Algorithmics I

Section 5 – Computability

Dr. Gethin Norman

School of Computing Science
University of Glasgow

gethin.norman@glasgow.ac.uk

Introduction to Computability

What is a computer?

`input x` ⟶ $\boxed{\texttt{black box}}$ ⟶ `output f(x)`

What can the black box do?

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

Computability concerns which functions can be computed

- a formal way of answering ‘what problems can be solved by a computer?’
- or alternatively ‘what 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
- or what an algorithm is if we view a computer as a device that can execute an algorithm

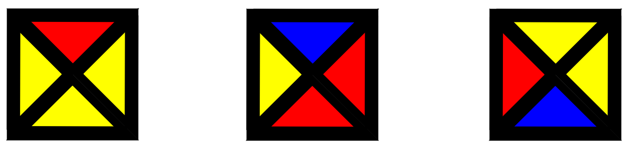
Unsolvable problems

Some problems cannot be solved by a computer

- even with unbounded time

Example: The Tiling Problem

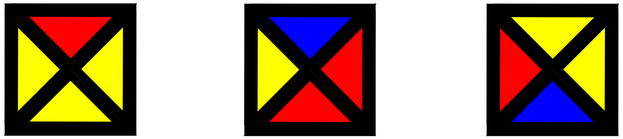
- a tile is a 1×1 square, divided into 4 triangles by its diagonals with each triangle is given a colour
- each tile has a fixed orientation (no rotations allowed)
- example tiles:



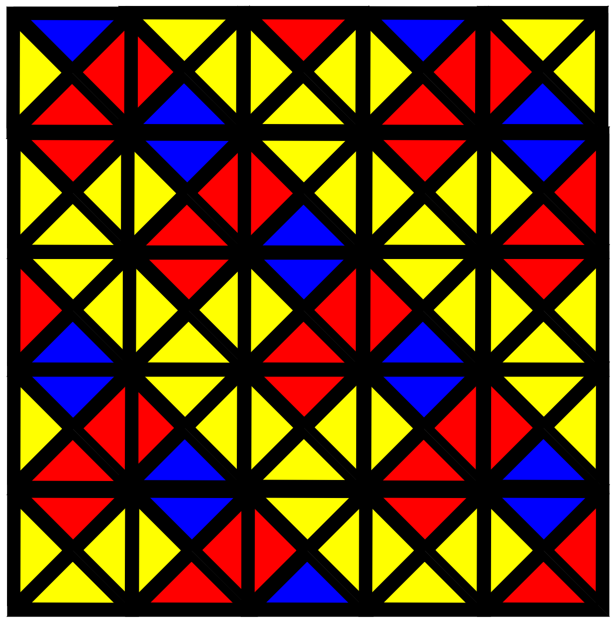
Instance: a finite set S of tile descriptions

Question: can any finite area, of any size, be completely covered using only tiles of types in S, so that adjacent tiles colour match?

Tiling problem – Tiling a 5×5 square

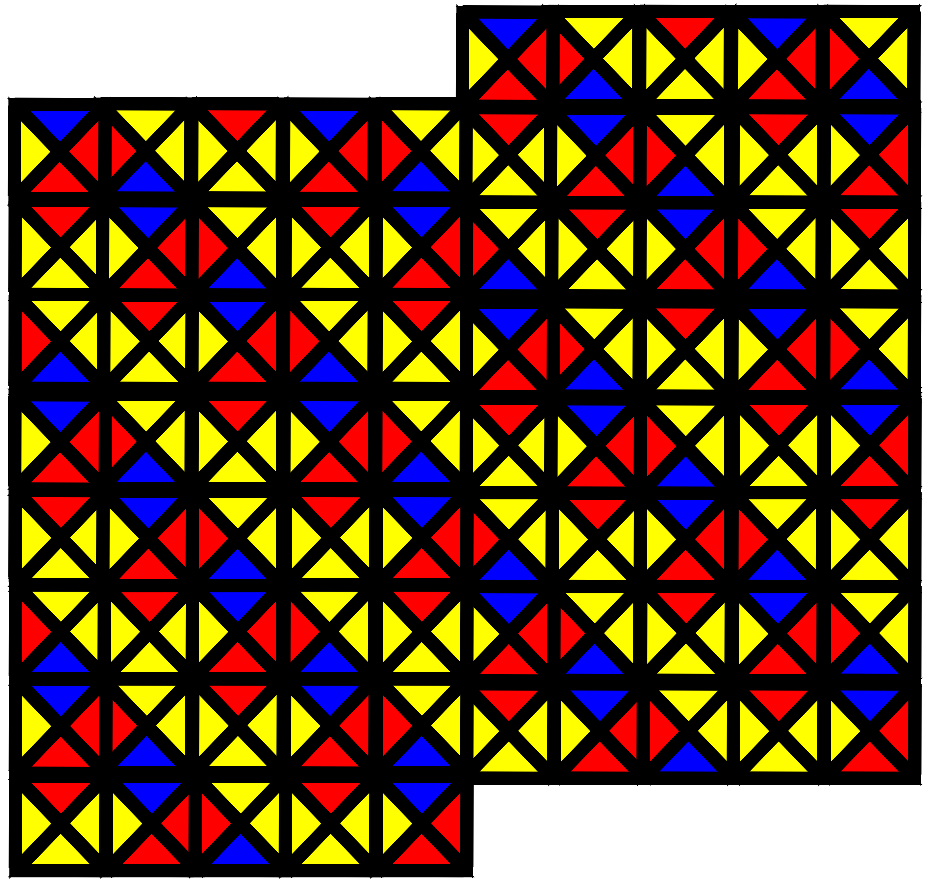
Available tiles: 

We can use these tiles to tile a 5×5 square as follows:



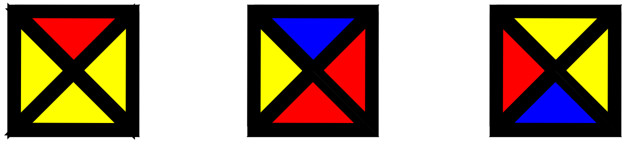
Tiling problem – Extending to a larger region

- Overlap the top two rows with the bottom two rows
  - obtain an 8×5 tiled area
- Next place two of these 8×5 rectangles side by side
  - with the right hand rectangle one row above the left hand rectangle
- By repeating this pattern it follows that any finite area can be tiled

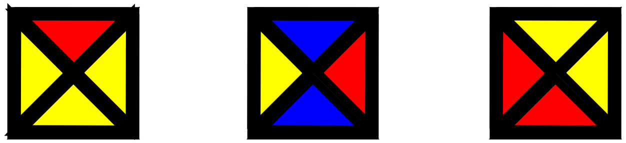


Tiling problem – Altering the tiles

Original tiles:



New tiles:



Now impossible to tile a 3×3 square

There are 3^9=19,683 possibilities if you want to try them all out…

Tiling problem

Tiling problem: given a set of tile descriptions, can any finite area, of any size, be completely 'tiled' using only tiles from this set?

There is no algorithm for the tiling problem

- for any algorithm A that we might try to formulate there is a set of tiles S for which either A does not terminate or A gives the wrong answer

The problem is that:

- "any size" means we have to check all finite areas and there are infinitely many of these
- and for certain sets of tile descriptions that can tile any area, there is no "repeated pattern" we can use
- so to be correct the algorithm would really have to check all finite areas

Undecidable problems

A problem Π that admits no algorithm is called non-computable or unsolvable

If Π is a decision problem and Π admits no algorithm it is called undecidable

The Tiling Problem is undecidable

Post's correspondence problem (PCP)

A **word** is a finite string over some given finite alphabet

Instance: two finite sequences of words $X_1,\dots,X_n$ and $Y_1,\dots,Y_n$

- the words are all over the same alphabet

Question: does there exist a sequence $i_1, i_2, \dots, i_r$ of integers chosen from $\{1,\dots,n\}$ such that $X_{i1}X_{i2}\dots X_{ir} = Y_{i1}Y_{i2}\dots Y_{ir}$?

- i.e. concatenating the X_{ij}'s and the Y_{ij}'s gives the same result

Example: **n=5**

- X_1=abb, X_2=a, X_3=bab, X_4=baba, X_5=aba
- Y_1=bbab, Y_2=aa, Y_3=ab, Y_4=aa, Y_5=a
- correspondence is given by the sequence 2, 1, 1, 4, 1, 5
  - word constructed from X_i's: **aabbabbbabaabbaba**
  - word constructed from Y_i's: **aabbabbbabaabbaba**

Post’s correspondence problem (PCP)

A **word** is a finite string over some given finite alphabet

Instance: two finite sequences of words $X_1,\dots,X_n$ and $Y_1,\dots,Y_n$

- the words are all over the same alphabet

Question: does there exist a sequence $i_1,i_2,\dots,i_r$ of integers chosen from $\{1,\dots,n\}$ such that $X_{i1}X_{i2}\dots X_{ir} = Y_{i1}Y_{i2}\dots Y_{ir}$?

- i.e. concatenating the X_{ij}'s and the Y_{ij}'s gives the same result

Example: $n=5$ (with first letter from X_1 and Y_1 removed)

- X_1=bb, X_2=a, X_3=bab, X_4=bab, X_5=aba
- Y_1=bab, Y_2=aa, Y_3=ab, Y_4=aa, Y_5=a
- to get a match we must start with either 2 or 5
- follows that we can now never get a correspondence

Post’s Correspondence Problem is undecidable

The halting problem

An impossible project: write a program Q that takes as input

- a legal program X (say in Java)
- an input string S for program X

and returns as output

- **yes** if program X halts when run with input S
- **no** if program X enters an infinite loop when run with input S

We will prove that no such program Q can exists, meaning the halting problem is undecidable

The halting problem

Example (small) programs

```
public void test(int n){
  if (n == 1)
    while (true)
      null;
}
```

The program '`test`' will terminates if and only if input $n \neq 1$

The halting problem

Example (small) programs

```
public int erratic(int n){
  while (n != 1)
    if (n % 2 == 0) n = n/2;
    else n = 3*n + 1;
}
```

For example if '**erratic**' is called with **n=7** sequence of values:

22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

Nobody knows whether '**erratic**' terminates for all values of **n**

The halting problem – Undecidability

A formal definition of the halting problem (**HP**)

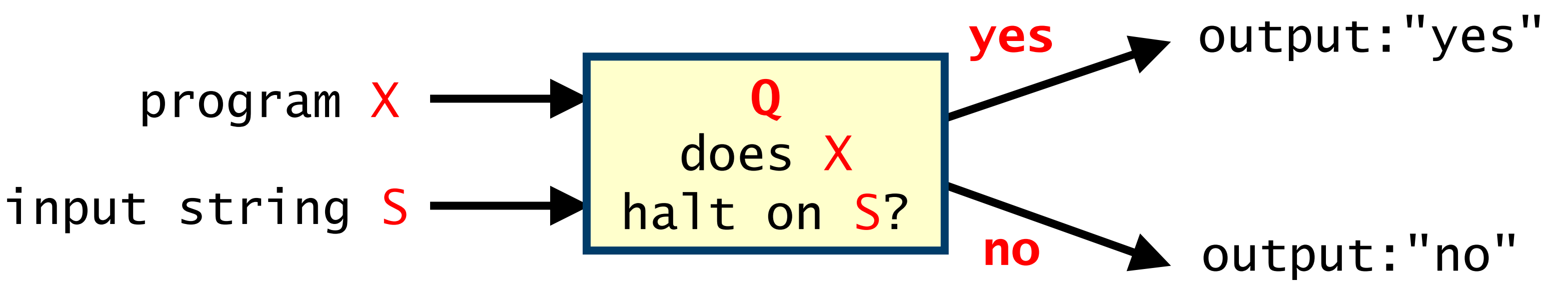
Instance: a legal Java program **X** and an input string **S** for **X**

- can substitute any language for Java

Question: does **X** halt when run on **S**?

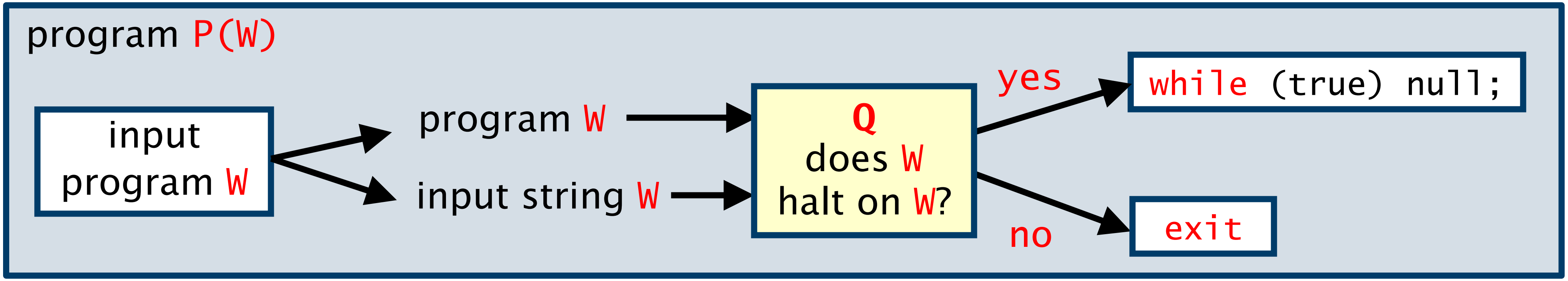
Theorem: **HP is undecidable proof** (by contradiction):

- suppose we have an algorithm A that decides (solves) HP
- let **Q** be an implementation of this algorithm as a Java method with X and S as parameters



The halting problem – Undecidability

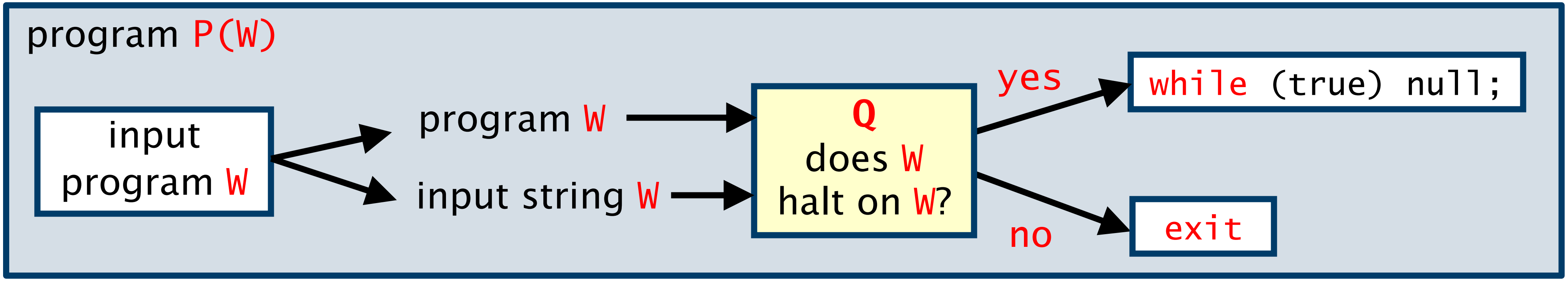
Define a new program P with input a legal program W in Java



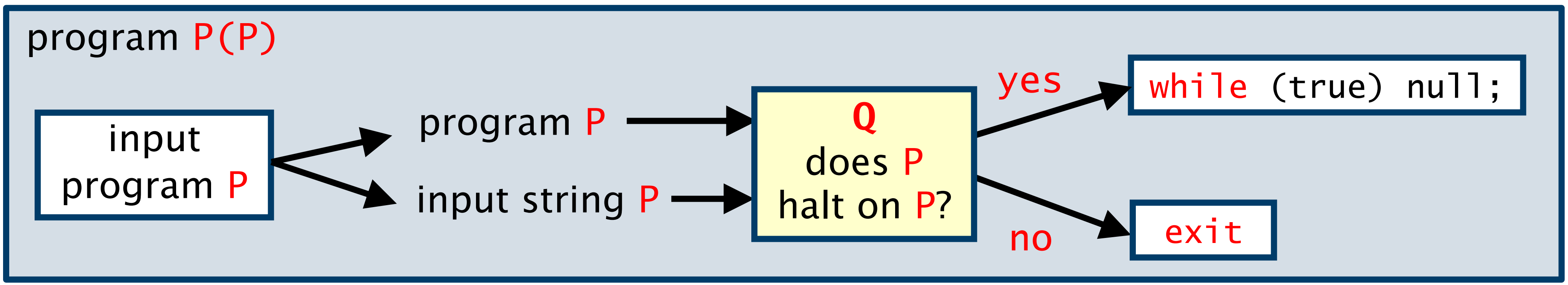
- P makes a copy of W and calls **Q**(W,W)
- **Q** terminates by assumption, returning either "**yes**" or "**no**"
- if **Q** returns "**yes**", then P enters an infinite loop
- if **Q** returns ”**no**", then P terminates

The halting problem – Undecidability

Define a new program **P** with input a legal program **W** in Java

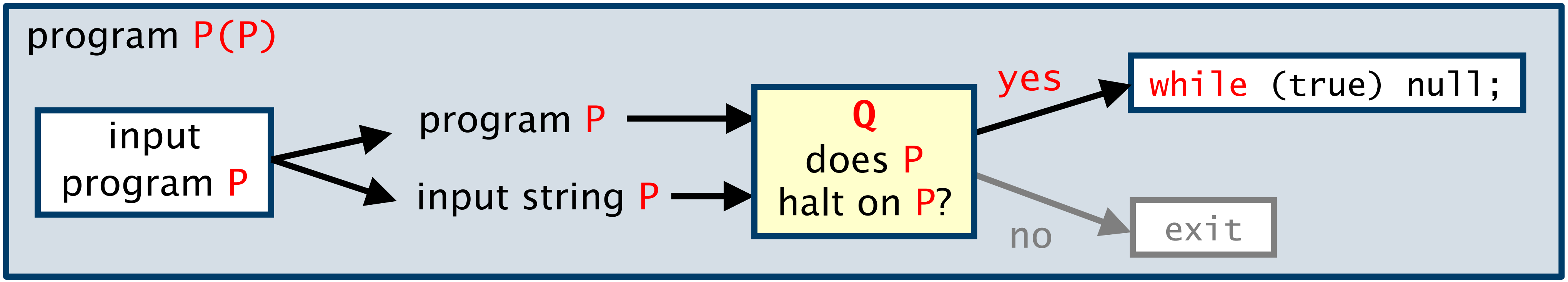


Now let the input **W** be the program **P** itself



The halting problem – Undecidability

Now let the input W to P be the program P itself

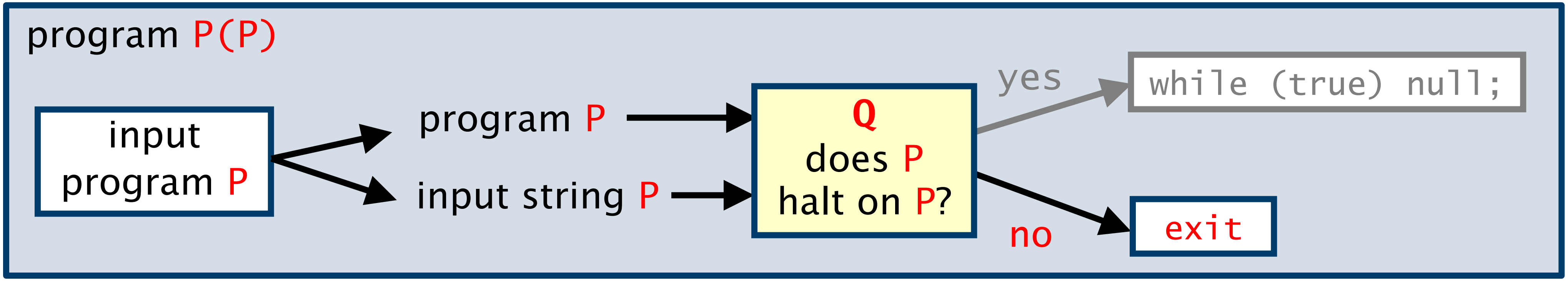


P calls Q(P,P)

- **Q** terminates by assumption, returning either "**yes**" or "**no**"
- recall we have assumed **Q** solves the halting problem
- suppose **Q** returns "**yes**", then by definition of **Q** this means P terminates
- but this also means P does not terminate (it enters the infinite loop)
- this is a contradiction therefore **Q** must return "**no**"

The halting problem – Undecidability

Now let the input W to P be the program P itself

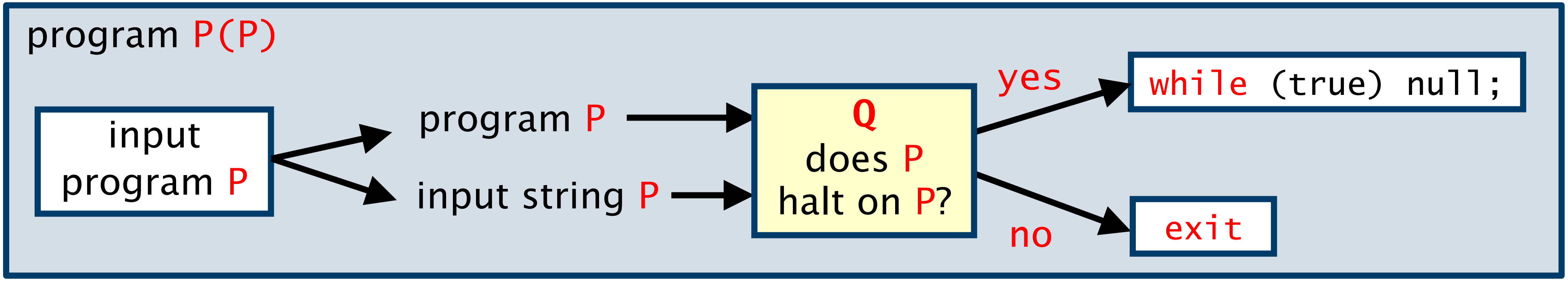


P calls Q(P,P)

- **Q** terminates by assumption, returning either "**yes**" or "**no**"
- recall we have assumed **Q** solves the halting problem
- therefore **Q** must return "**no**"
- this means by definition of **Q** that P does not terminate
- but this also means P does terminate
- so again a contradiction

The halting problem – Undecidability

Now let the input W to P be the program P itself



P calls Q(P,P)

- **Q** terminates by assumption, returning either "**yes**" or "**no**"
- recall we have assumed **Q** solves the halting problem
- therefore **Q** can return neither "**yes**" nor "**no**"
- meaning no such program **Q** can exist
- if no such **Q** can exist, then no algorithm can solve the halting problem
- hence the problem is undeciable

The halting problem – Undecidability

To summarise the proof

- we assumed the existence of an algorithm A that solved HP
- implemented this algorithm as the program **Q**
- then constructed a program P which contains **Q** as a subroutine
- showing that if **Q** gives the answer "**yes**", we reach a contradiction
- so **Q** must give the answer "**no**", but this also leads to a contradiction
- the contradiction stems from assuming that **Q**, and hence A exists
- therefore no algorithm A exists and HP is undecidable

Notice we are not concerned with the complexity of A just the existence of A

Proving undecidability by reduction

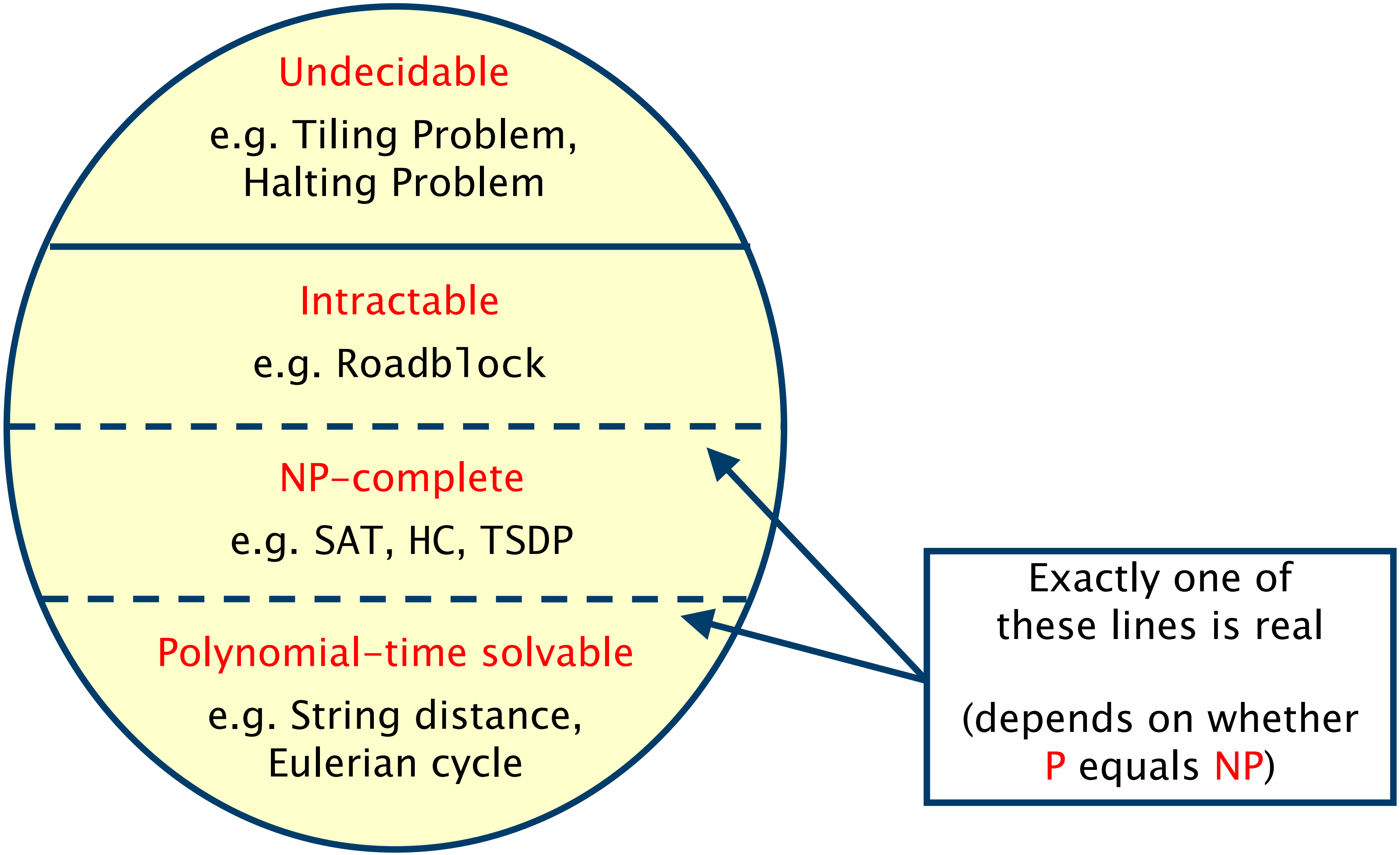
Suppose we can reduce any instance I of Π_1 into an instance J of Π_2 such that

- I has a '**yes**'-answer for Π_1 if and only if J has a "**yes**"-answer for Π_2 (like PTRs but no need for J to be constructed in polynomial time)

If Π_1 is undecidable and we can perform such a reduction, then Π_2 is undecidable

- suppose for a contradiction Π_2 is decidable
- then using this reduction we can decide Π_1
- however Π_1 is undecidable, therefore Π_2 cannot be decidable

Hierarchy of decision problems



Models of computation

input x → black box → output f(x)

Attempts to define "the black box”

- we will look at three classical models of computation of increasing power

- **Finite-State Automata**
  - simple machines with a fixed amount of memory
  - have very limited (but still useful) problem-solving ability
- **Pushdown Automata (PDA)**
  - simple machines with an unlimited memory that behaves like a stack
- **Turing machines (TM)**
  - simple machines with an unlimited memory that can be used essentially arbitrarily
  - these have essentially the same power as a typical computer

Deterministic finite-state automata

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

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'

- deterministic means that if

  $$((q,a_1),q_1),\ ((q,a_2),q_2) \in T \text{ either } a_1 \neq a_2 \text{ or } q_1 = q_2$$
- i.e. for any state and action there is at most one move (i.e. no choice)

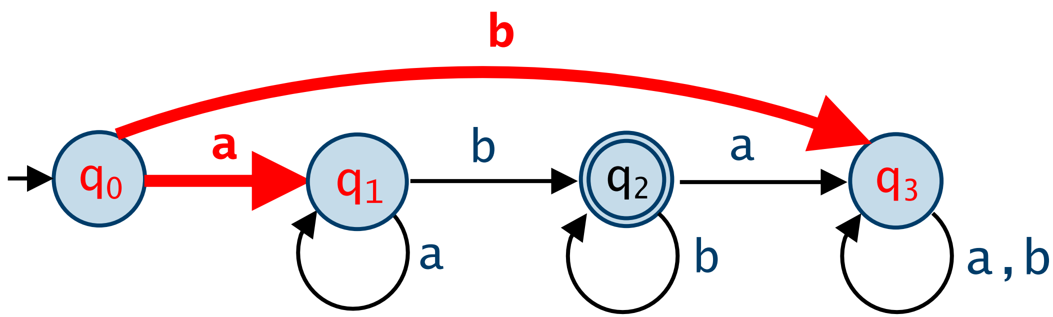
Deterministic finite-state automata

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

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 $T \subseteq (Q \times \Sigma) \times Q$

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



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

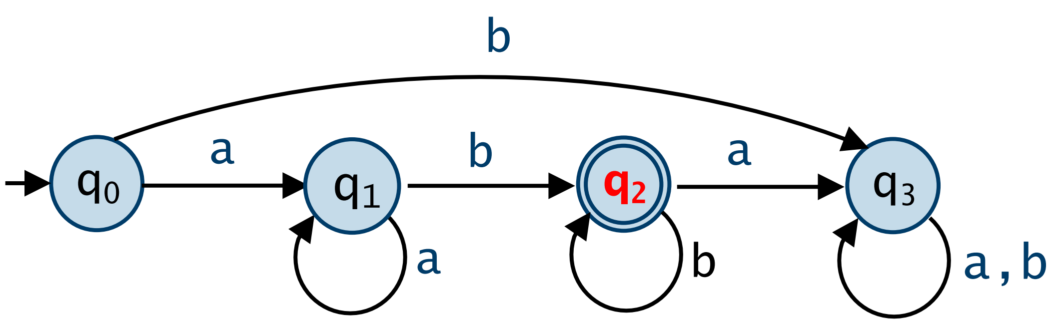
A DFA define a language

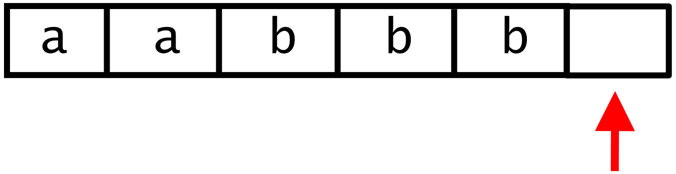
- determines whether the string on the input tape belongs to that language
- in other words, it solves a decision problem

More precisely a DFA recognises or accepts a language

- the input strings which when 'run' end in an accepting state

Question: what language does this DFA recognise?





string is accepted

Deterministic finite-state automata

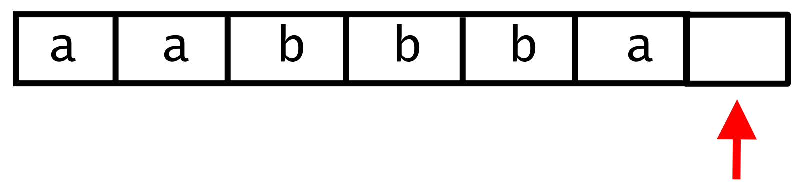
A DFA define a language

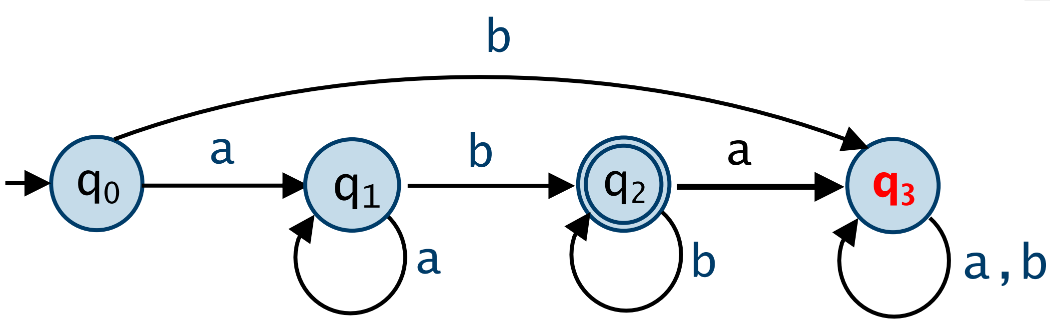
- determines whether the string on the input tape belongs to that language
- in other words, it solves a decision problem

More precisely a DFA recognises or accepts a language

- the input strings which when ‘run’ end in an accepting state

Question: what language does this DFA recognise?





string is not accepted

Deterministic finite-state automata

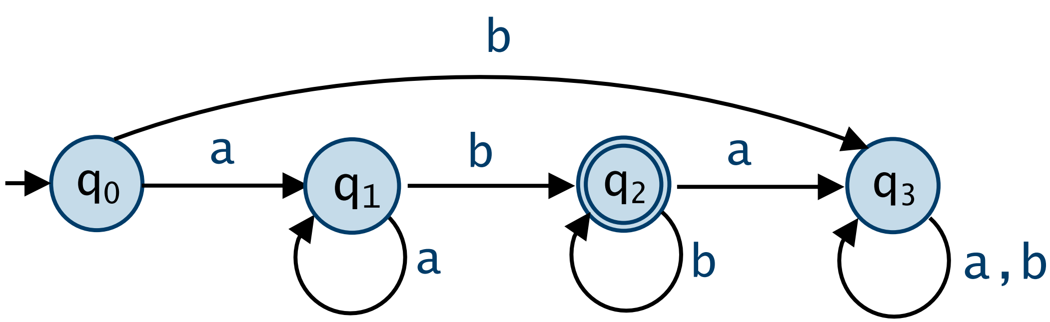
A DFA define a language

- determines whether the string on the input tape belongs to that language
- in other words, it solves a decision problem

More precisely a DFA recognises or accepts a language

- the input strings which when 'run' end in an accepting state

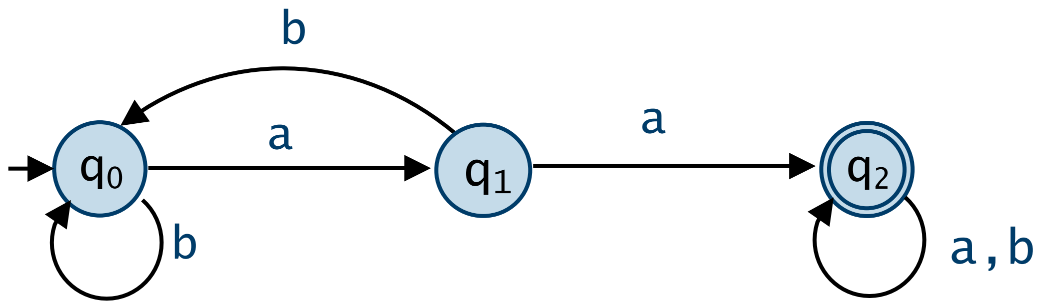
Question: what language does this DFA recognise?



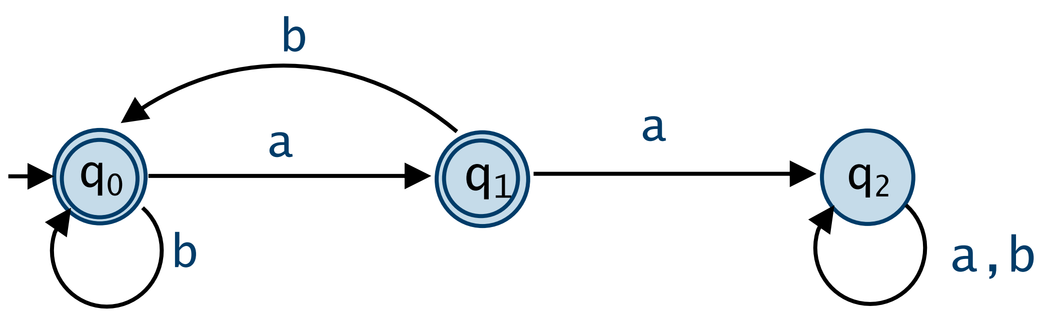
answer: the language consisting of the set of all strings comprising one or more a's followed by one or more b's

Deterministic finite-state automata

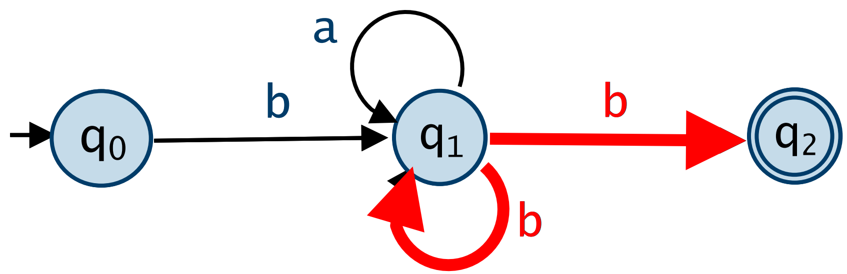
Recognises the language of strings containing two consecutive a's



Recognises the complement, i.e., the language of strings that do not contain two consecutive a's



Another example



Recognises strings that start and end with b

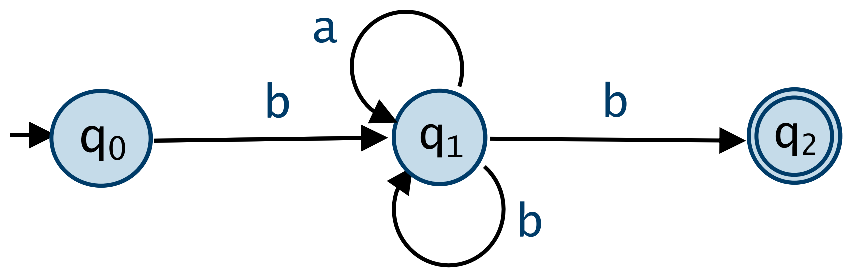
However this is not a DFA, but a non-deterministic finite-state automaton (NFA)

- 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

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

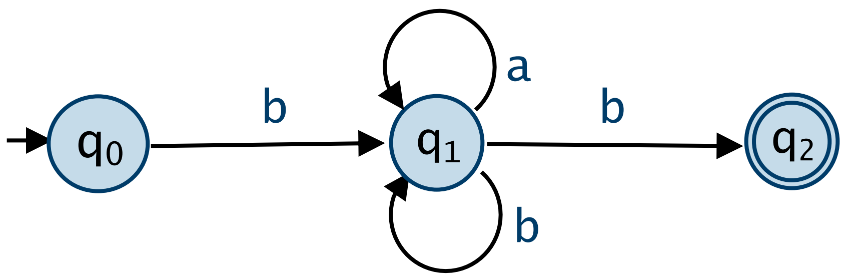
NFA to DFA reduction

Can reduce a NFA to a DFA using the subset construction

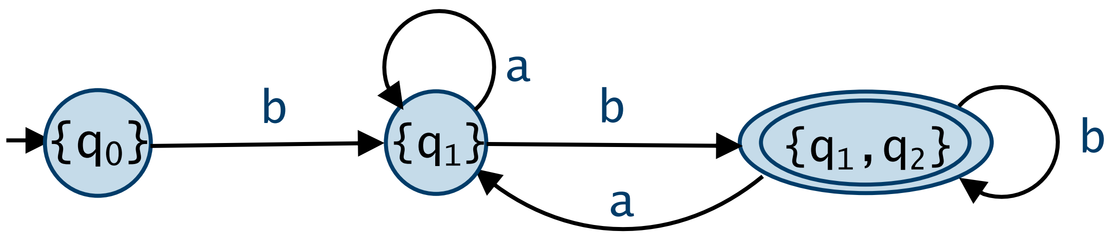
- states of the DFA are sets of states of the NFA
- construction can cause a blow-up in the number of states
  - in the worst case from N states to 2^N states

Example (without blow-up)

- recognises strings that start and end with b
- NFA



- DFA



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 $\sigma \in \Sigma$)

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) which is needed to override precedence between operators

Regular expressions

Order of precedence (highest first)

- closure (*) then concatenation then choice (|)
- as mentioned brackets can be 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 – Examples

The examples from earlier

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)*

3) the language of strings that do not contain two consecutive a's (harder)
   – 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 $L(R^{i+1})=L(RR^i)$
- note (a*b*)* is in fact equivalent to (a|b)*

L(R*) does not mean $\{\ r^* \mid r \in L(R)\ \}$

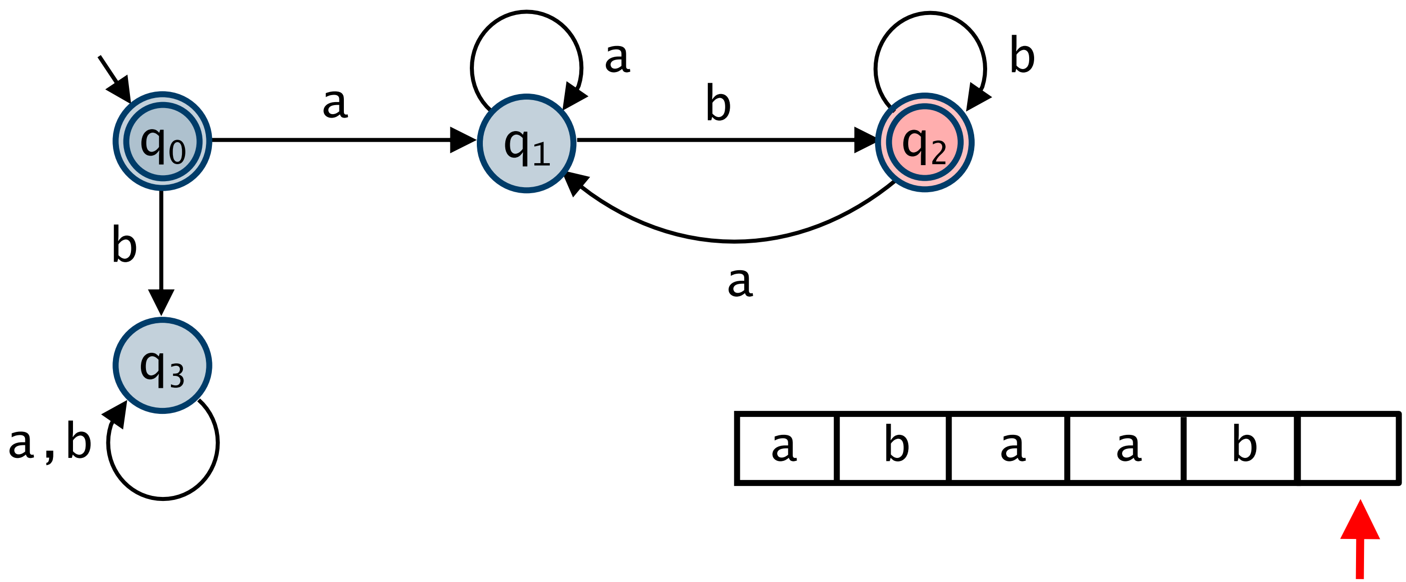
- which for certain regular expressions cannot be recognized by any DFA
- essentially for such a language would need a memory to remember which string in $r \in L(R)$ is repeated and there might be an unbounded number

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

A DFA cannot recognise { r* | r∈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'

Next: 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
- a finite set of states Q including start state and set of accepting states
- control or transition relation $T \subseteq (Q \times \Sigma \cup \{\varepsilon\} \times G \cup \{\varepsilon\}) \times (Q \times G \cup \{\varepsilon\})$

current state

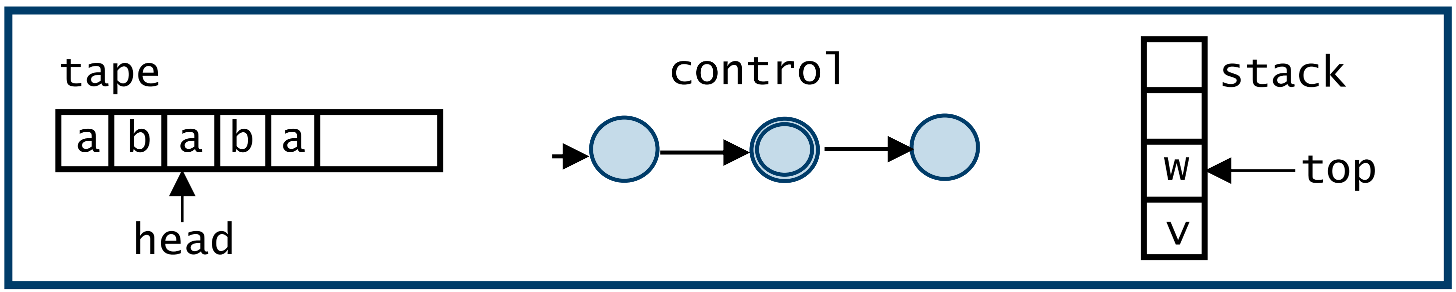
tape symbol or ε

old stack symbol or ε

new state

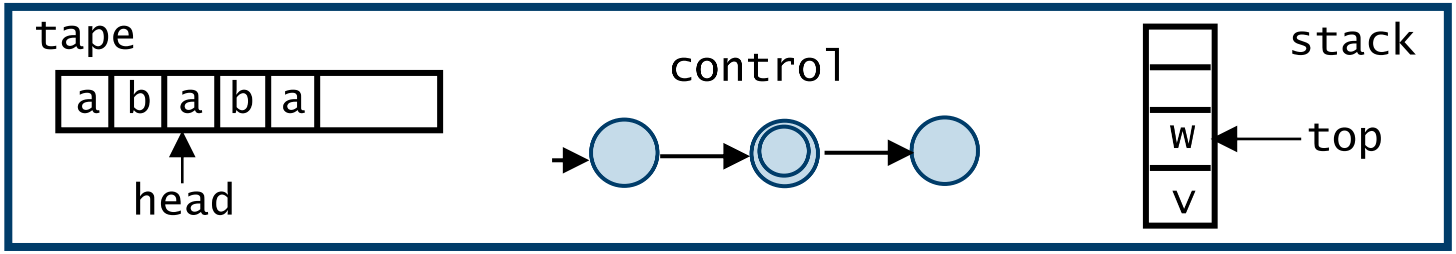
new stack symbol or ε

ε – empty string



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 $\mathbf{(q_1,a,w) \rightarrow (q_2,v)}$ **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

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

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

How to recognize palindrones 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

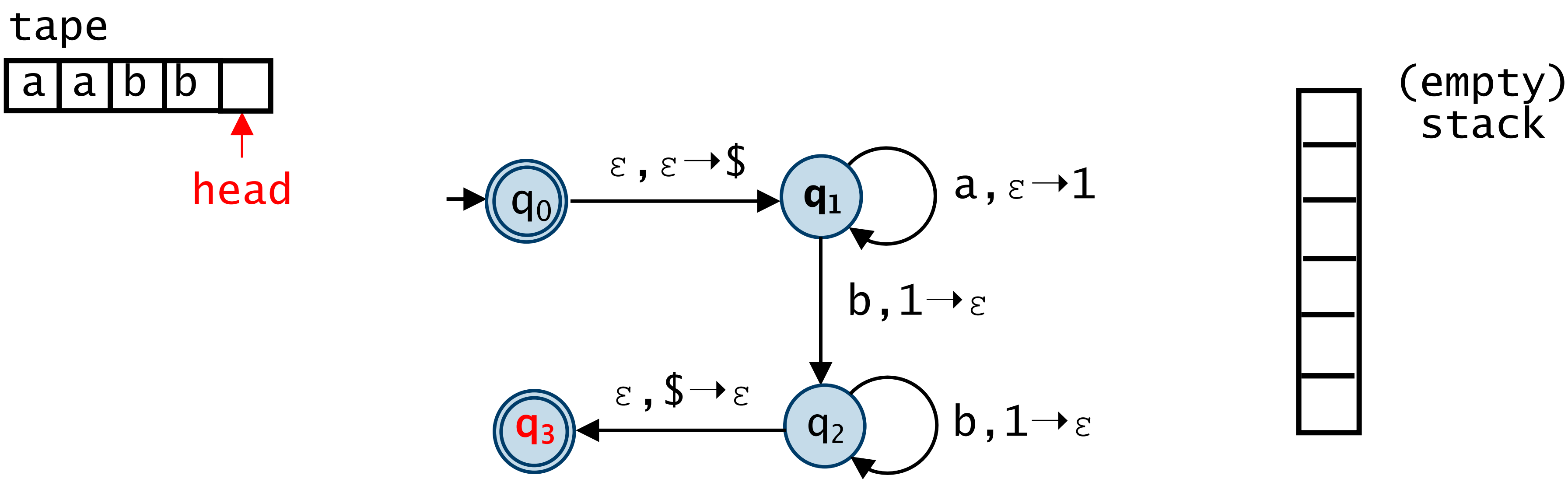
Why do we need non-determinism?

- 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 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
- $(q_0,\varepsilon,\varepsilon)\rightarrow(q_1,\$)$ 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

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

Turing machines

A **Turing Machine T** to recognise a particular language consists of

- a finite alphabet **Σ**, including a blank symbol (denoted by **#**)
- an unbounded **tape** of squares
  - each can hold a single symbol of Σ
  - tape unbounded in both directions
- a **tape head** that scans a single square
  - it can read from it and write to the square
  - then moves one square `left` or `right` along the tape
- a set **S** of **states**
  - includes a single start state s_0 and two halt (or terminal) states s_Y and s_N
- a **transition function**
  - essentially the inbuilt program

Turing machines – Computation

The **transition function** is of the form

$$f : ((S/\{s_Y,s_N\}) \times \Sigma) \rightarrow (S \times \Sigma \times \{\texttt{Left, Right}\})$$

For each non-terminal state and symbol the function **f** specifies

- a new state (perhaps unchanged)
- a new symbol (perhaps unchanged)
- a direction to move along the tape

$f(s,\sigma)=(s',\sigma',d)$ means reading symbol **σ** from the tape in state **s**

- move to state $s' \in S$
- overwrite the symbol σ on the tape with the symbol $\sigma' \in \Sigma$
  - if you do not want to overwrite the symbol write the symbol you read
- move the tape head one square in direction $d \in \{\texttt{Left, Right}\}$

Turing machines – Computation

The (finite) input string is placed on the tape

- assume initially all other squares of the tape contain blanks

The tape head is placed on the first symbol of the input

T starts in state s_0 (scanning the first symbol)

- if T halts in state s_Y, the answer is 'yes' (accepts the input)
- if T halts in state s_N, the answer is 'no' (rejects the input)

The palindrome problem

Instance: a finite string **Y**

Question: is **Y** a palindrome, i.e. is **Y** equal to the reverse of itself

- simple Java method to solve the above:

```
public boolean isPalindrome(String s){
  int n = s.length();
  if (n < 2) return true;
  else
    if (s.charAt(0) != s.charAt(n-1)) return false;
    else return isPalindrome(s.substring(1,n-2));
}
```

We will design a Turing Machine that solves this problem

- in fact, as stated previously, a NDPDA can recognise palindromes

For simplicity, we assume that the string is composed of a's and b's

The palindrome problem – Turing machine

Formally defining a Turing Machine for even simple problems is hard

- much easier to design a pseudocode version

Recall: for pushdown automata we needed nondeterminism to solve the palindrome problem

- needed to guess where the middle of the palindrome was

However as we will show using Turing machines we do not need nondeterminism

The palindrome problem – Turing machine

Formally defining a Turing Machine for even simple problems is hard

- much easier to design a pseudocode version

TM Algorithm for the Palindrome problem

```
read the symbol in the current square;
erase this symbol;
enter a state that 'remembers' it;
move tape head to the end of the input;
if (only blank characters remain)
   enter the accepting state and halt;
else if (last character matches the one erased)
   erase it too;
else
   enter rejecting state and halt;
if (no input left)
   enter accepting state and halt;
else
   move to start of remaining input;
   repeat from first step;
```

The palindrome problem – Turing machine

We need the following states (assuming alphabet is Σ={#,a,b}):

- s_0 reading and erasing the leftmost symbol
- s_1, s_2 moving right to look for the end, remembering the symbol erased
  - i.e. s_1 when read (and erased) a and s_2 when read (and erased) b
- s_3, s_4 testing for the appropriate rightmost symbol
  - i.e. s_3 testing against a and s_4 testing against b
- s_5 moving back to the leftmost symbol

The palindrome problem – Turing machine

Transitions:

- from s_0, we enter s_Y if a blank is read, or move to s_1 or s_2 depending on whether an a or b is read, erasing it in either case
- we stay in s_1/s_2 moving right until a blank is read, at which point we enter s_3/s_4 and move left
- from s_3/s_4 we enter s_Y if a blank is read, s_N if the 'wrong' symbol is read, otherwise erase it, enter s_5, and move left
- in s_5 we move left until a blank is read, then move right and enter s_0

States:

- s_0 reading, erasing and remembering the leftmost symbol
- s_1, s_2 moving right to look for the end, remembering the symbol erased
- s_3, s_4 testing for the appropriate rightmost symbol
- s_5 moving back to the leftmost symbol

The palindrome problem – Turing machine

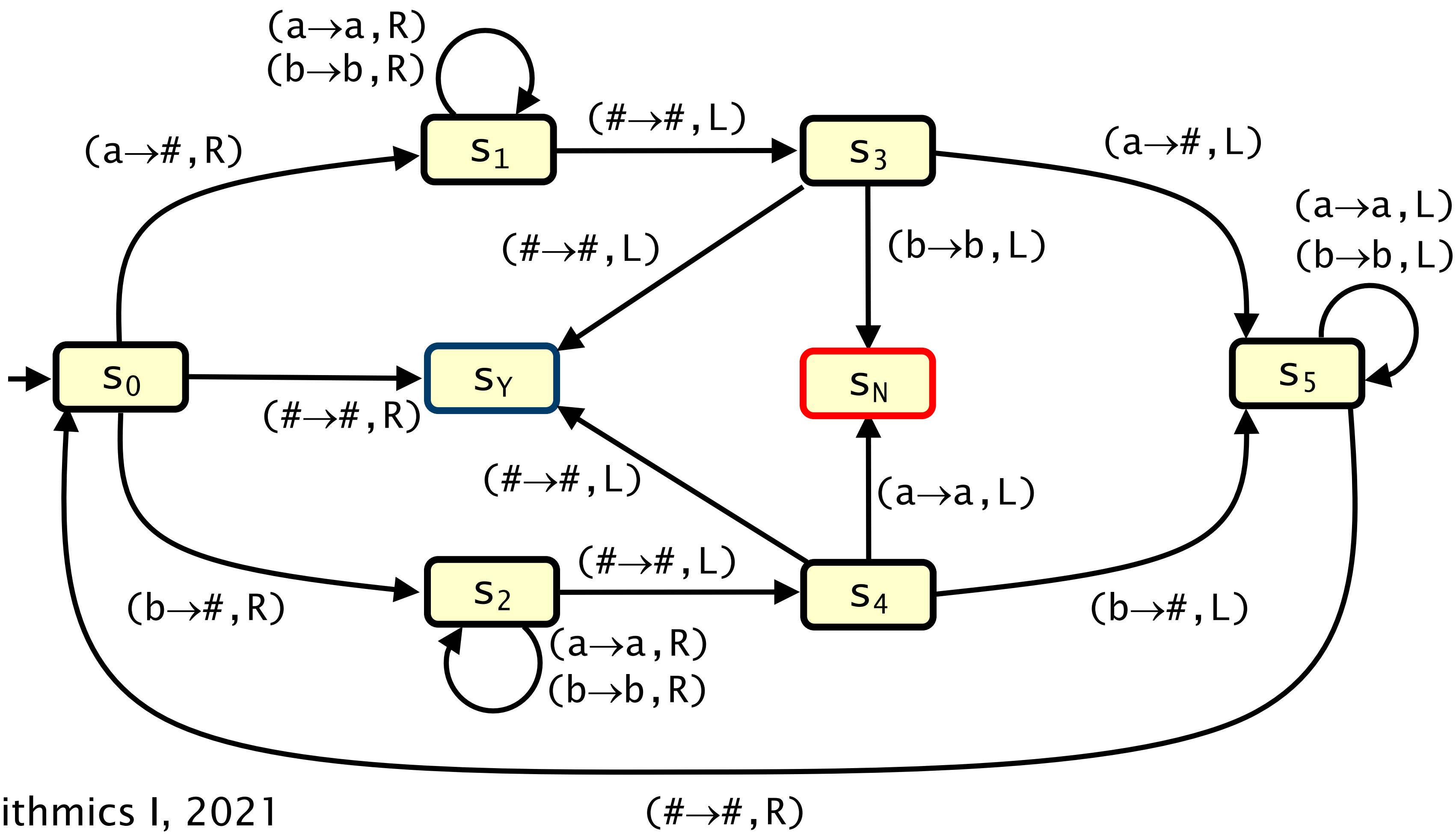
A Turing machine can be described by its state transition diagram which is a directed graph where

- each state is represented by a vertex
- $f(s,\sigma) = (s',\sigma',d)$ is represented by an edge from vertex s to vertex s', labelled $(\sigma \rightarrow \sigma',d)$
  - edge from s to s' represents moving to state s'
  - $\sigma \rightarrow \sigma'$ represents overwriting the symbol σ on the tape with the symbol σ'
  - d represents moving the tape head one square in direction d

TM for the Palindrome problem (see next slide)

- alphabet is $\Sigma = \{\#,a,b\}$
- states are $S = \{s_0,s_1,s_2,s_3,s_4,s_5,s_Y,s_N\}$

The palindrome problem – Turing machine



Turing machines – Functions

The Turing machine that accepts language L actually computes the function f where $f(x)$ equals 1 if $x \in L$ and 0 otherwise

The definition of a TM can be amended as follows:

- to have a set H of halt states
- the function it computes is defined by $f(x)=x'$ where
  - x is the initial string on the tape
  - x' is the string on the tape when the machine halts

For example, the palindrome TM could be redefined such that it deletes the tape contents and

- instead of entering s_Y it writes 1 on the tape and enters a halt state
- instead of entering s_N it writes 0 on the tape and enters a halt state

Turing machines – Functions – Example

Design a Turing machine to compute the function `f(k) = k+1`

- where the input is in binary

Example 1

- input: 1 0 0 0 1 **0**
- output: 1 0 0 0 1 **1**

Example 2

- input: 1 0 **0** 1 1 1
- output: 1 0 **1** 0 0 0

Example 3 (special case)

- input 1 1 1 1 1
- output: 1 0 0 0 0 0

pattern: replace right-most 0 with 1
then moving **right**:
  if 1 replace with 0 and continue **right**
  if `blank` **halt**

special case: no right-most 0, i.e. only 1's
in the input pattern:
replace first `blank` before input with 1
then moving right:
  if 1 replace with 0 and continue **right**
  if `blank` **halt**

Turing machines – Functions – Example

Design a Turing machine to compute the function **f(k) = k+1**

- where the input is in binary

TM Algorithm for the function **f(k) = k+1**

```
move right seeking first blank square;
move left looking for first 0 or blank;
when 0 or blank found
   change it to 1;
   move right changing each 1 to 0;
   halt when blank square reached;
```

Now to translate this pseudocode into a TM description

- identify the states and specify the transition function

Turing machines – Functions – Example

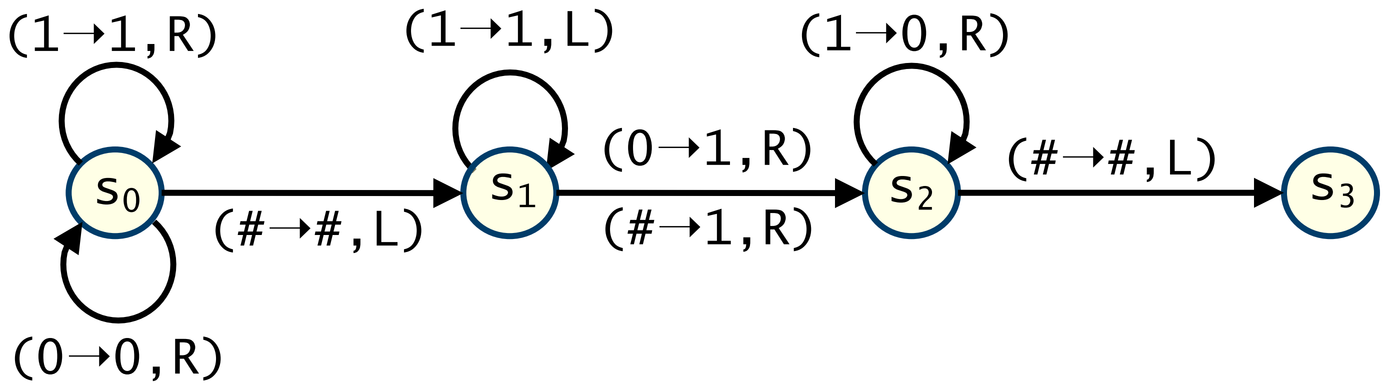
We need the following states

- s_0: (start state) moving right seeking start of the input (first blank)
- s_1: moving left to right-most 0 or blank
- s_2: find first 0 or blank, changed it to 1 and moving right changing 1s to 0s
- s_3: the halt state

and the following transitions

- from s_0 we enter s_1 at the first blank
- from s_1 we enter s_2 if a 0 (found right-most 0) or blank is read
- from s_2 we enter s_3 (halt) at the first blank

Transition state diagram



Exercise: execute this TM for inputs:

- **1 0 0 1 1 1**
- **1 0 0 0 1 0**
- **1 1 1 1 1**

Turing recognizable and decidable

A language L is **Turing-recognizable** if some Turing Machine **recognizes** it, that is given an input string x:

- if $x \in L$, then the TM halts in state s_Y
- if $x \notin L$, then the TM halts in state s_N or **fails to halt**

A language L is **Turing-decidable** if some Turing Machine **decides** it, that is given an input string x:

- if $x \in L$, then the TM halts in state s_Y
- if $x \notin L$, then the TM halts in state s_N

Every decidable language is recognizable, but **not** every recognizable language is decidable

- e.g., the language corresponding to the Halting Problem (if a program terminates we will enter s_Y, but not s_N if it does not)

Turing computable

A function $f: \Sigma^* \rightarrow \Sigma^*$ is Turing-computable if there is a Turing machine M such that

- for any input x, the machine M halts with output f(x)

Enhanced Turing machines

A Turing machines may be enhanced in various ways:

- two or more tapes, rather than just one, may be available
- a 2-dimensional 'tape' may be available
- the TM may operate non-deterministically
  - i.e. the transition 'function' may be a relation rather than a function
- and many more …

None of these enhancements change the computing power

- every language/function that is recognizable/decidable/computable with an enhanced TM is recognizable/decidable/computable with a basic TM
  - so nondeterminism adds power to pushdown automata but neither to finite-state automata or Turing machines…
- proved by showing that a basic TM can simulate any of these enhanced Turing machines

Turing machines – P and NP

The class **P** is often introduced as the class of decision problems solvable by a Turing machine in polynomial time

and the class **NP** is introduced as the class of decision problems solvable by a **non-deterministic** Turing machine in polynomial time

- in a non-deterministic TM the transition function is replaced by a relation
  $f \subseteq ((S \times \Sigma) \times (S \times \Sigma \times \{Left, Right\}))$
  i.e. can make a number of different transitions based on the current state and the symbol at the tape head
- nondeterminism does to change what can be computed, but can speed up the computation

Hence to show $P \neq NP$ sufficient to show a (standard) Turing machine cannot solve an **NP-complete** problem in polynomial time

Counter programs

A completely different model of computation

- all general purpose programming languages have essentially the same computational power
- a program written in one language could be translated (or compiled) into a functionally equivalent program in any other

So how simple can a programming language be and still have this same computational power?

Counter programs

Counter programs have

- variables of type `int`
- labelled statements are of the form:
  - `L : unlabelled_statement`
- unlabelled statements are of the form:
  - `x = 0;` (set a variable to zero)
  - `x = y+1;` (set a variable to be the value of another variable plus 1)
  - `x = y-1;` (set a variable to be the value of another variable minus 1)
  - `if x==0 goto L;` (conditional goto where L is a label of a statement)
  - `halt;` (finished)

Counter programs – Example

A counter program to evaluate the product **x·y**

(A, B and C are labels)

```
// initialise some variables
u = 0;
z = 0; // this will be the product of x and y when we finish

A: if x==0 goto C; // end of outer for loop
  x = x-1; // perform this loop x times
  v = y+1; // each time around the loop we set v to equal y
  v = v-1; // in a slightly contrived way

B: if v==0 goto A; // end of inner for loop (return to outer loop)
  v = v-1; // perform this loop v times (i.e. y times)
  z = z+1; // each time incrementing z
         // so really added y to z by the end of the inner loop
  if u==0 goto B; // really just goto B (return to start of inner loop)

C: halt;
```

The Church–Turing Thesis

So is the Turing machine an appropriate model for the 'black box'?

The answer is 'yes' this is known as the Church–Turing thesis

- it is based on the fact that a whole range of different computational models turn out to be equivalent in terms of what they can compute
- so it is reasonable to infer that any one of these models encapsulates what is effectively computable

Put simply it states that everything "effectively computable" is computable by a Turing machine

- a thesis not a theorem as uses the informal term "effectively computable"
- means there is an effective procedure for computing the value of the function including all computers/programming languages that we know about at present and even those that we do not

The Church–Turing Thesis

So is the Turing machine an appropriate model for the ‘black box’?

The answer is ‘yes’ this is known as the Church–Turing thesis

- it is based on the fact that a whole range of different computational models turn out to be equivalent in terms of what they can compute
- so it is reasonable to infer that any one of these models encapsulates what is effectively computable

Equivalent computational models (each can 'simulate' all others)

- Lambda calculus (Church)
- Turing machines (Turing)
- Recursive functions (Kleene)
- Production systems (Post)
- Counter programs and all general purpose programming languages