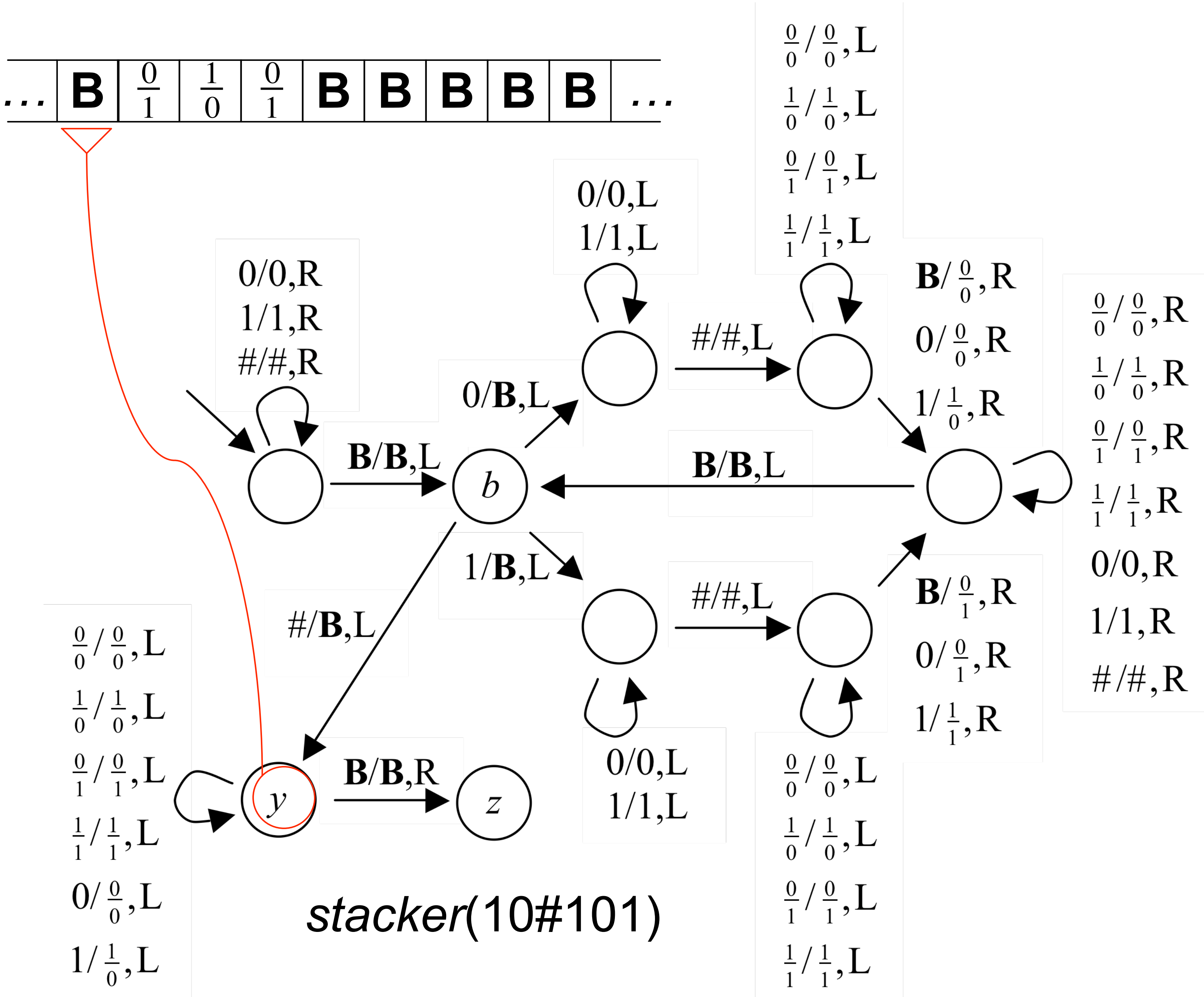
stacker(10#101)



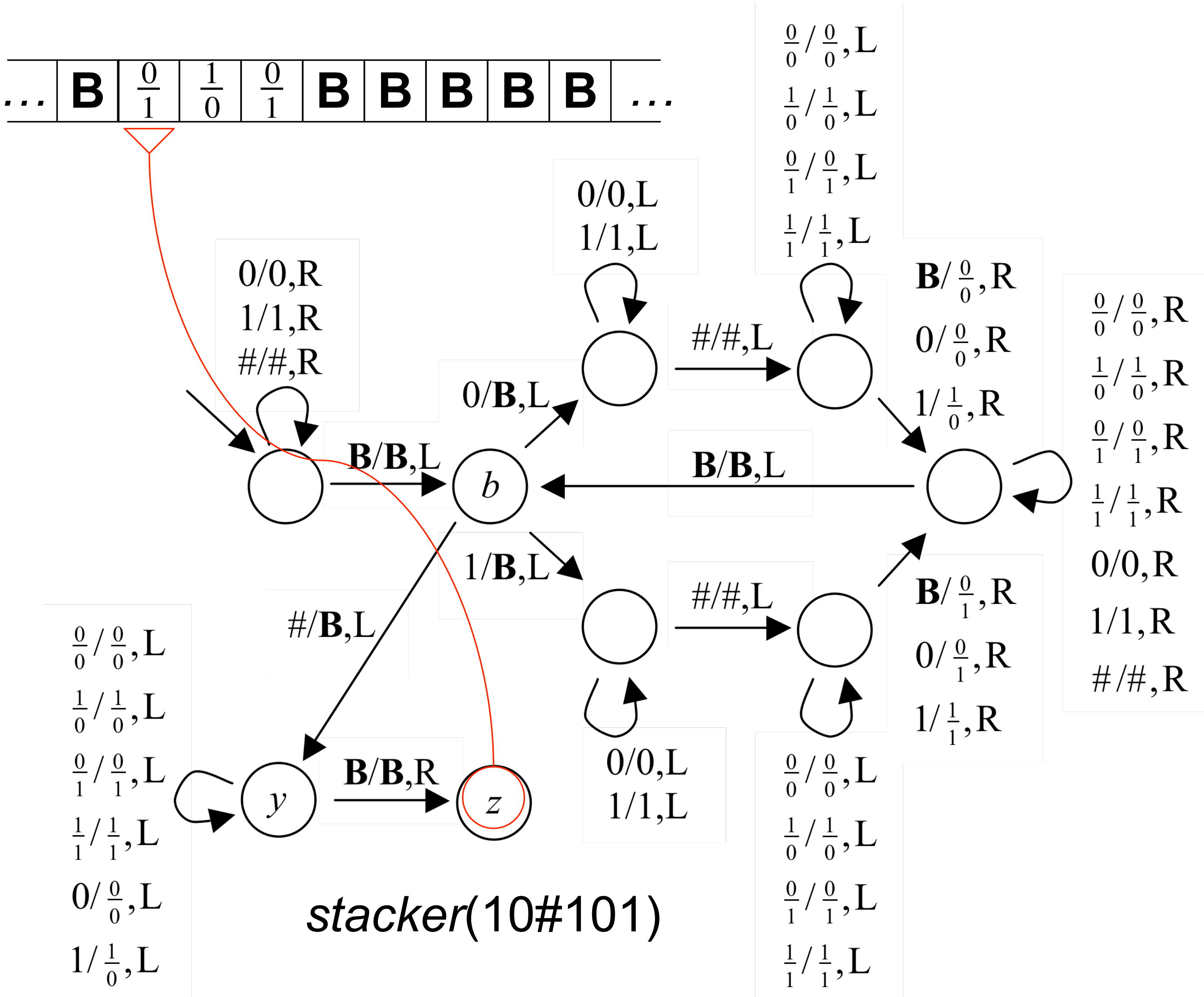
stacker(10#101)



stacker(10#101)



stacker(10#101)

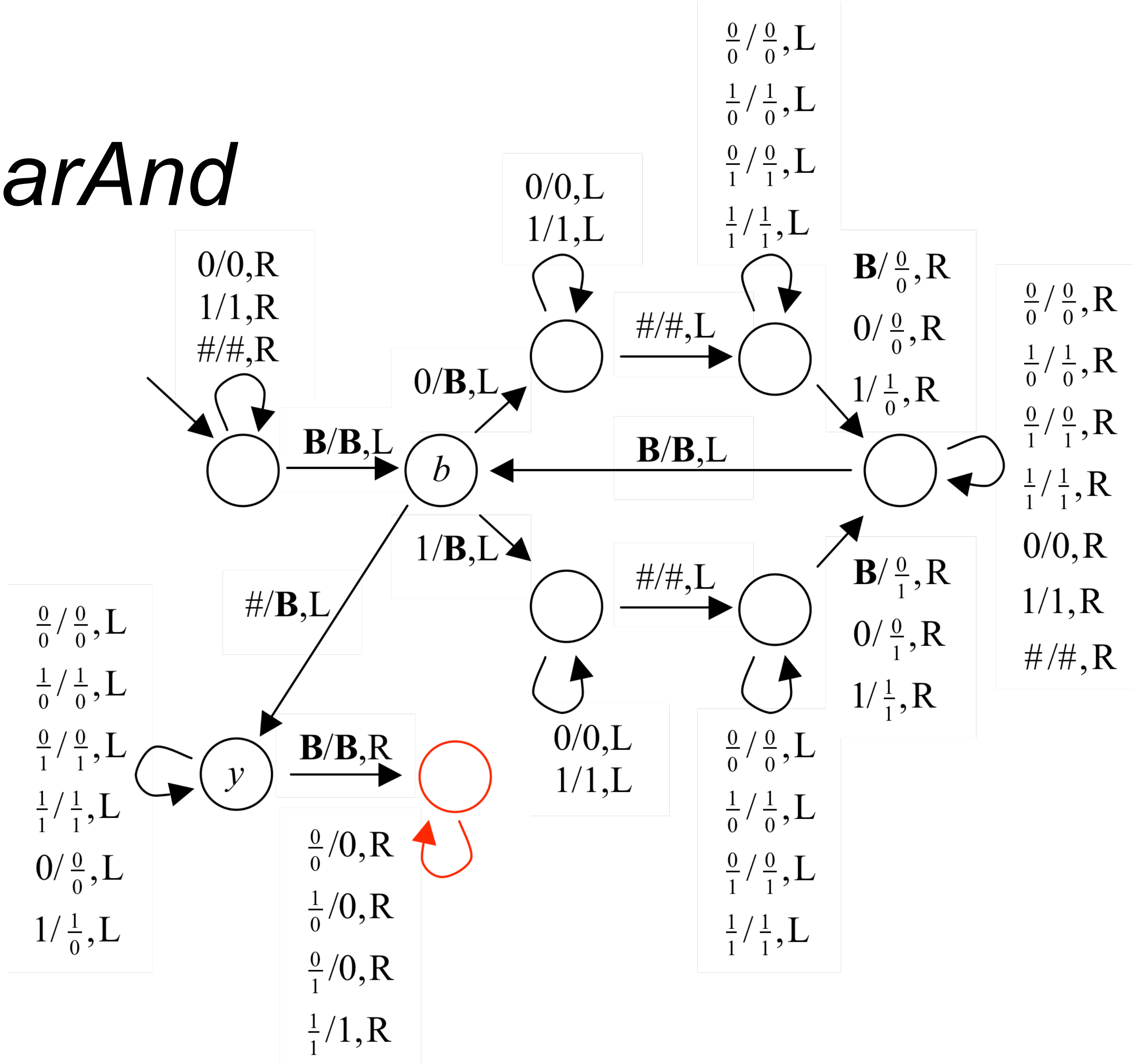


stacker(10#101)

Composition

- The *stacker* machine leaves the head at the left end of the stacked input
- That's just what *and* wants
- So we can make a machine for the function *linearAnd*(*y*) = *and*(*stacker*(*y*))
- Just use the start state of the *and* machine in place of the final state of the *stacker* machine

linearAnd



0/0,R
1/1,R
#/#,R
B/**B**,L
b
0/**B**,L
0/0,L
1/1,L
#/#,L
$\frac{0}{0}/\frac{0}{0}$,L
$\frac{1}{0}/\frac{1}{0}$,L
$\frac{0}{1}/\frac{0}{1}$,L
$\frac{1}{1}/\frac{1}{1}$,L
B/$\frac{0}{0}$,R
0/$\frac{0}{0}$,R
1/$\frac{1}{0}$,R
$\frac{0}{0}/\frac{0}{0}$,R
$\frac{1}{0}/\frac{1}{0}$,R
$\frac{0}{1}/\frac{0}{1}$,R
$\frac{1}{1}/\frac{1}{1}$,R
0/0,R
1/1,R
#/#,R
B/**B**,L
1/**B**,L
#/#,L
0/0,L
1/1,L
B/$\frac{0}{1}$,R
0/$\frac{0}{1}$,R
1/$\frac{1}{1}$,R
$\frac{0}{0}/\frac{0}{0}$,L
$\frac{1}{0}/\frac{1}{0}$,L
$\frac{0}{1}/\frac{0}{1}$,L
$\frac{1}{1}/\frac{1}{1}$,L
#/**B**,L
$\frac{0}{0}/\frac{0}{0}$,L
$\frac{1}{0}/\frac{1}{0}$,L
$\frac{0}{1}/\frac{0}{1}$,L
$\frac{1}{1}/\frac{1}{1}$,L
0/$\frac{0}{0}$,L
1/$\frac{1}{0}$,L
y
B/**B**,R
$\frac{0}{0}$/0,R
$\frac{1}{0}$/0,R
$\frac{0}{1}$/0,R
$\frac{1}{1}$/1,R

Outline

- 17.1 Turing-Computable Functions
- 17.2 TM Composition
- 17.3 TM Arithmetic
- 17.4 TM Random Access
- 17.5 Functions And Languages
- 17.6 The Church-Turing Thesis
- 17.7 TM and Java Are Interconvertible
- 17.8 Infinite Models, Finite Machines

Functions And Languages

- For every language L we can define a corresponding function, such as $f(x) = 1$ if $x \in L$, 0 if $x \notin L$
- For every function f we can define a corresponding language, such as $L = \{x\#y \mid y = f(x)\}$
- L is recursive if and only if f is Turing computable
- Function implementation and language definition are just two different perspectives on the same computational problem

The Power Of TMs

- Our examples have shown that TMs can
  - Implement boolean logic
  - Get the effect of subroutine calls by composition
  - Perform binary arithmetic
  - Index memory using binary addresses
- All the building blocks of modern computer systems
- Evidence for the extraordinary power of TMs
- *TMs can do anything that can be done with a high-level programming language on a modern computer*
  - TM are *models* of computation ignoring some of the details of actual machines.

Outline

Computational Procedures

- Many familiar old parts of mathematics are concerned with computational procedures
  - Find the GCD of two natural numbers
  - Find the inverse of a decimal number
  - Find the roots of a second-degree polynomial
  - Construct a regular pentagon with a compass and straightedge

Big Questions

- In the early 1900s: we can’t find effective computational procedures for some fundamental mathematical problems
  - Given an assertion in first-order logic, decide whether it is true or false
  - Given a polynomial equation, find integer solutions if any
- 1920s and 1930s: what exactly is an *effective computational procedure,* anyway?

Formalisms For Computation

- A number of different formalisms were developed to try to capture *effective computational procedure*:
  - Alan Turing: Turing machine
  - Alonzo Church, Stephen Kleene: λ-calculus
  - Emil Post: Post systems
  - Kurt Gödel: μ-recursive functions
  - Moses Schönfinkel, Haskell Curry: combinator logic
- They operate on different kinds of data: strings (for Turing machines), natural numbers (for μ-recursive functions), etc.
- They all started out in different directions, but…

Interconvertibility

- …they all ended up in the same place!
- With suitable data conversions, all those formalisms are interconvertible:
  - Any Turing machine can be simulated by a Post system and vice versa
  - Any Post system can be simulated by a λ-calculus term and vice versa
  - And so on
- Since they all can simulate each other they are all *computationally equivalent*.

Turing Equivalence

- Any formalism for computation that is interconvertible with Turing machines is Turing equivalent
- All Turing-equivalent formalisms have the same computational power as Turing machines
- They also have the same lurking possibility of infinite computation
- In 1936, Church and Turing suggested their (Turing-equivalent) formalisms had captured the elusive idea of an effective computational procedure

Church-Turing Thesis

- In effect, they suggested the following definition:

“Computability” means Turing computability

or

Anything an Algorithm can do a TM can do.

- (Recall that Turing computability requires a total TM: nonterminating procedures are not considered effective!)
- This is known as Church’s Thesis, or the Church-Turing Thesis
- A thesis, not a theorem; a definition like this is not subject to proof or disproof
- Now generally accepted: Turing computability is the border of our happy realm

Church-Turing Thesis

- Why is it generally accepted?
  - In the 1930s, it kept turning up: many researchers walking down different paths of mathematical inquiry arrived at the same idea of effective computation
  - Today, we have the additional evidence of modern programming languages and physical computer systems, which are also Turing equivalent
  - All are interconvertible
- "Computable by a Java program that always halts"
  = "Computable by a total Turing machine"
  = Computable

Related Terminology

- A (*Turing*-) *computable function*
  - Addition
- Language $L(M)$ for a total TM: a *recursive language*
  - $\{a^n b^n c^n\}$
- A property of strings that can be recognized by some total TM: a *decidable property*
  - Primality
- An *algorithm*: like *effective computational procedure*, but not just for functions
  - Fault-tolerant distributed algorithms, probabilistic algorithms, interactive algorithms….

Outline

- 17.1 Turing-Computable Functions
- 17.2 TM Composition
- 17.3 TM Arithmetic
- 17.4 TM Random Access
- 17.5 Functions And Languages
- 17.6 The Church-Turing Thesis
- 17.7 TM and Java Are Interconvertible
- 17.8 Infinite Models, Finite Machines

TM vs. Physical Computer

- A physical computer is limited by finite memory
- There are always program executions that demand more memory than you have (and more time)
- TMs have no such limitations
- TMs can do a better job of implementing high-level languages than an ordinary computer
- In the abstract, high-level languages like Java are Turing equivalent
- In physical implementations they are not: they are subject to limitations that TMs don’t share

TM vs. DFA

- Any physical computer has a finite memory
- Therefore, it has finitely many possible states
- It’s really more like a giant DFA than a TM: it has an enormous, but still finite, set of states
- The unbounded model of TM computation is patently unrealistic
- So why does it feel so natural?

Looks Unbounded To Us

- One explanation: the number of states is so large that it appears unbounded to humans
- For example, word processors:
  - Represent only finitely many different documents
  - But writers do not think of writing as the act of selecting one of finitely many representable texts
- Similarly, general programs:
  - Put the machine in finitely many different possible states
  - But programmers do not think of programs as DFAs

Mathematical Idealizations

- Mathematical idealizations can be useful:
  - Geometry: no perfect points, lines, or circles in the world, but still useful concepts
  - Calculus: no infinites or infinitesimals in the world, but still useful concepts
- Similarly, TMs:
  - No unbounded computations in the world, but still useful
  - In particular, the TM concept of *uncomputability* is useful
  - If a computation is provably impossible for an idealized, unbounded machine, it is certainly impossible for a limited physical computer as well
  - More about uncomputability in the next chapter