Algorithmics

Lecture 11

Dr. Oana Andrei

School of Computing Science
University of Glasgow

oana.andrei@glasgow.ac.uk

Section 5 – Computability

Introduction

Models of computation

- finite-state automata – regular languages and regular expressions
- pushdown automata
- Turing machines
- Counter machines
- Church–Turing thesis

Computability recap

What is a computer?

input x → Black box → output f(x)

What can the black box do?

– it computes a function that maps an input to an output

Computability concerned with which functions can be computed

– a formal way of answering ‘which problems can be solved by a computer?’
– or alternatively ‘which problems cannot be solved by a computer?’

To answer such questions we require a formal definition

– i.e. a definition of what a computer is
– alternatively of what an algorithm is if we view a computer as a device that can execute an algorithm

Deterministic finite-state automata recap

Simple machines with limited memory which recognise input on a read-only tape

A DFA consists of

- a finite input alphabet Σ – i.e. symbols on the read-only tape
- a finite set of states Q – i.e. memory
- a initial/start state $q_0 \in Q$ and set of accepting states $F \subseteq Q$
- control/program or transition relation $T \subseteq (Q \times \Sigma) \times Q$
    - $((q,a),q') \in T$ means if in state q and read a, then move to state q'

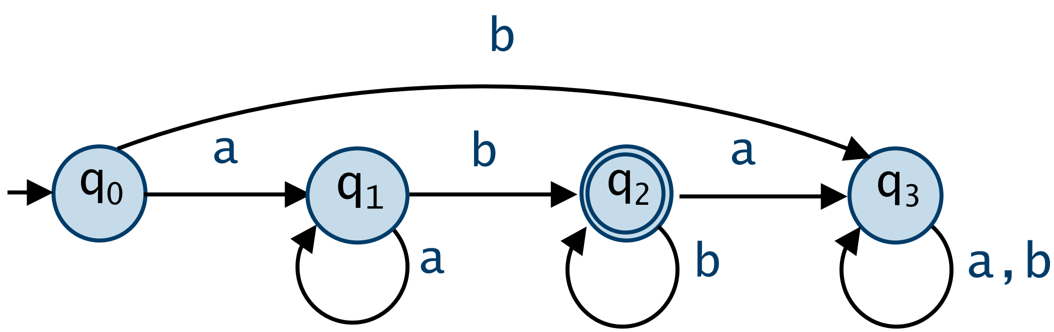
Deterministic finite-state automata recap

Simple machines with limited memory which recognise input on a read-only tape

add input tape (finite sequence of elements/actions from the alphabet)

A DFA consists of

- a finite input alphabet Σ
- a finite set of states Q
- a initial/start state $q_0 \in Q$ and set of accepting states $F \subseteq Q$
- control/program or transition relation $T \subseteq (Q \times \Sigma) \times Q$
  - $((q,a),q') \in T$ means if in state q and read a, then move to state q'



control/program

((q0,a), q1)
((q0,b), q3)
((q1,a), q1)
((q1,b), q2)
((q2,a), q3)
((q2,b), q2)
((q3,a), q3)
((q3,b), q3)

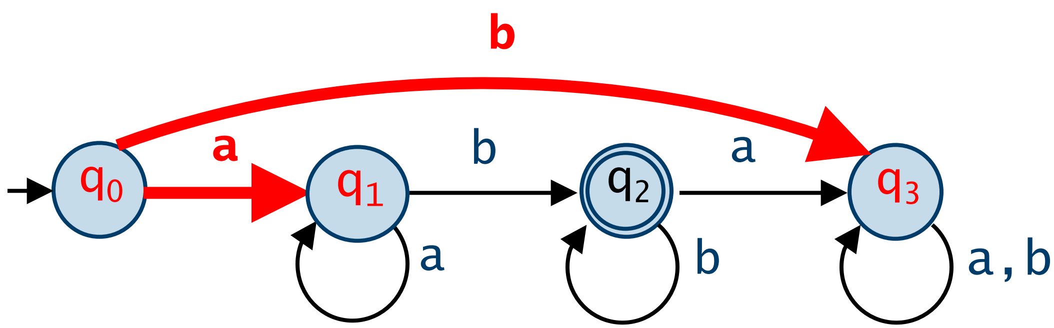
Deterministic finite-state automata recap

Simple machines with limited memory which recognise input on a read-only tape

add input tape (finite sequence of elements/actions from the alphabet)

A DFA consists of

- a finite input alphabet Σ
- a finite set of states Q
- a initial/start state $q_0 \in Q$ and set of accepting states $F \subseteq Q$
- control/program or transition relation $T \subseteq (Q \times \Sigma) \times Q$
  - $((q,a),q') \in T$ means if in state q and read a, then move to state q'



control/program

```
((q0,a), q1)
((q0,b), q3)
((q1,a), q1)
((q1,b), q2)
((q2,a), q3)
((q2,b), q2)
((q3,a), q3)
((q3,b), q3)
```

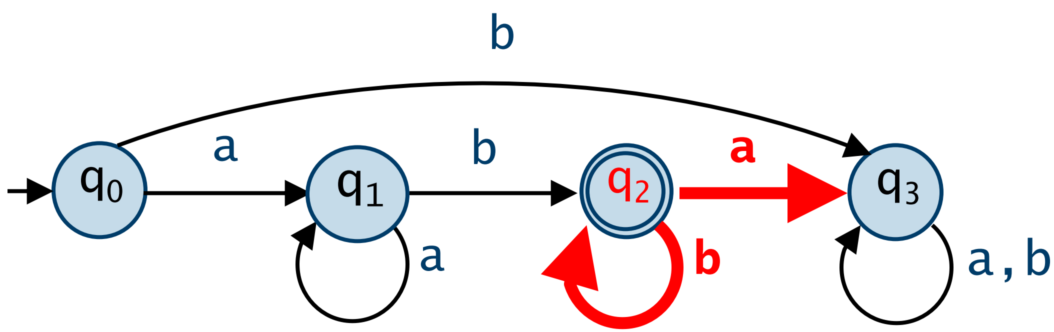
Deterministic finite-state automata recap

Simple machines with limited memory which recognise input on a read-only tape

A DFA consists of

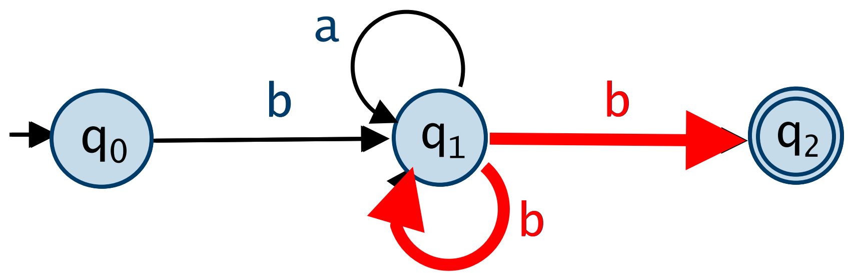
add input tape (finite sequence of elements/actions from the alphabet)

- a finite input alphabet Σ
- a finite set of states Q
- a initial/start state $q_0 \in Q$ and set of accepting states $F \subseteq Q$
- control/program or transition relation $T \subseteq (Q \times \Sigma) \times Q$
  - $((q,a),q') \in T$ means if in state q and read a, then move to state q'



control/program
((q0,a), q1)
((q0,b), q3)
((q1,a), q1)
((q1,b), q2)
((q2,a), q3)
((q2,b), q2)
((q3,a), q3)
((q3,b), q3)

Another example



Recognises strings that start and end with b

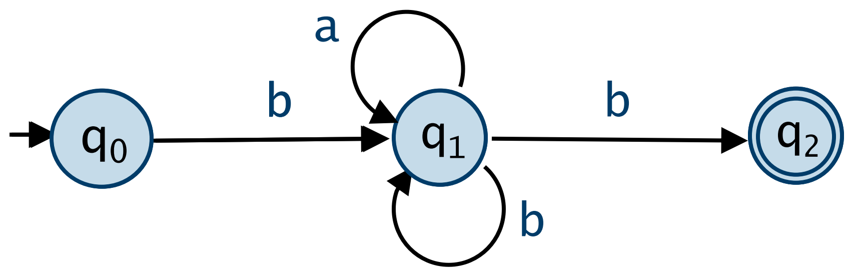
However this is not a DFA, but a non-deterministic finite-state automaton (NFA)

- in state q_1 under b can move to q_1 or q_2

Recognition for NFA is similar to non-deterministic algorithms "solving" a decision problem

- only require there exists a 'run' that ends in an accepting state
- i.e. under one possible resolution of the nondeterministic choices the input is accepted

Another example



Recognises strings that start and end with b

However this is not a DFA, but a non-deterministic finite-state automaton (NFA)

- in state q_1 under b can move to q_1 or q_2

But any NFA can be converted into a DFA

Therefore non-determinism does not expand the class of languages that can be recognised by finite state automata

- being able to guess does not give us any extra power

DFAs and languages

A (formal) language with alphabet Σ is a set of words over Σ

- a word over Σ is a sequence of symbols taken from Σ
- a word w over Σ is recognised or accepted by a finite-state automaton with alphabet Σ if the the automaton reaches an accepting state when started in the initial state on the tape that contains the word w

The languages that can be recognised by finite-state automata are called the regular languages

Regular languages and regular expressions

The languages that can be recognised by finite-state automata are called the regular languages

A regular language (over an alphabet Σ) can be specified by a regular expression over Σ

- ε (the empty string) is a regular expression
- σ is a regular expression (for any single character $\sigma \in \Sigma$)

if R and S are regular expressions, then so are

- RS which denotes concatenation
- R | S which denotes choice between R or S
- R* which denotes 0 or more copies of R (sometimes called closure)
- (R) bracketing is sometimes needed to override precedence between operators

Regular expressions

Order of precedence (highest first)

- closure (*) then concatenation then choice (|)
- with brackets used to override this order

Example: suppose Σ = {a,b,c,d}

- R = (ac|a*b)d means ((ac) | ((a*) b)) d
- corresponding language L_R is

  {acd, bd, abd, aabd, aaabd, aaaabd, … }

Regular expressions

Order of precedence (highest first)

- closure (*) then concatenation then choice (|)
- with brackets used to override this order

Example: suppose Σ = {a,b,c,d}

- R = (ac|a*b)d means ((ac) | ((a*) b)) d
- corresponding language L_R is

  {acd, bd, abd, aabd, aaabd, aaaabd, … }

Regular expressions

Order of precedence (highest first)

- closure (*) then concatenation then choice (|)
- with brackets used to override this order

Example: suppose **Σ = {a,b,c,d}**

- R = (ac|a*b)d means ((ac) | ((a*) b)) d
- corresponding language L_R is

  {acd, bd, abd, aabd, aaabd, aaaabd, … }

Regular expressions

Order of precedence (highest first)

- closure (*) then concatenation then choice (|)
- with brackets used to override this order

Example: suppose Σ = {a,b,c,d}

- R = (ac|a*b)d means ((ac) | ((a*) b)) d
- corresponding language L_R is

  {acd, bd, abd, aabd, aaabd, aaaabd, … }

Regular expressions

Order of precedence (highest first)

- closure (*) then concatenation then choice (|)
- with brackets used to override this order

Example: suppose **Σ = {a,b,c,d}**

- `R = (ac|a*b)d` means `( ( ac ) | ( (a*) b ) ) d`
- corresponding language L_R is

  `{acd, bd, abd, aabd, aaabd, aaaabd, … }`

Additional operations

- complement ¬x
  - equivalent to the 'or' of all characters in Σ except x
- any single character ?
  - equivalent to the 'or' of all characters

Regular expressions

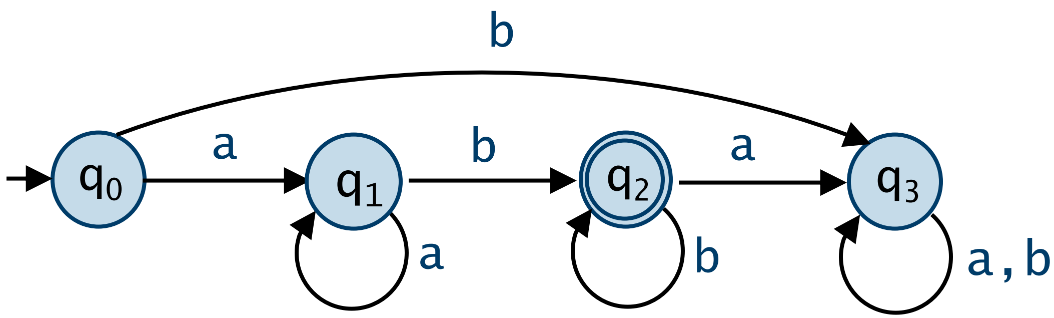
Regular expressions (regex or regexp) have an important role in CS applications especially those involving text

- i.e. searching for strings that satisfy certain patterns
- search engines, text editors, word processors
- e.g. text processing utilities in Unix such as `awk`, `grep`, `sed`
  - the IEEE POSIX standard for basic regular expression syntax
- built into the syntax of `Perl`
- supported by standard libraries of programming languages
  - e.g. `Python`, `Java`, `JavaScript`, C++, C#, etc.

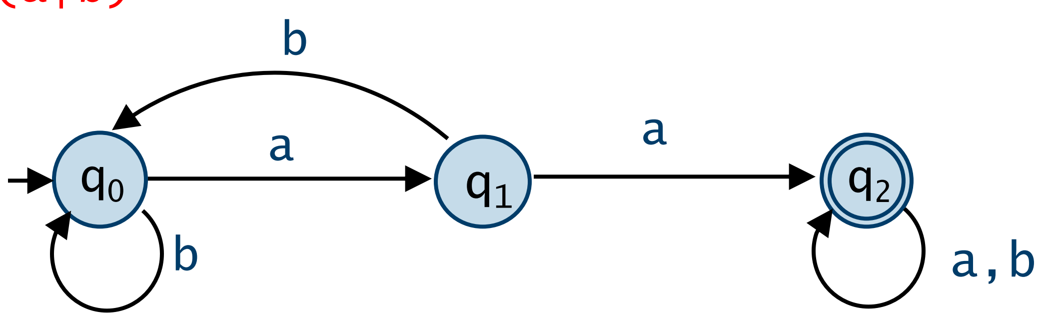
Regular expressions – Examples

The examples from previous lecture over **{a,b}**

1) the language comprising one or more a's followed by one or more b’s
   - aa*bb*



2) the language of strings containing two consecutive a’s
   - (a|b)*aa(a|b)*

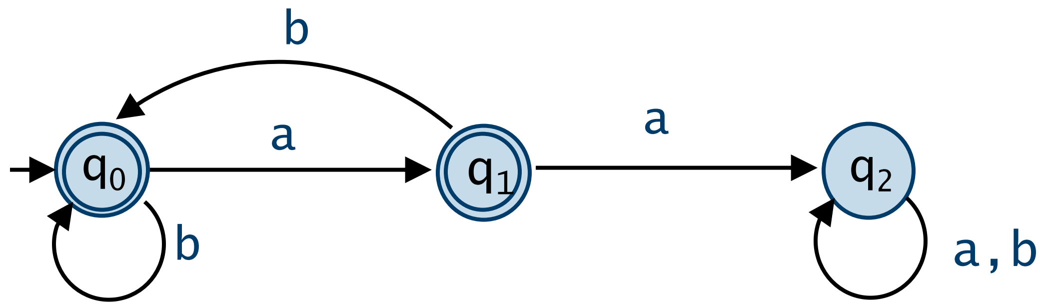


Regular expressions – Examples

The examples from previous lecture over {a,b}

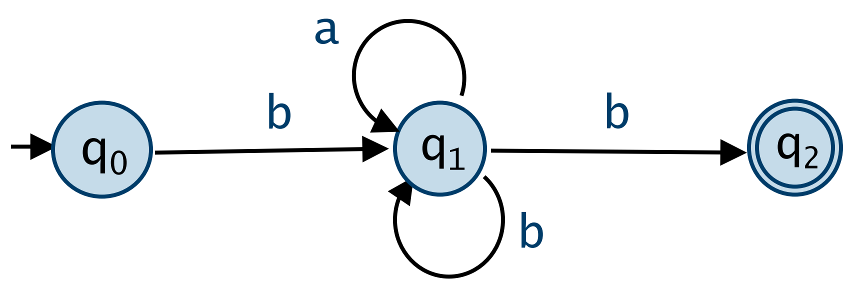
3) the language of strings that do not contain two consecutive a's

– b*(abb*)*(ε|a)



4) the language of strings that start and end with b

– b(a|b)*b



Regular expressions – Closure

To clarify what **R*** means

– corresponds to 0 or more copies of the regular expression R

Let **L(R)** be the language corresponding to the regular expression **R**

– then concatenation is given by $L(RS) = \{ rs \mid r \in L(R) \text{ and } s \in L(S) \}$

and $L(R^*) = L(R^0) \cup L(R^1) \cup L(R^2)\ldots$

where $L(R^0) = \{\varepsilon\}$ and $L(R^{i+1}) = L(RR^i)$

– for example: $ab,\ aab \in L(a^*b^*)$, and hence $abaab \in L((a^*b^*)^*)$

Regular expressions – Closure

To clarify what **R*** means

- corresponds to 0 or more copies of the regular expression R

Let **L(R)** be the language corresponding to the regular expression **R**

- then concatenation is given by L(RS) = { rs | r∈L(R) and s∈L(S) }
  and $L(R^*) = L(R^0) \cup L(R^1) \cup L(R^2)\ldots$
  where $L(R^0) = \{\varepsilon\}$ and $L(R^{i+1}) = L(RR^i)$
- note (a*b*)* is in fact equivalent to (a|b)*

L(R*) does not mean **{ r* | r ∈ L(R) }**

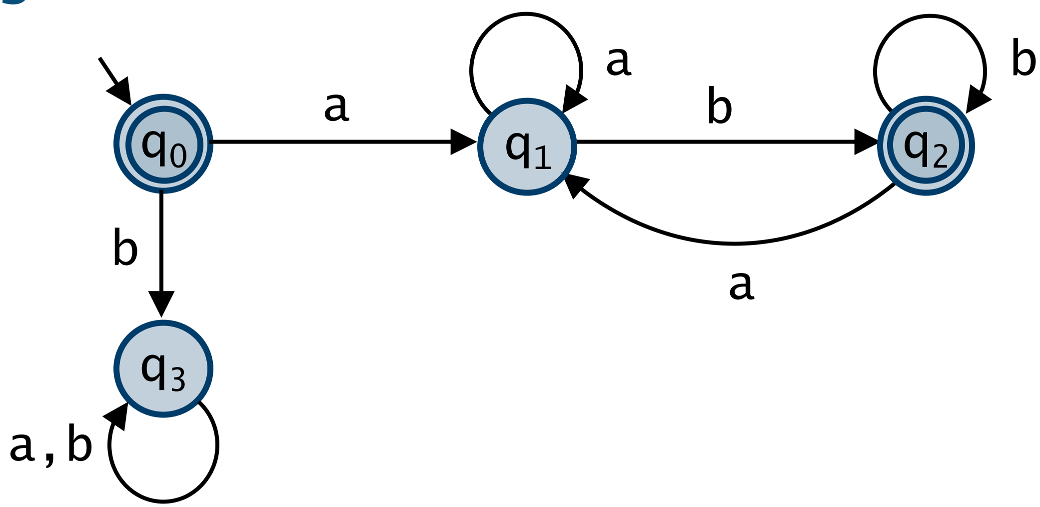
- not zero or more copies of one element from L(R)
- but zero or more copies where each copy can be anything from L(R)
- for certain regular expressions cannot be recognized by any DFA
- essentially for such a language would need a memory to remember which string in r∈L(R) is repeated and there might be an unbounded number

Regular expressions – Example

Consider the language **(aa*bb*)***

- i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

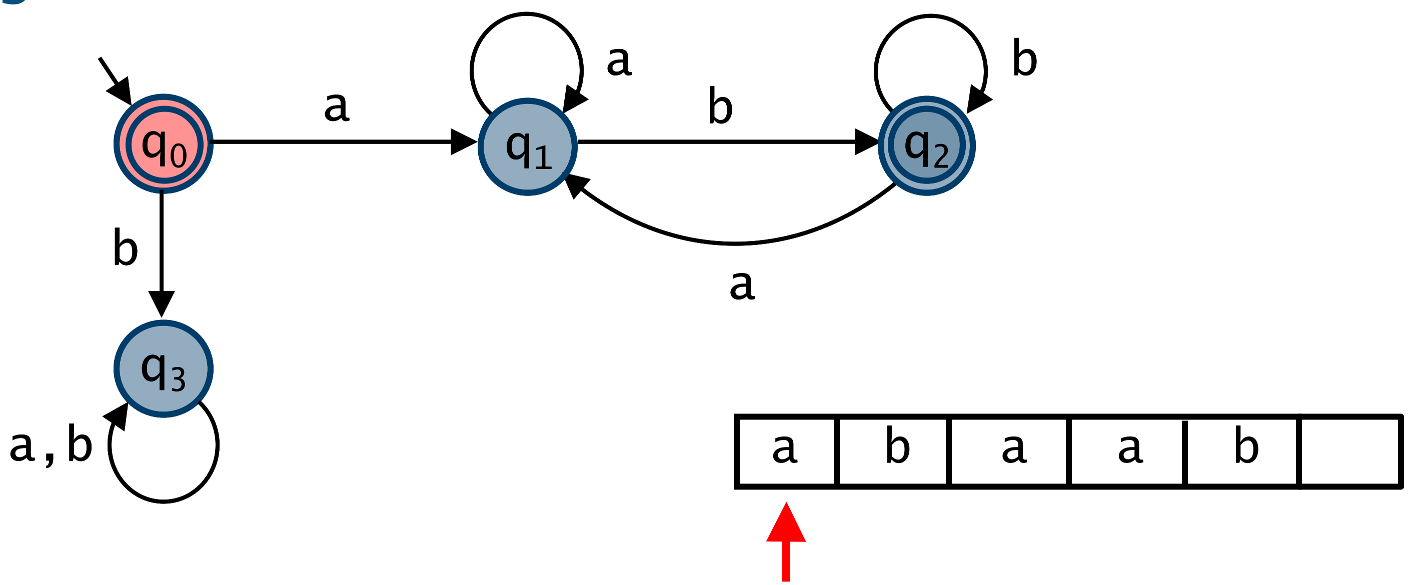


Regular expressions – Example

Consider the language **(aa*bb*)***

– i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

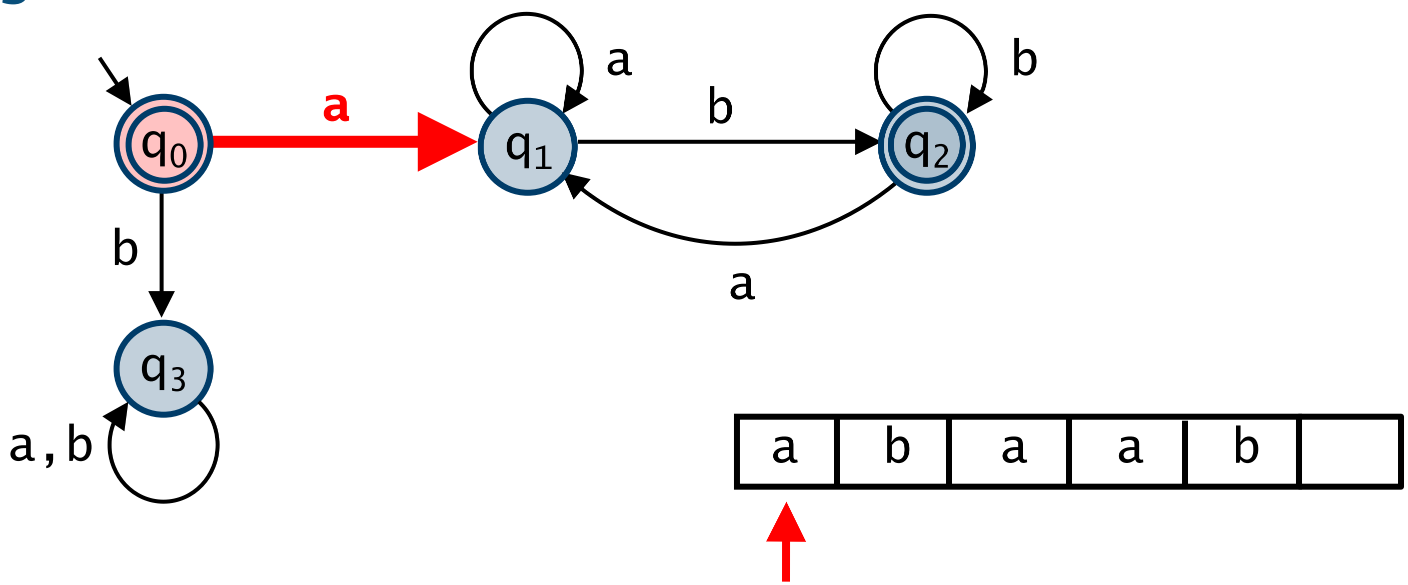


Regular expressions – Example

Consider the language (aa*bb*)*

- i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

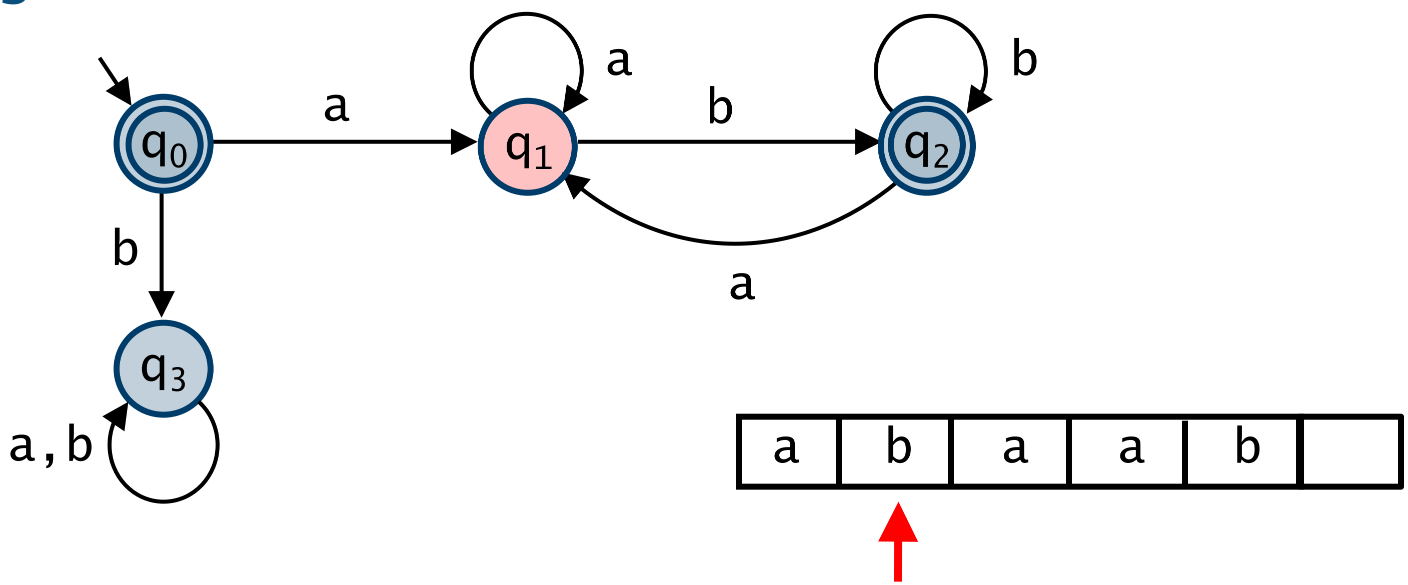


Regular expressions – Example

Consider the language **(aa*bb*)***

- i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

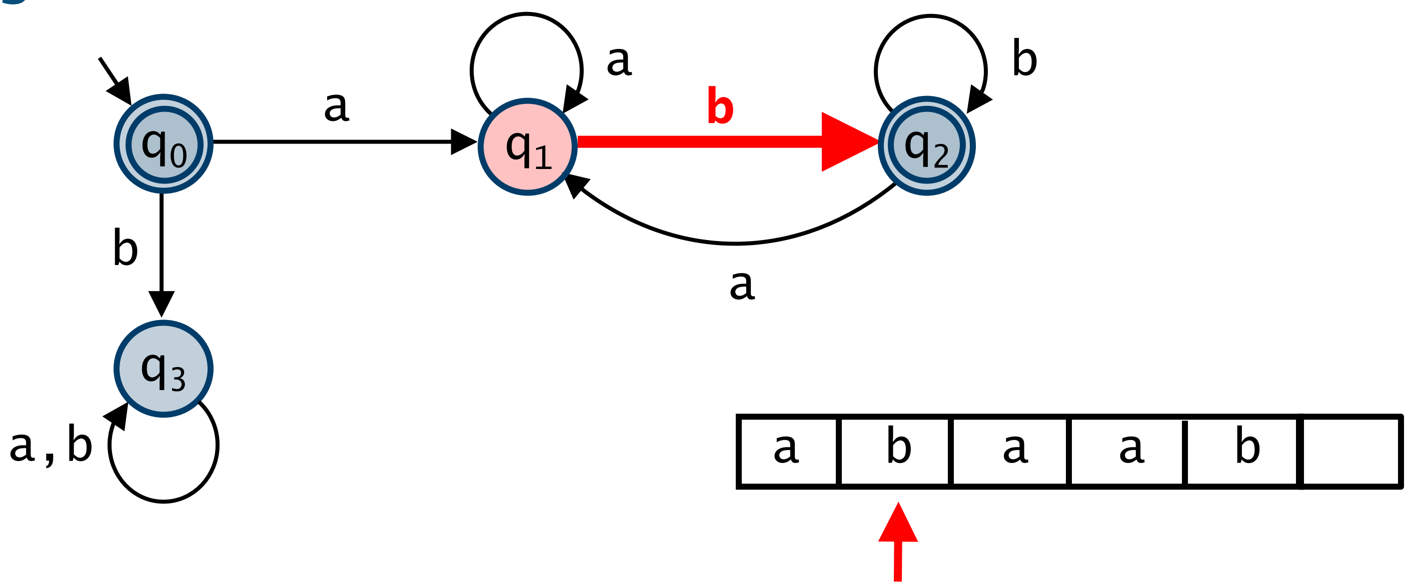


Regular expressions – Example

Consider the language (aa*bb*)*

– i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

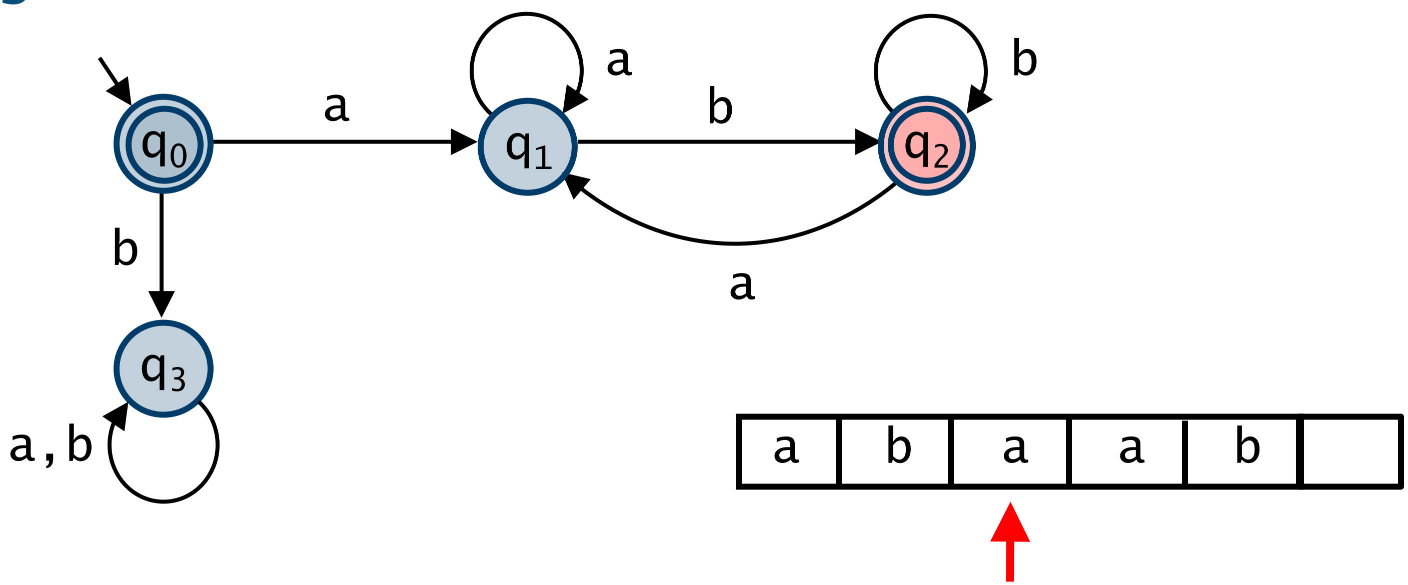


Regular expressions – Example

Consider the language (aa*bb*)*

- i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

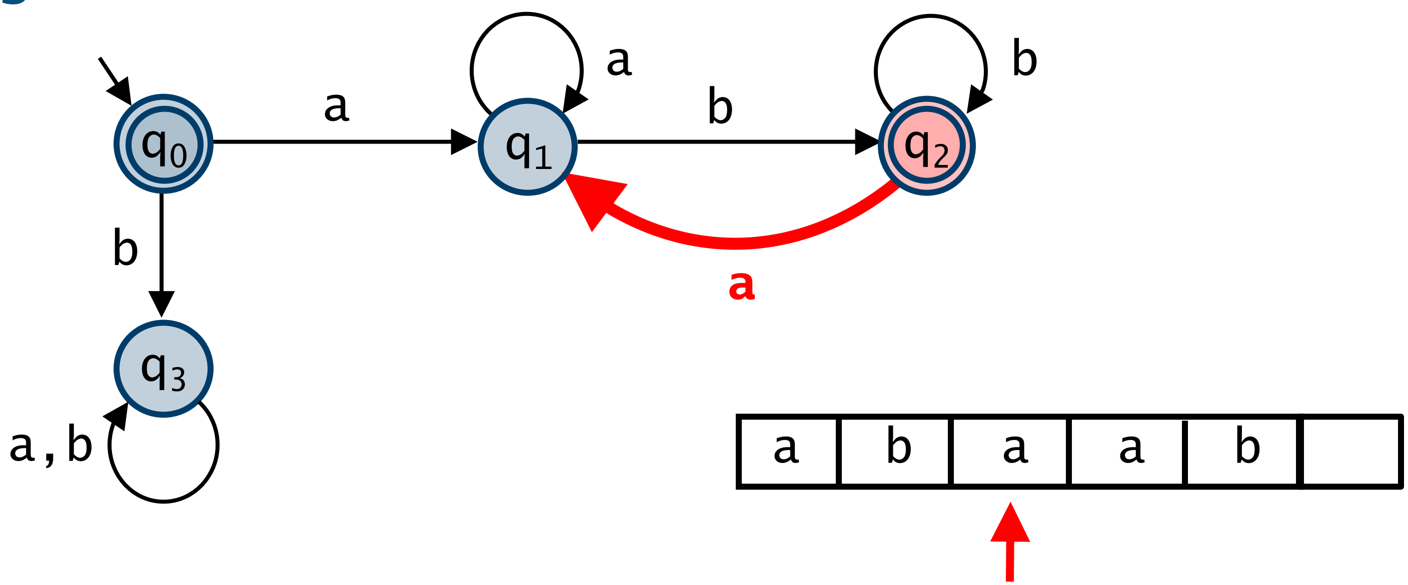


Regular expressions – Example

Consider the language (aa*bb*)*

- i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

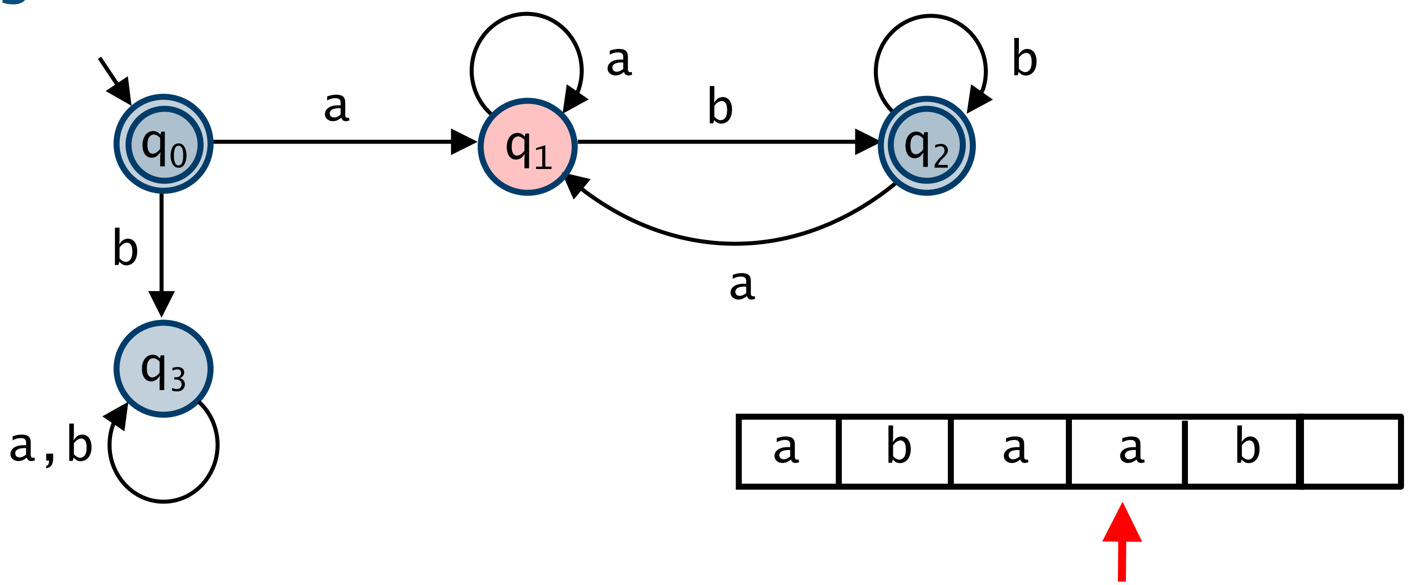


Regular expressions – Example

Consider the language **(aa*bb*)***

- i.e. zero or more sequences which consist of a non-zero number of a’s followed by a non-zero number of b’s

Corresponding DFA:

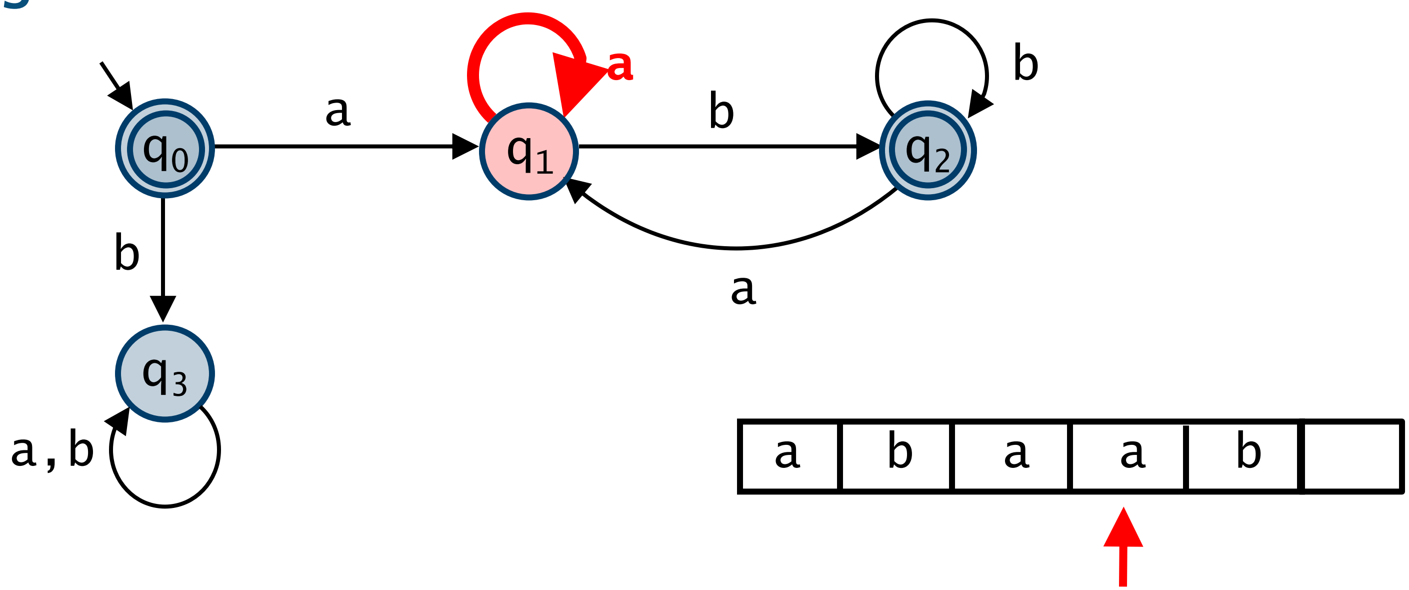


Regular expressions – Example

Consider the language (aa*bb*)*

- i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

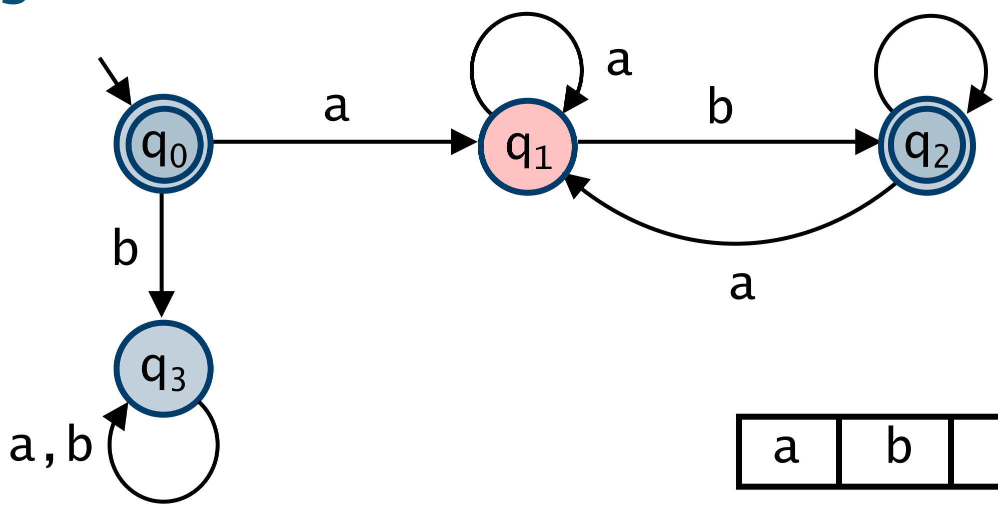


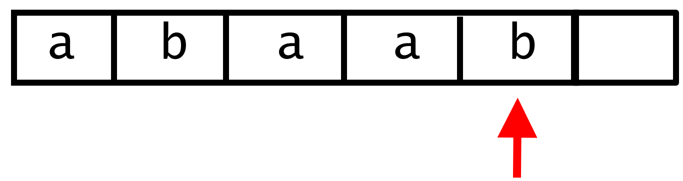
Regular expressions – Example

Consider the language **(aa*bb*)***

– i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:



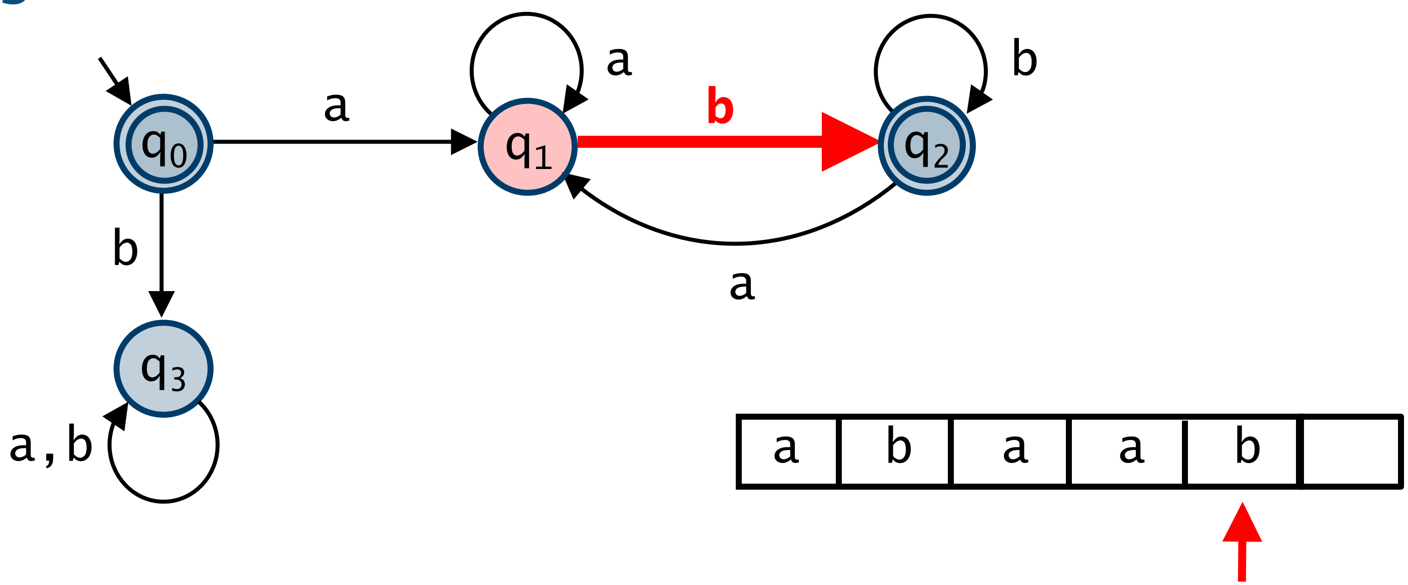


Regular expressions – Example

Consider the language (aa*bb*)*

- i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:

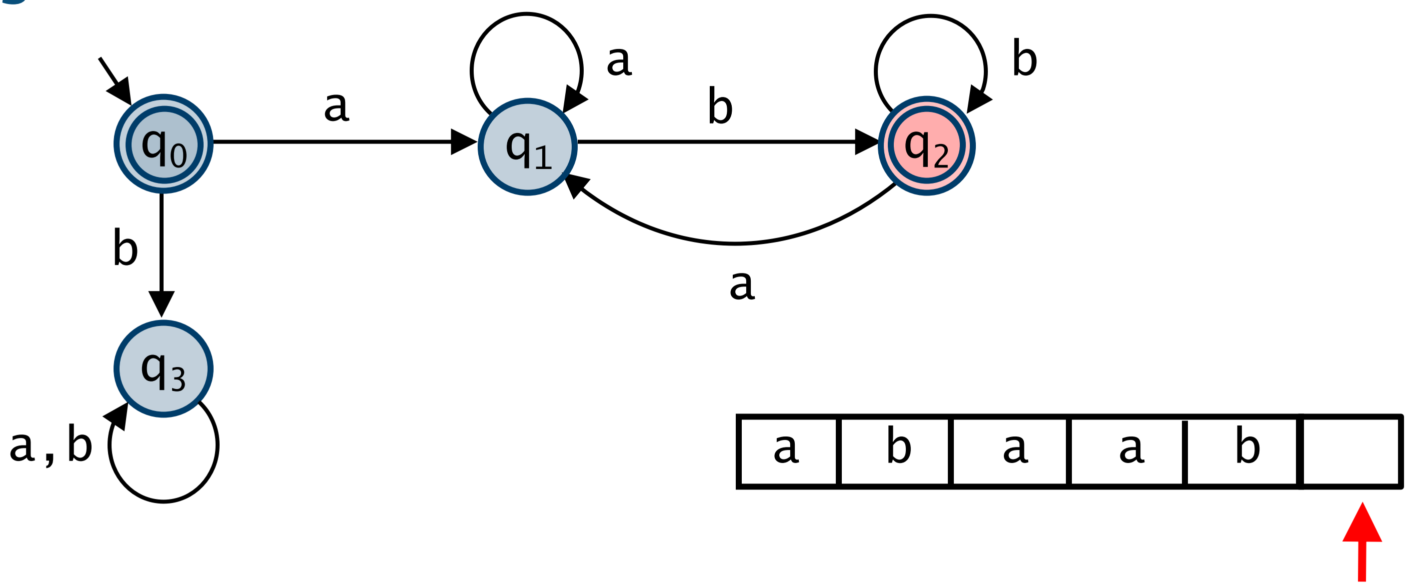


Regular expressions – Example

Consider the language (aa*bb*)*

- i.e. zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's

Corresponding DFA:



Regular expressions – Example

A DFA cannot recognise $\{\ r^* \mid r \in L(aa^*bb^*)\ \}$

- i.e. $\{\ (a^m b^n)^* \mid m > 0 \text{ and } n > 0\ \}$
- the problem is the DFA would need to remember the m and n to check that a string is in the langauge
- but there are infinitely many values for m and n
- hence the DFA would need infinitely many states
- and we only have a finite number (DFA = deterministic **finite** automaton)

Similarly a DFA cannot recognise $\{\ a^n b^n \mid n > 0\ \}$

- i.e. a number of a's followed by the same number of b's

Languages that are recognised by DFAs are called regular languages so, for example $\{\ a^n b^n \mid n > 0\ \}$ is not regular

Regular expressions – Example

How can we recognising strings of the form $a^n b^n$?

- i.e. a number of a's followed by the same number of b's

It turns out that there is no DFA that can recognise this language

- it cannot be done without some form of memory, e.g. a stack

Idea: as you read a' s, push them onto a stack, then pop the stack as you read b' s, i.e. the stack works like a counter

So there are some functions (languages) that we would regard as computable that cannot be computed by a finite-state automaton

- DFAs are not an adequate model of a general-purpose computer i.e. our 'black box'

Pushdown automata extend finite-state automata with a stack

Section 5 – Computability

Introduction

The halting problem

Models of computation

- finite-state automata
- pushdown automata
- Turing machines
- Counter machines
- Church–Turing thesis

Why extending DFAs

A deterministic finite-state automaton (DFA) cannot recognise the language $\{ a^n b^n \mid n \geq 0 \}$

- i.e. a number of a's followed by the same number of b's
- the problem is the DFA would need to remember n to check that a string is in the language
- but there are infinitely many values for n
- hence the DFA would need infinitely many states and we only have a finite number

Why extending DFAs

How can we recognising strings of the form $a^n b^n$?

- i.e. a number of a's followed by the same number of b's
- it cannot be done without some form of memory, e.g. a stack

Idea: as you read a' s, push symbols onto a stack, then pop the stack as you read b' s, i.e. the stack works like a counter

So there are some functions (languages) that we would regard as computable that cannot be computed by a finite-state automaton

- finite-state automata are not an adequate model of a general-purpose computer i.e. our 'black box'

Pushdown automata extend finite-state automata with a stack

Pushdown automata

A **pushdown automaton (PDA)** consists of:

- a finite input alphabet Σ, a finite set of stack symbols G (same or different)
- a finite set of states Q including start state and set of accepting states
- control or transition relation $T \subseteq (Q \times \Sigma \cup \{\varepsilon\} \times G \cup \{\varepsilon\}) \times (Q \times G \cup \{\varepsilon\})$

ε – empty string

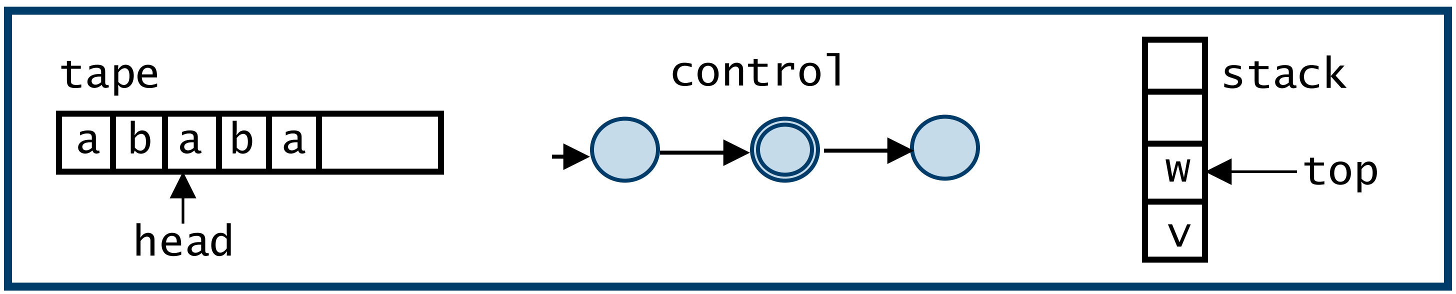
current state

tape symbol or ε

old stack symbol or ε

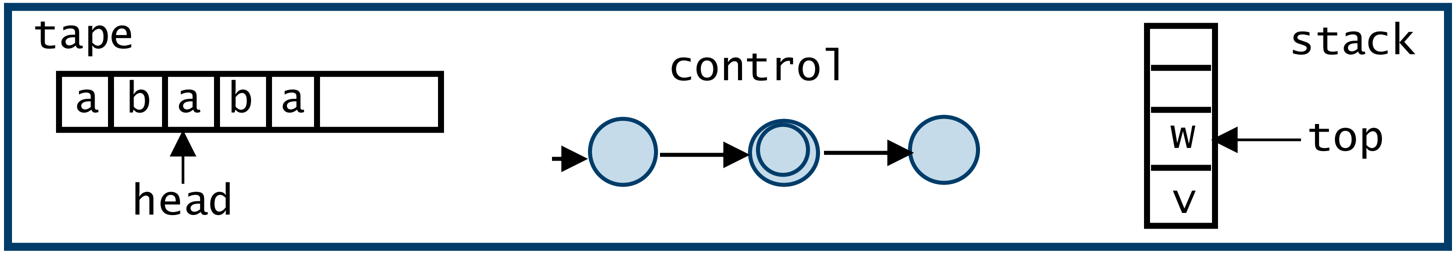
new state

new stack symbol or ε



Pushdown automata

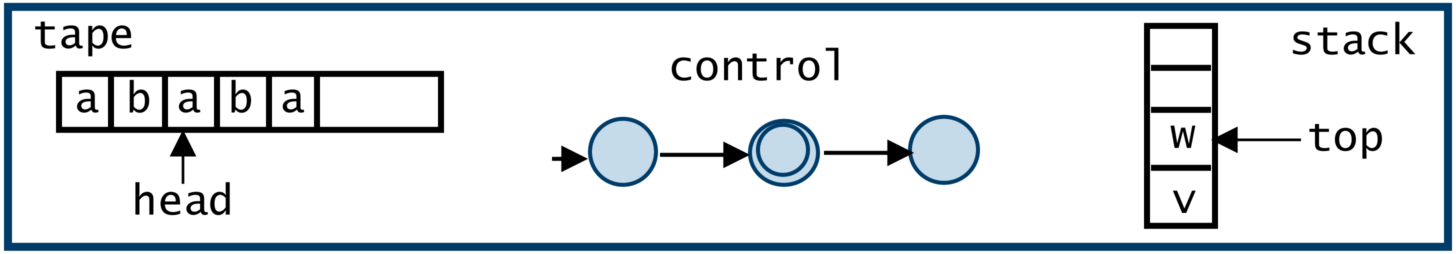
Transition relation $T \subseteq (Q \times \Sigma\cup\{\varepsilon\} \times G\cup\{\varepsilon\}) \times (Q \times G\cup\{\varepsilon\})$



Informally, the transition $(q_1,a,w) \rightarrow (q_2,v)$ means that

Pushdown automata

Transition relation $\mathbf{T \subseteq (Q \times \Sigma\cup\{\varepsilon\} \times G\cup\{\varepsilon\}) \times (Q \times G\cup\{\varepsilon\})}$

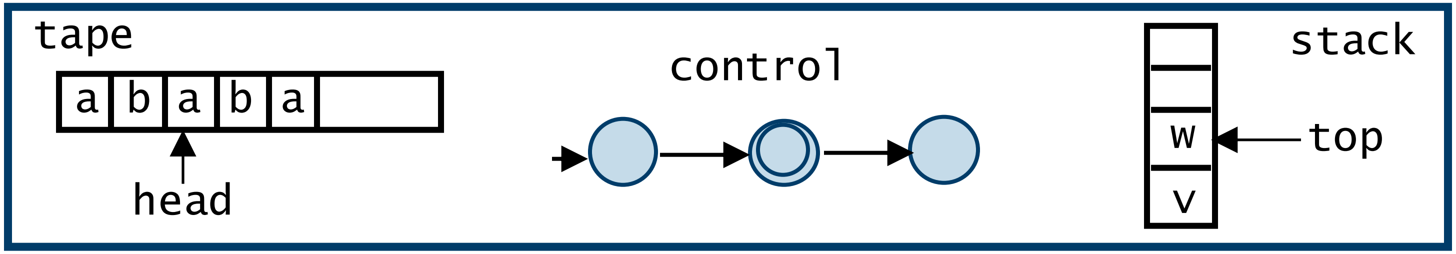


Informally, the transition $(q_1,a,w) \rightarrow (q_2,v)$ means that

– if we are in state q_1

Pushdown automata

Transition relation $T \subseteq (Q \times \Sigma\cup\{\varepsilon\} \times G\cup\{\varepsilon\}) \times (Q \times G\cup\{\varepsilon\})$

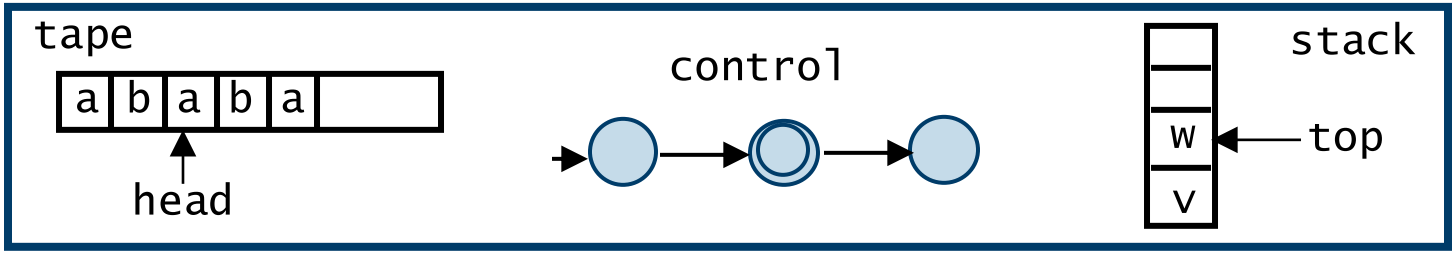


Informally, the transition $(q_1, a, w) \rightarrow (q_2, v)$ means that

- if we are in state q_1
- if $a \neq \varepsilon$, then the symbol a is at the head of the tape

Pushdown automata

Transition relation $T \subseteq (Q \times \Sigma\cup\{\varepsilon\} \times G\cup\{\varepsilon\}) \times (Q \times G\cup\{\varepsilon\})$

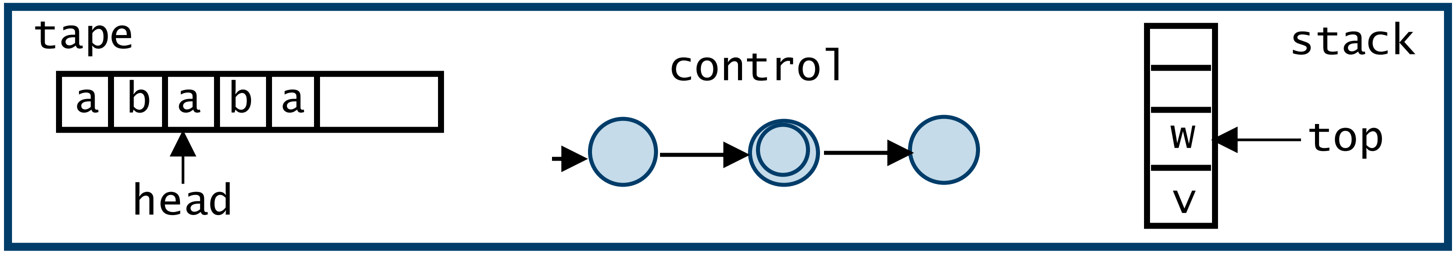


Informally, the transition $(q_1, a, w) \rightarrow (q_2, v)$ means that

- if we are in state q_1
- if $a \neq \varepsilon$, then the symbol a is at the head of the tape
- if $w \neq \varepsilon$, then the symbol w is is on top of the stack

Pushdown automata

Transition relation $T \subseteq (Q \times \Sigma \cup \{\varepsilon\} \times G \cup \{\varepsilon\}) \times (Q \times G \cup \{\varepsilon\})$

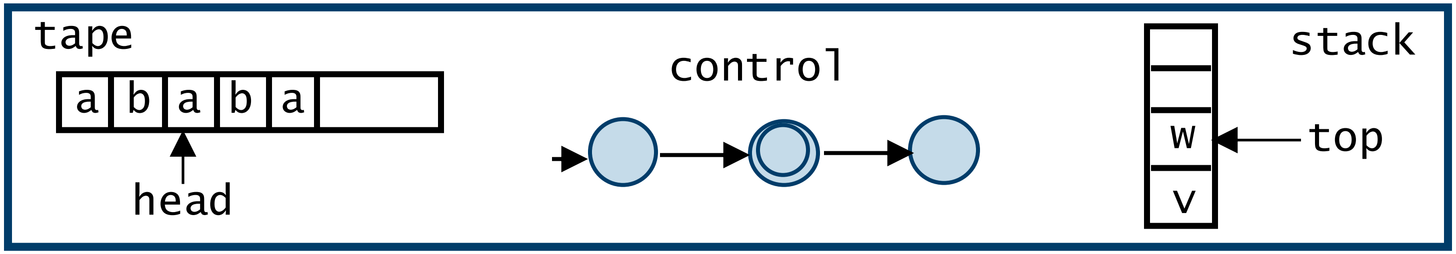


Informally, the transition $(q_1, a, w) \rightarrow (q_2, v)$ means that

– if we are in state q_1
– if $a \neq \varepsilon$, then the symbol a is at the head of the tape
– if $w \neq \varepsilon$, then the symbol w is is on top of the stack
– then move to state q_2 and

Pushdown automata

Transition relation $T \subseteq (Q \times \Sigma\cup\{\varepsilon\} \times G\cup\{\varepsilon\}) \times (Q \times G\cup\{\varepsilon\})$

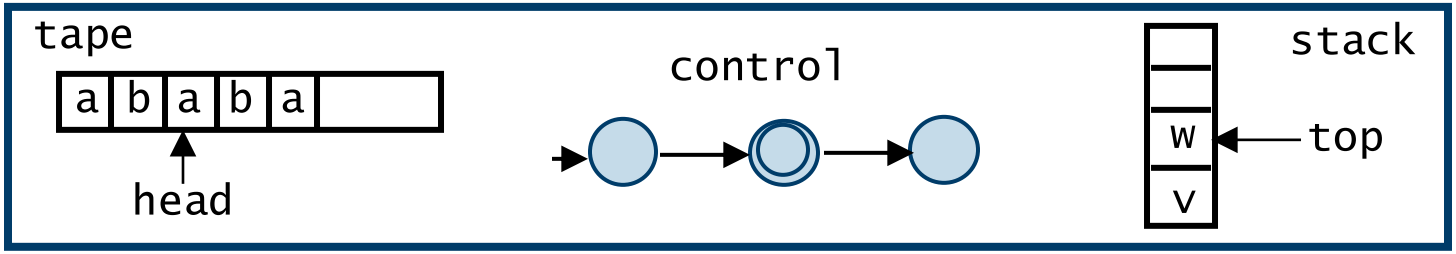


Informally, the transition $(q_1, \mathbf{a}, w) \rightarrow (q_2, v)$ means that

- if we are in state q_1
- if $a \neq \varepsilon$, then the symbol a is at the head of the tape
- if $w \neq \varepsilon$, then the symbol w is is on top of the stack
- then move to state q_2 and
- if $a \neq \varepsilon$, then move head forward one position
  - i.e. we have read the symbol a from the head of the tape

Pushdown automata

Transition relation $T \subseteq (Q \times \Sigma\cup\{\varepsilon\} \times G\cup\{\varepsilon\}) \times (Q \times G\cup\{\varepsilon\})$

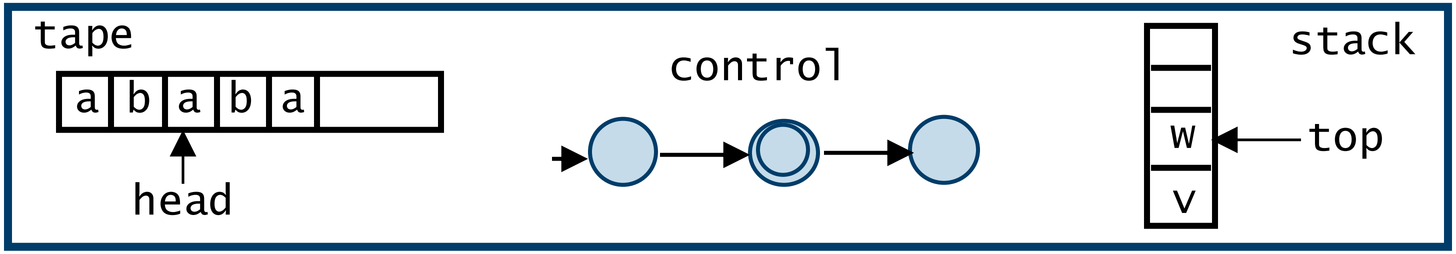


Informally, the transition $(q_1, a, w) \rightarrow (q_2, v)$ means that

- if we are in state q_1
- if $a \neq \varepsilon$, then the symbol a is at the head of the tape
- if $w \neq \varepsilon$, then the symbol w is is on top of the stack
- then move to state q_2 and
- if $a \neq \varepsilon$, then move head forward one position
- if $w \neq \varepsilon$, then **pop** w from the stack
  - a requirement was that w is on the top of the stack

Pushdown automata

Transition relation $T \subseteq (Q \times \Sigma\cup\{\varepsilon\} \times G\cup\{\varepsilon\}) \times (Q \times G\cup\{\varepsilon\})$



Informally, the transition $(q_1, a, w) \rightarrow (q_2, v)$ means that

- if we are in state q_1
- if $a \neq \varepsilon$, then the symbol a is at the head of the tape
- if $w \neq \varepsilon$, then the symbol w is is on top of the stack
- then move to state q_2 and
- if $a \neq \varepsilon$, then move head forward one position
- if $w \neq \varepsilon$, then **pop** w from the stack
- if $v \neq \varepsilon$, then **push** v onto the stack

Pushdown automata

A PDA accepts an input if and only if after the input has been read, the stack is empty and control is in an accepting state

Example tuples from a PDA program when in state q_1

- $(q_1,\varepsilon,\varepsilon)\rightarrow(q_2,\varepsilon)$ move to q_2
- $(q_1,a,\varepsilon)\rightarrow(q_2,\varepsilon)$ if head of tape is a, move to q_2 & move head forward
- $(q_1,a,\varepsilon)\rightarrow(q_2,v)$ if head of tape is a, move to q_2, move head forward & push v onto stack
- $(q_1,a,w)\rightarrow(q_2,\varepsilon)$ if head of tape is a & w is top stack, move to q_2, move head forward & pop w from stack
- $(q_1,a,w)\rightarrow(q_2,v)$ if head of tape is a & w is top of stack, move to q_2, move head forward, pop w & push v onto stack

Pushdown automata

There is no explicit test that the stack is empty

- this can be achieved by adding a special symbol (\$) to the stack at the start of the computation
- i.e. we add the symbol to the stack when we know the stack is empty and we never add \$ at any other point during the computation
  - unless we pop it from the stack as at this point we again know its empty
- then can check for emptiness by checking \$ is on top of the stack

- when we want to finish in an accepting state we just need to make sure we pop \$ from the stack (we will see this in an example later)

Pushdown automata

Note PDA defined here are non-deterministic (NDPDA)

- deterministic PDAs (DPDAs) are less powerful
- this differs from DFAs where non-determinism does not add power
- i.e. there are languages that can be recognised by a NDPDA but not by a DPDA, e.g. the language of palindromes
  - palindromes: strings that read the same forwards and backwards

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

abcdeedcba

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

abcdeedcba   abcde   edcba

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

abcdeedcba

abcde

edcba

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

abcdeedcba

abcde
edcba

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

abcdefedcba

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

abcdefedcba   abcde f edcba

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

abcdefedcba

abcde

f

edcba

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

abcdefedcba

abcde

edcba

f

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

Why do we need non-determinism?

- we need to "guess" where the middle of the stack is
  - and if there are even or odd number of characters
- cannot work this out first and then check the string as would need
  - to read the string twice (as only have a stack)
  - an unbounded number of states as the string could be of any finite length

Pushdown automata – Palindromes

Palindromes are sequences of characters that read the same forwards and backwards (second half is the reverse of the first half)

How to recognize palindromes with a pushdown automaton?

- push the first half of the sequence onto the stack
- then as we read each new character check it is the same as the top element on the the stack and pop this element
- then enter an accepting state if all checks succeed

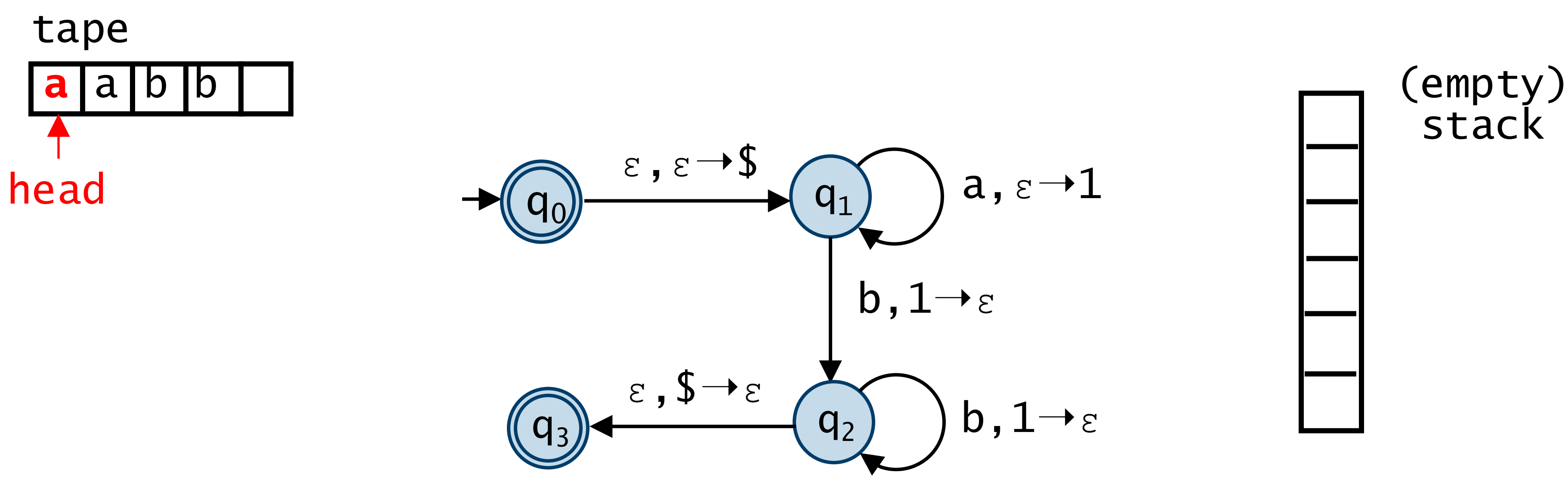
Why do we need non-determinism?

- we need to "guess" where the middle of the stack is
  - and if there are even or odd number of characters
- cannot work this out first and then check the string as would need
  - to read the string twice (as only have a stack)
  - an unbounded number of states as the string could be of any finite length

Pushdown automata – Example

Consider the following PDA program (alphabet is **{a,b}**)

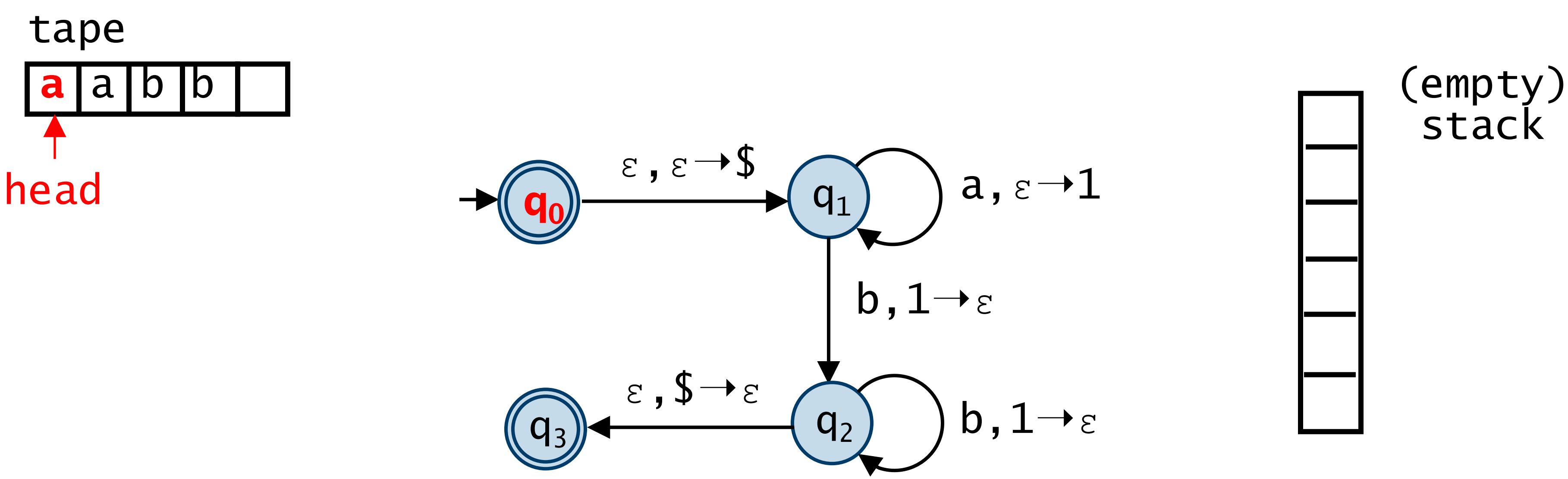
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\to(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\to(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\to(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\to(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\to(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

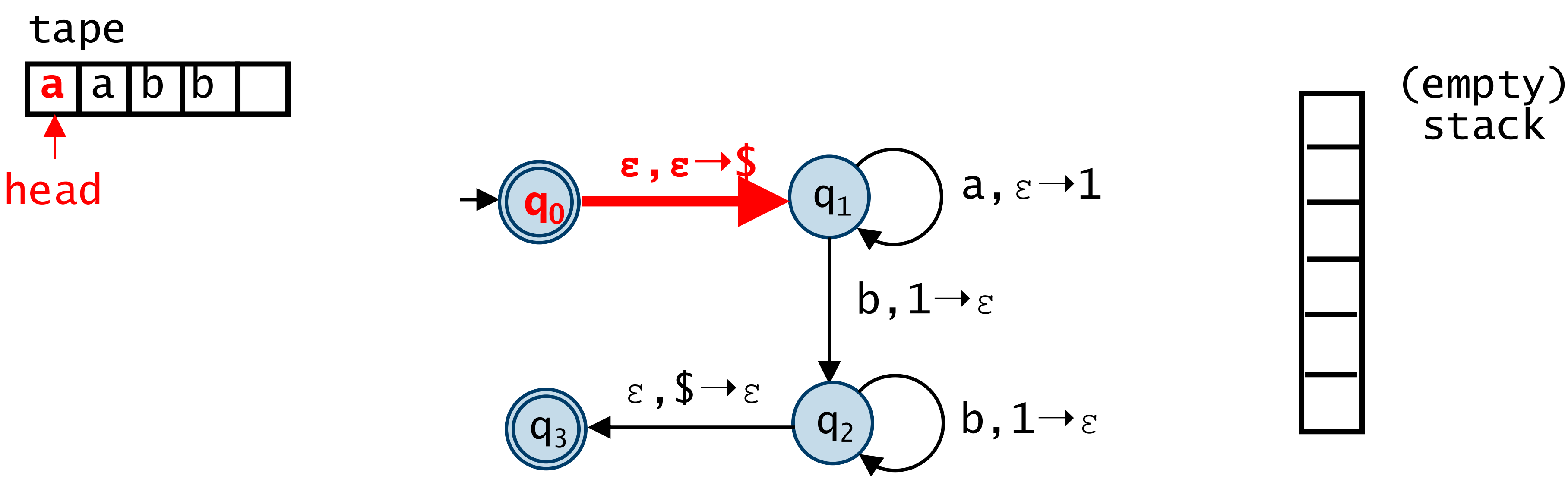
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

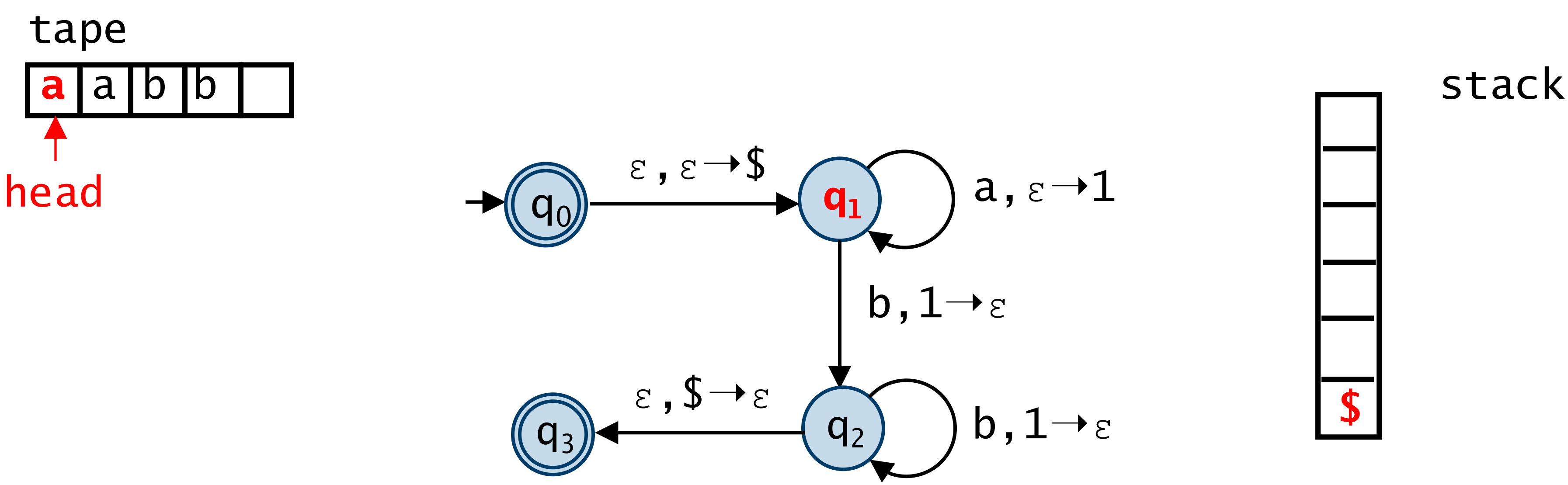
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

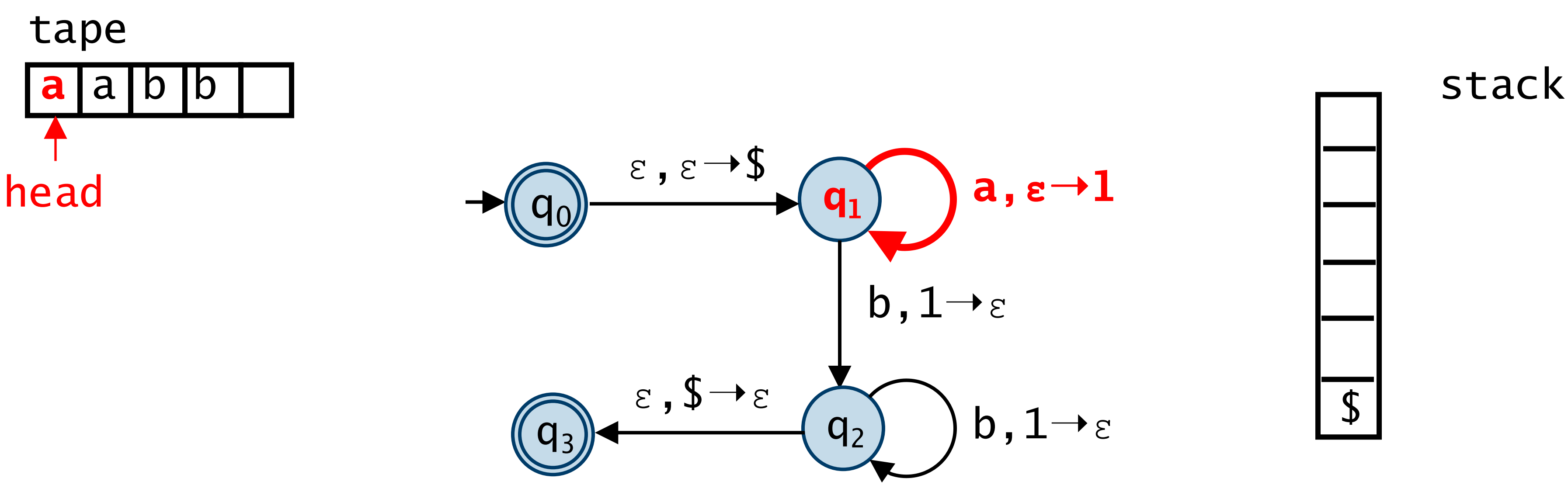
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

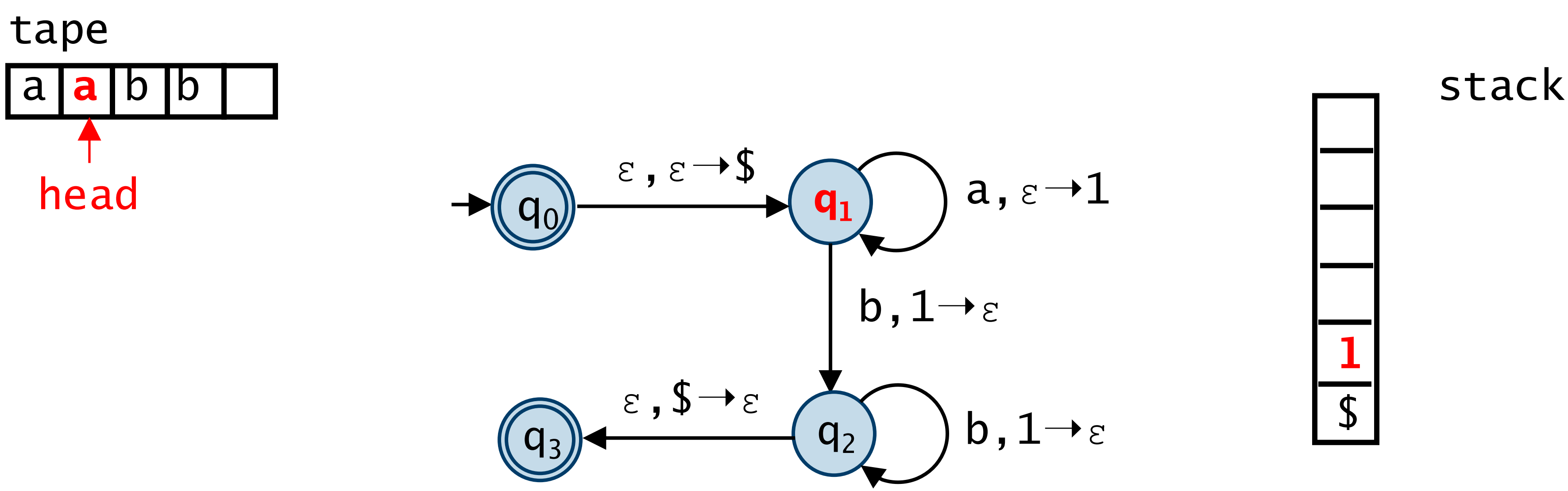
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

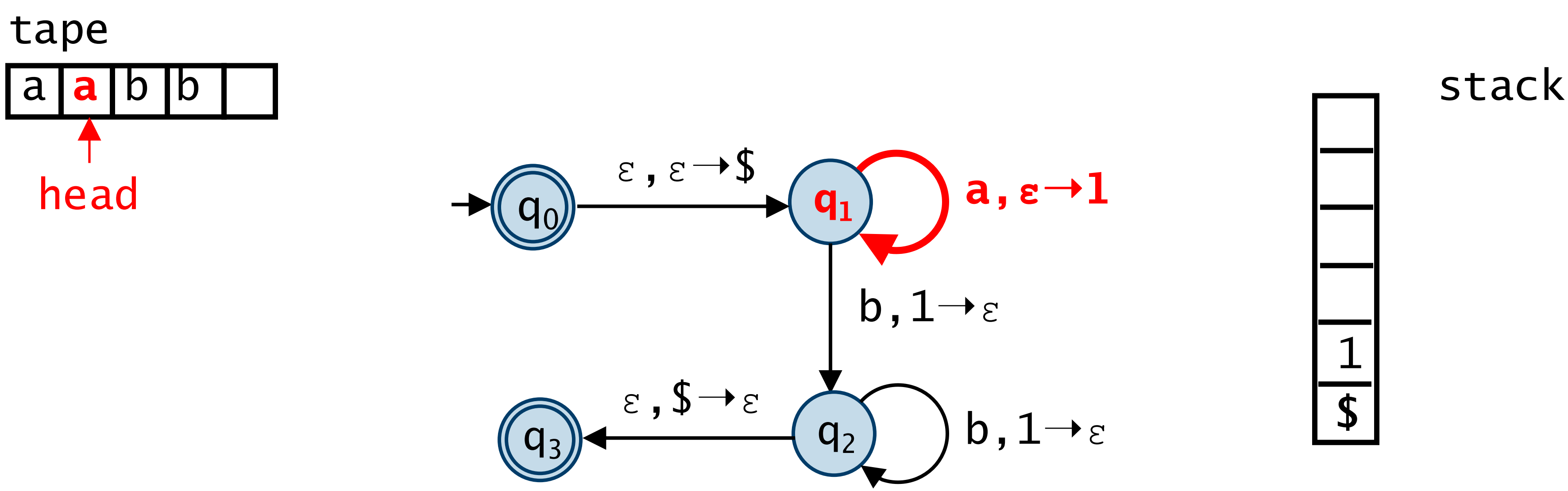
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

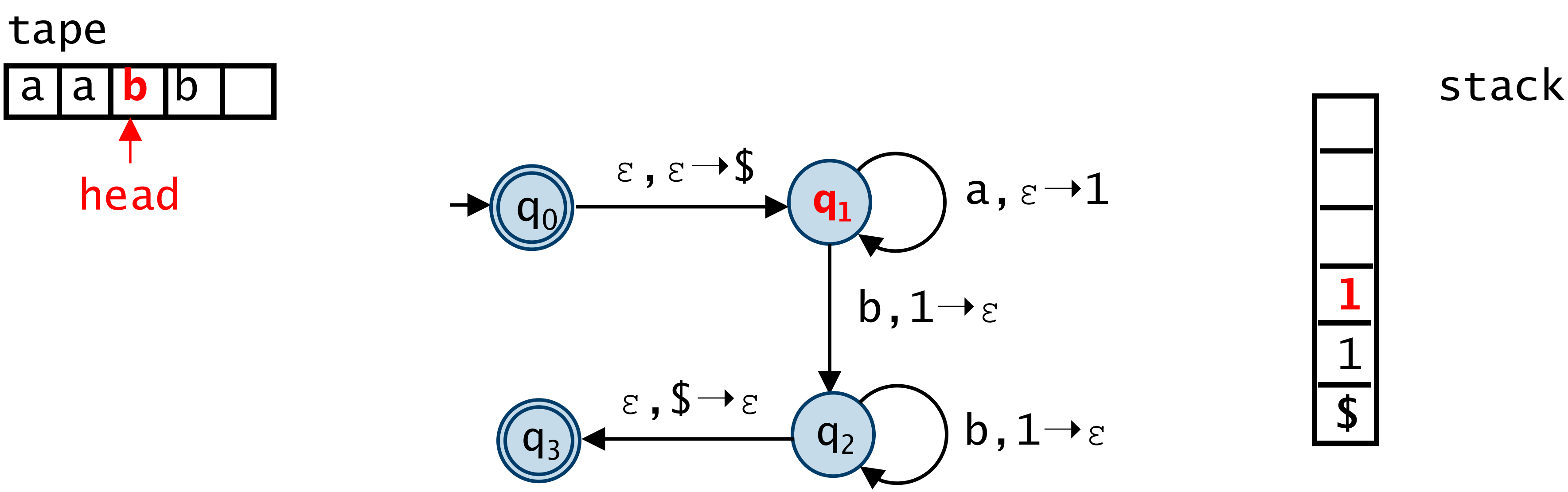
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

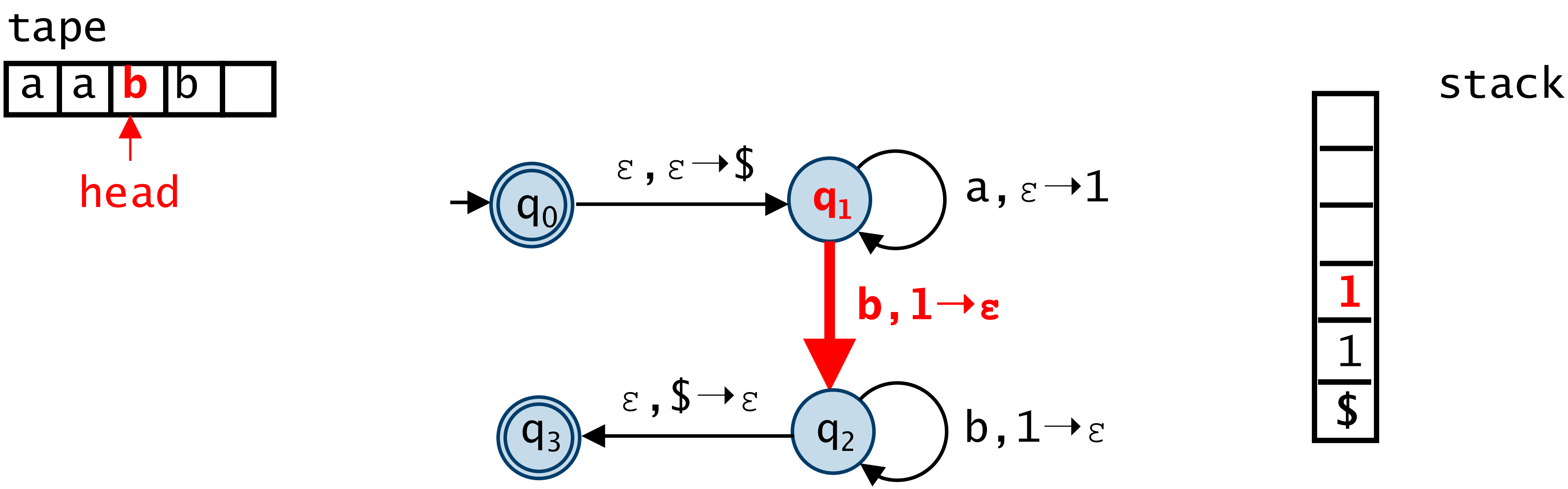
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

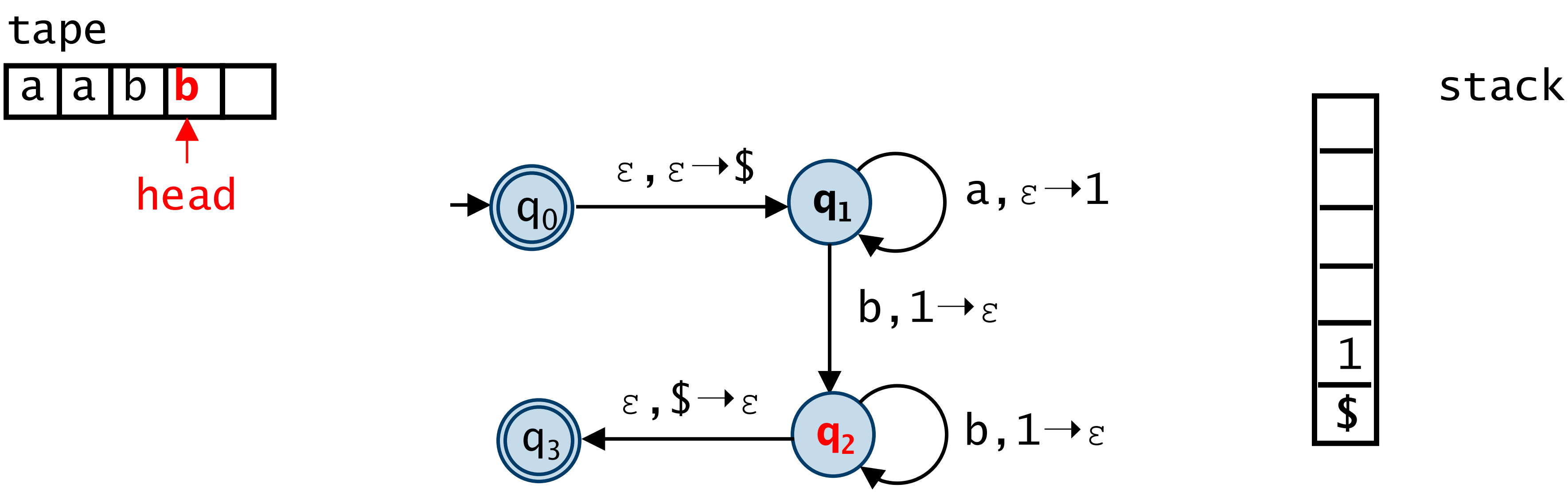
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

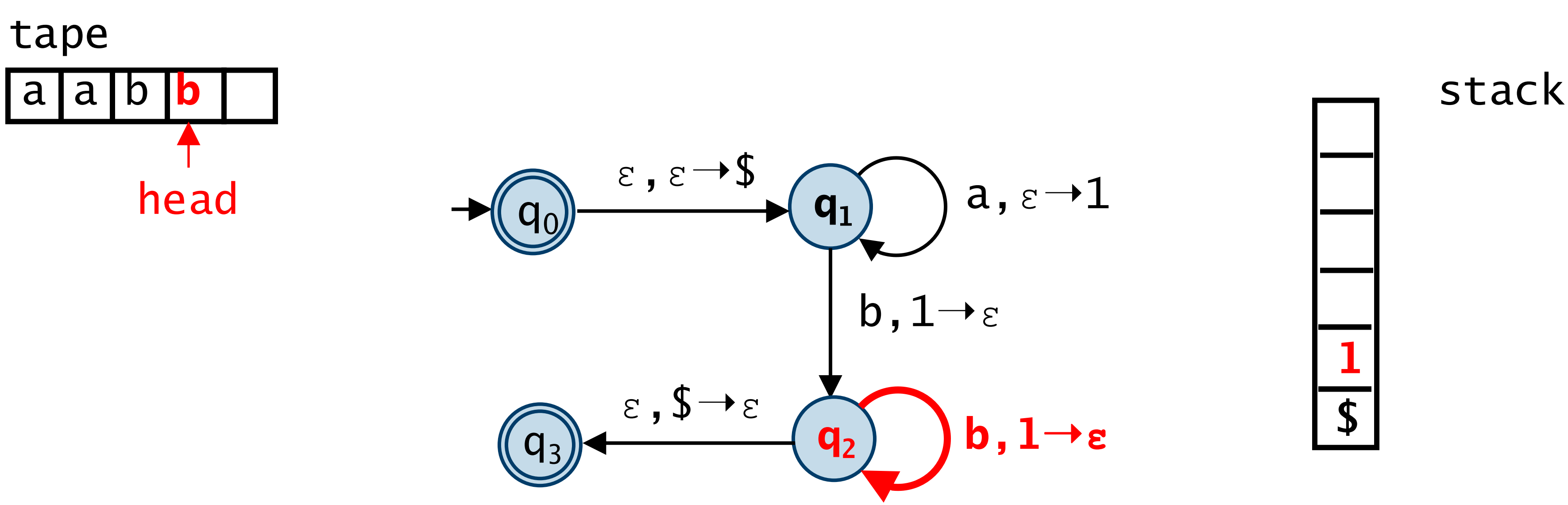
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

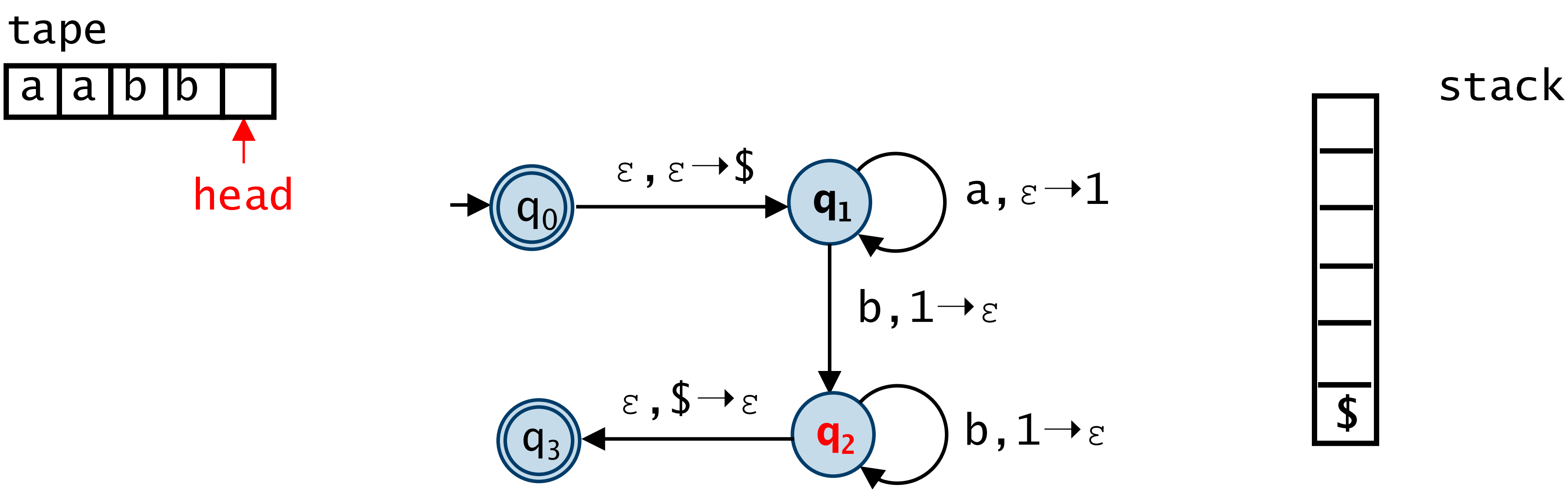
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

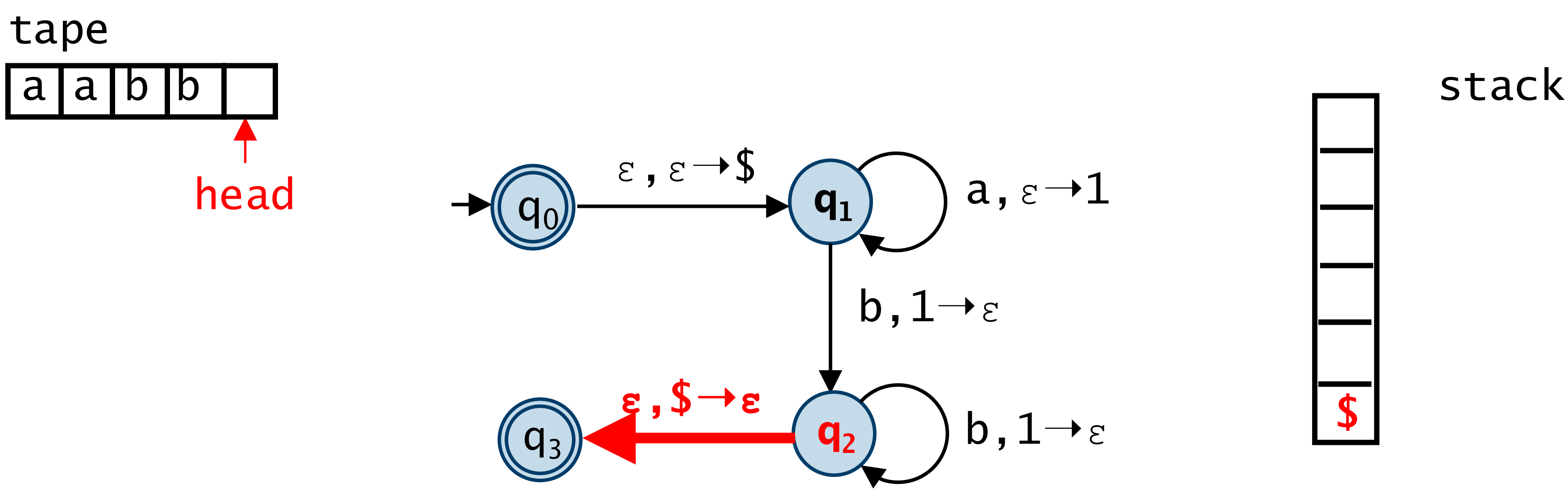
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\to(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\to(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\to(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\to(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\to(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

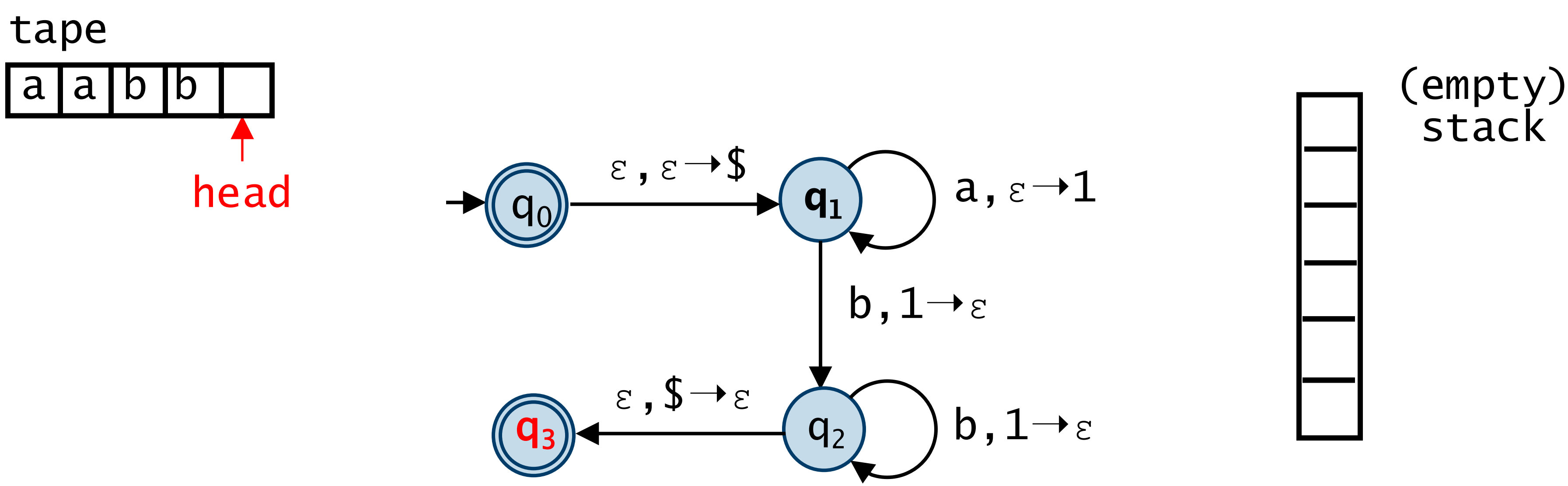
- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0, \varepsilon, \varepsilon) \rightarrow (q_1, \$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1, a, \varepsilon) \rightarrow (q_1, 1)$ read a & push 1 onto stack
- $(q_1, b, 1) \rightarrow (q_2, \varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2, b, 1) \rightarrow (q_2, \varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2, \varepsilon, \$) \rightarrow (q_3, \varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\to(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\to(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\to(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\to(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\to(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3



Pushdown automata – Example

Consider the following PDA program (alphabet is {a,b})

- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3

Example inputs

- if you try to recognise aabb, all of the input is read, as we have just seen end up in an accepting state, and the stack is empty
- if you try to recognise aaabb, all the input is read, you end up in state q_2 and the stack in not empty
- if you try to recognise aabbb, you are left with b on the tape, which cannot be read because of an empty stack

Pushdown automata – Example

Consider the following PDA program (alphabet is **{a,b}**)

- q_0 is the start state and q_0 and q_3 are the only accepting states
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ move to q_1 and push \$ onto stack (\$ – special symbol)
- $(q_1,a,\varepsilon)\rightarrow(q_1,1)$ read a & push 1 onto stack
- $(q_1,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack & move to q_2
- $(q_2,b,1)\rightarrow(q_2,\varepsilon)$ read b & 1 is top of stack, pop stack
- $(q_2,\varepsilon,\$)\rightarrow(q_3,\varepsilon)$ if \$ is the top of the stack, pop stack & move to q_3

Automaton recognises the language: $\{\ a^n b^n \mid n \geq 0\ \}$

Pushdown automata

Pushdown automata are more powerful than finite-state automata

- a PDA can recognise some languages that cannot be recognised by a DFA
- e.g. $\{a^n b^n \mid n \geq 0\}$ is recognised by the PDA example

The languages that can be recognised by a PDA are the context-free languages

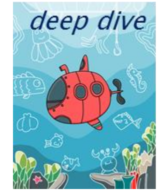
Are all languages regular or context-free?

i.e. is a PDA an adequate model of a general-purpose computer (our 'black box')?

No, for example, consider the language $\{a^n b^n c^n \mid n \geq 0\}$

- this cannot be recognised by a PDA
- but it is easy to write a program (say in Java) to recognise it
- next lecture – Turing machines as general model of a computer

Applications of DFAs & PDAs



Theoretical

- understanding computability and complexity
- formal specification of software behaviour and verifying properties

Practical applications in software development

- DFAs - design compilers and interpreters, lexical analysis
- regexp - string matching and validation
- DFAs - modelling and analysing communication protocols as finite state machines in embedded systems, network protocols, user interfaces
- context-free grammars/CFGs (which generate context-free languages/CFLs) used for defining the syntactic structure of programming languages - useful for designing new programming languages
- CFGs are used for data interchange formats like JSON and XML (nested structures)

Computational thinking skills, problem-solving

Hierarchy of grammars and automata

Regular Languages (type 3)

- grammar: Regular Grammars
- automaton: Finite Automata (FA) – both Deterministic (DFA) and Non-deterministic (NFA)

Context-Free Languages (type 2)

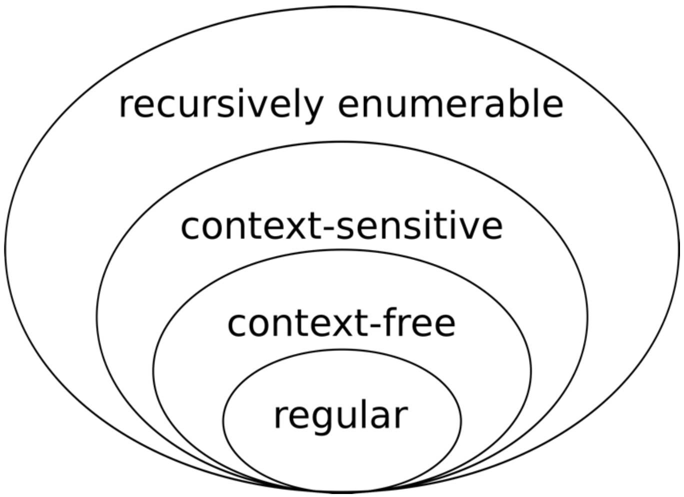
- grammar: Context-Free Grammars (CFG)
- automaton: Pushdown Automata (PDA)

Context-Sensitive Languages (type 1)

- grammar: Context-Sensitive Grammars (CSG)
- automaton: Linear Bounded Automaton (LBA)

Recursively Enumerable Languages (type 0)

- grammar: Unrestricted Grammars
- automaton: Turing Machine (TM) – both Deterministic (DTM) and Non-deterministic (NTM)



https://en.wikipedia.org/wiki/Chomsky_hierarchy

Chomsky hierarchy
introduced in the 1950's by Noam Chomsky

Next time – Section 5 – Computability

Introduction

Models of computation

- finite-state automata – regular languages and regular expressions
- pushdown automata
- Turing machines
- Counter machines
- Church–Turing thesis