Monash University
Faculty of Information Technology

FIT2014 Theory of Computation

Lecture 18

Turing machines and computability

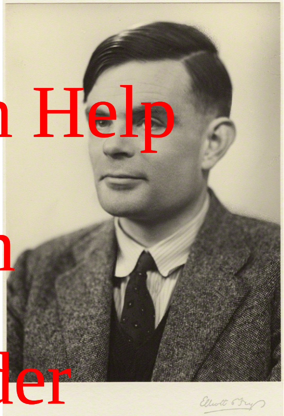
slides by Graham Farr
based in part on previous slides by David Albrecht

Overview

- Turing Machines
- Converting Finite Automaton to a Turing Machine
- Building Turing Machines
- Turing machines for computing functions
- Church's thesis

Effective process = Algorithmic process

- Can be done with pencil and paper.
- Follows a finite set of instructions.
- Demands neither insight or ingenuity.
- Will definitely work without error.
- Produces in a finite number of steps either:
  - A final result, or
  - If the result is a sequence, each symbol in the sequence.



Alan Turing (1912–1954)
`http://www.npg.org.uk/collections/search/portrait/mw165875`

How to model computation?

Consider a person doing a computation (pencil & paper).

At any given time, the person is . . .

- focused on some particular position on the paper;
- reading the symbol at the current position;
- in some particular mental state, i.e., is doing some particular part of the computation.

Depending on the state and symbol, the person then . . .

- writes a symbol there
  (possibly overwriting what is already there);
- may change their state;
- moves their attention nearby.

www.google.com
11 Dec 2014

carry . . .

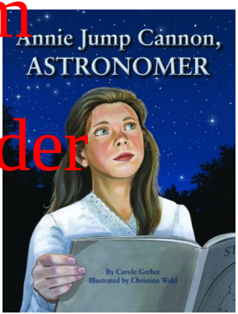
go to top of column . . .

add column . . .

Annie Jump Cannon (1863–1941)

Photo: Harvard College Observatory

http://www.skyandtelescope.com/astronomy-news/digitizing-harvards-century-of-sky/

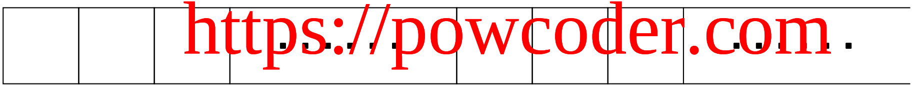


http://www.pelicanpub.com/prodetail.php?prod=9781589809116

Turing machine

Tape:

- infinite sequence of **cells** (or **squares**)
- each cell may contain a **symbol** from a finite alphabet
- initially, the tape contains the input string, followed by empty cells (blanks)



Tape Head:

- at any time, it is positioned at one cell of the tape
- can read the letter from the current tape cell.
- can write a letter onto the current tape cell.
- can move one unit left or right, at each step

Turing machine

Program:

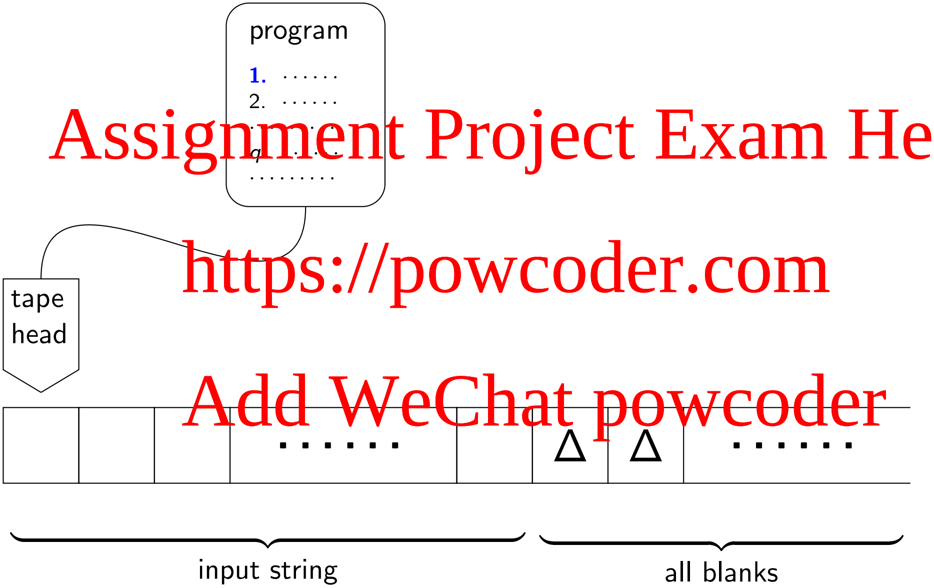
- has a set of **states**, each numbered by an integer, including
  - Start State (1)
  - Accept State (2)
  - optionally, a Reject State
    - crash = reject.
- at any time, the machine is in one state
  - initially, it's in the Start State
- a state $\simeq$ a single instruction or statement (but very low level!)
- **transitions**: for each *state* and *symbol*,
  specify: *next state*, *next symbol*, and *direction* (one step left, or one step right).

$$(\text{state}, \text{symbol}) \longmapsto (\text{next state}, \text{next symbol}, \text{direction}).$$

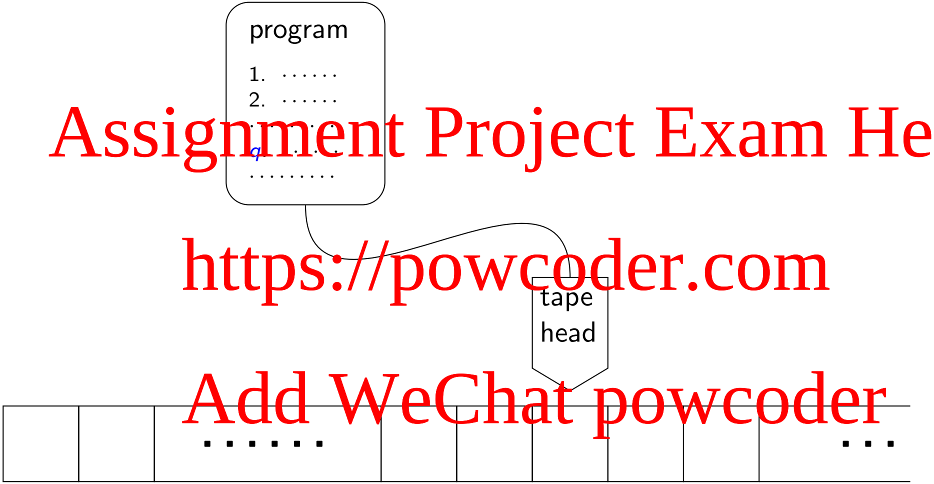
Computation:

- At each step, apply the appropriate instruction.
- Computation is *deterministic*.

Initially:



Later:



program

1. . . . . . .

2. . . . . . .

. . . .

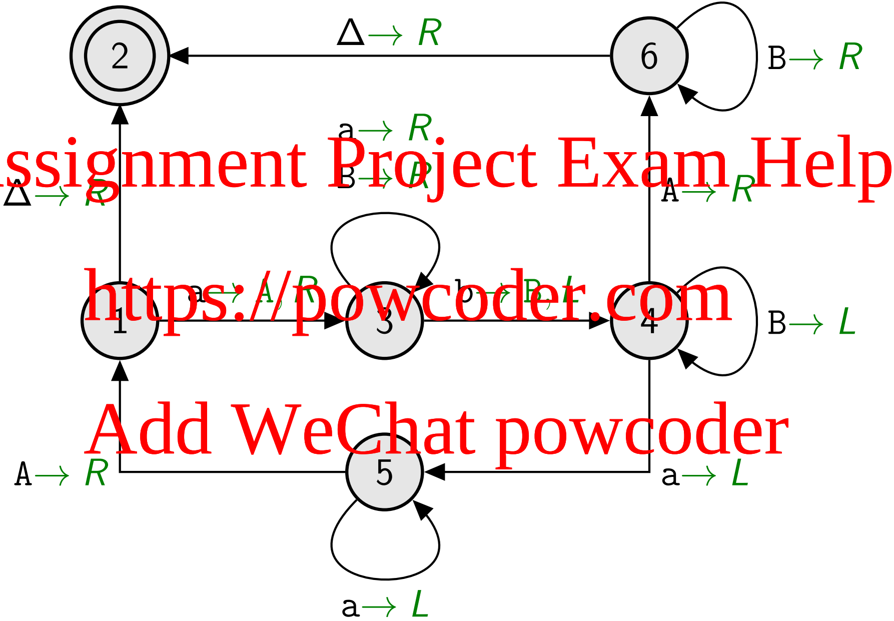
q. . . . . .

. . . .

tape head

. . . . . .

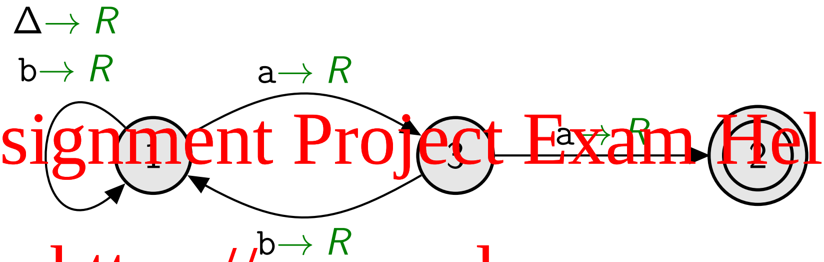
. . .


2
6
Δ→ R
B→ R
a→ R
B→ R
Δ→ R
A→ R
1
3
4
a→ A, R
b→ B, L
B→ L
5
A→ R
a→ L
a→ L

Languages associated with Turing machines

For a Turing Machine T:

- Accept(T)
  - the set of strings leading to the Accept state
  - called **the language accepted by** T.
- Reject(T)
  - the set of strings that crash, or lead to a Reject state (if there is one), during execution.
- Loop(T)
  - the set of strings that cause T to loop forever.



Accept(T) = strings with aa

Reject(T) = strings without aa that end in a

Loop(T) = ε or strings without aa that end in b

Deciders, decidability

A **decider** is a Turing machine T that halts for all inputs.

- i.e., $\text{Loop}(T) = \emptyset$

Let L be a language.
A Turing machine T is a **decider for** L if T is a decider and $\text{Accept}(T) = L$.

- So, $\text{Reject}(T) = \overline{L}$.
- Such a TM always *decides*, in finite time, whether or not any input string is in L. It never “dithers” forever.

A language L is **decidable** if there is a decider T for it.

Finite Automaton $\longrightarrow$ Turing Machine

Every Regular Language has a decider.

FA $\longrightarrow$ TM:

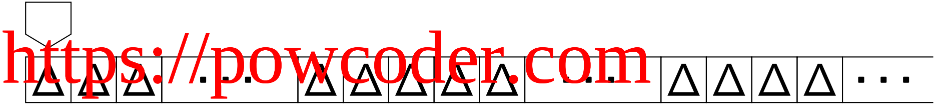
1. Label start state with 1.
2. Label all other states with an integer ≥ 3.
3. Change the edge labels:
   - a to $\mathtt{a} \to R$
   - b to $\mathtt{b} \to R$
4. Delete the second circle from all the Final states, and add an edge from each Final state to State 2, labelled with $\Delta \to$ R.
5. Make State 2 the sole Final state.

Problem

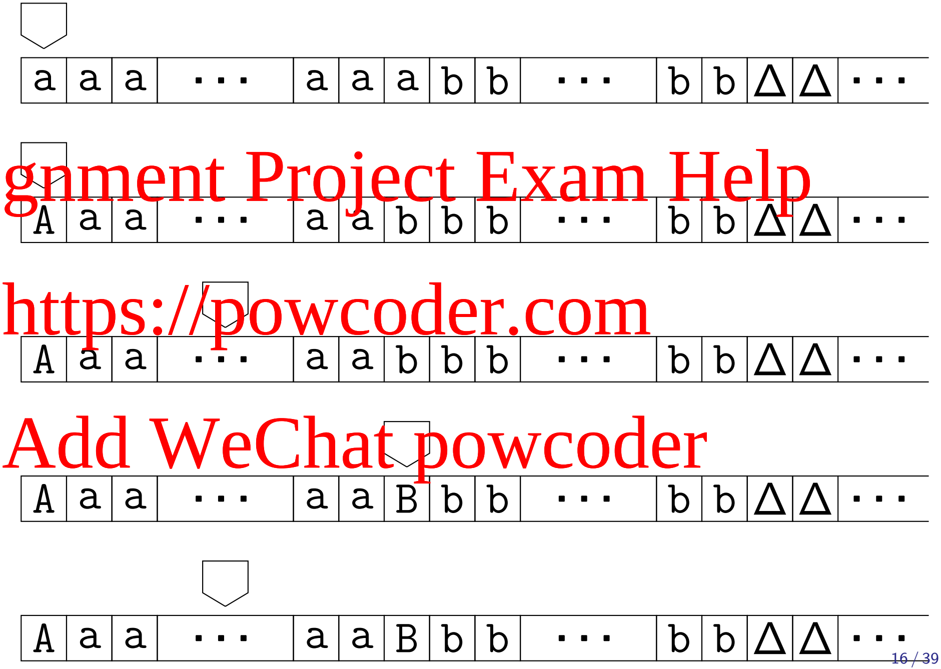
Build a Turing Machine that accepts the language $\{a^n b^n : n \geq 0\}$.

At start:

if Δ: Accept.



Otherwise . . .



At first a

a	a	a	···	a	a	a	b	b	···	b	b	Δ	Δ	···

Mark first a

A	a	a	···	a	a	b	b	b	···	b	b	Δ	Δ	···

Move to right, until b

A	a	a	···	a	a	b	b	b	···	b	b	Δ	Δ	···

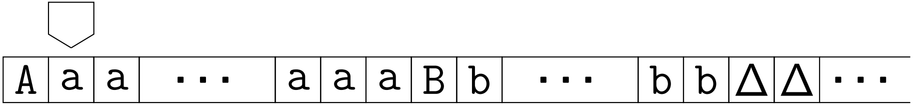
Mark b

A	a	a	···	a	a	B	b	b	···	b	b	Δ	Δ	···

Move to first a

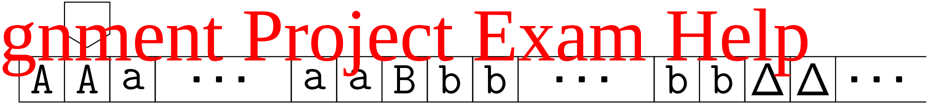
A	a	a	···	a	a	B	b	b	···	b	b	Δ	Δ	···

At first a



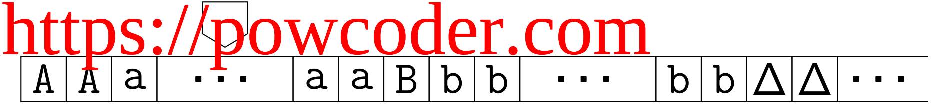
A	a	a	···	a	a	a	B	b	···	b	b	Δ	Δ	···

Mark first a



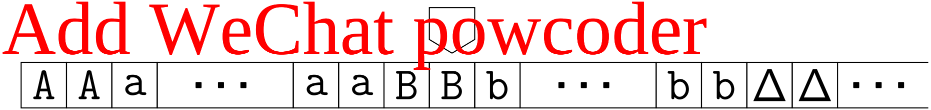
A	A	a	···	a	a	B	b	b	···	b	b	Δ	Δ	···

Move to right, until b



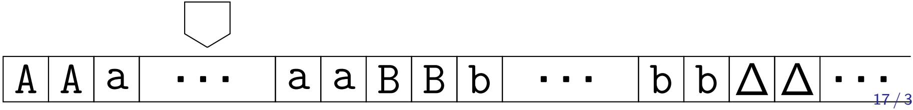
A	A	a	···	a	a	B	b	b	···	b	b	Δ	Δ	···

Mark b



A	A	a	···	a	a	B	B	b	···	b	b	Δ	Δ	···

Move to first a



A	A	a	···	a	a	B	B	b	···	b	b	Δ	Δ	···

. . . and so on . . .

. . . and so on . . .

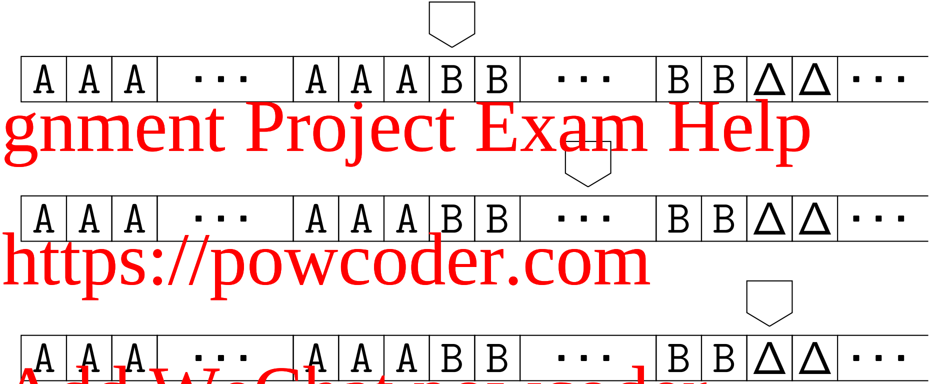
. . . until eventually . . .

No first a!
Last A
followed by B

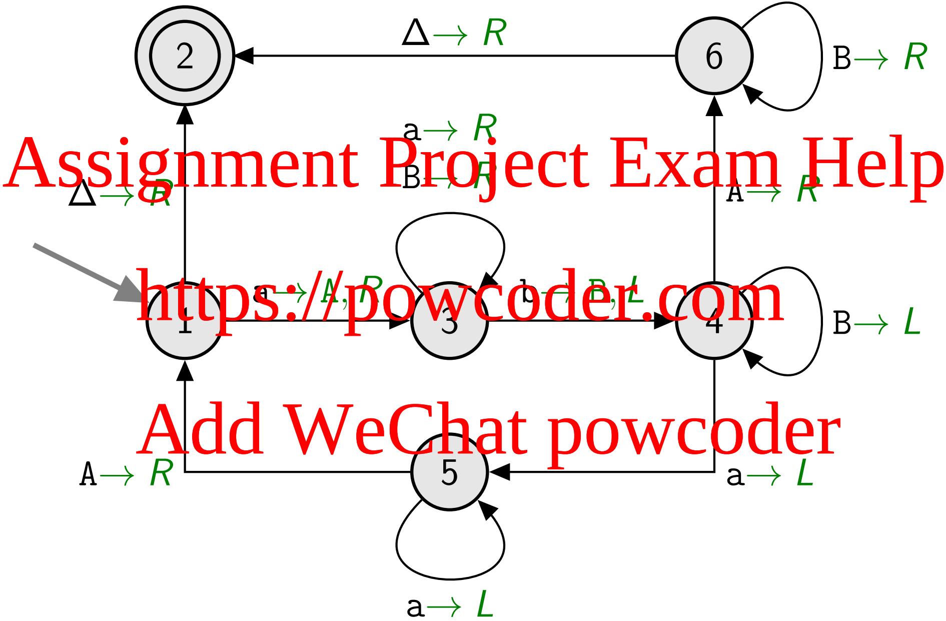
Move to right,
past every B

What is just
after last B?

If it's Δ,
then Accept,
else Reject.


A A A · · · A A A B B · · · B B Δ Δ · · ·
A A A · · · A A A B B · · · B B Δ Δ · · ·
A A A · · · A A A B B · · · B B Δ Δ · · ·

```
If the current letter is blank, then Accept string.
Loop {
    If current letter is a then change a to A & move right.
    Move right over every a and B.
    If current letter is b then change b to B & move left.
    Move left over every B.
    If current letter is A then move right & exit the loop.
    If current letter is a then move left over every a.
    If current letter is A then move right.
}
Move right over every B.
If current letter is blank, then Accept string.
```

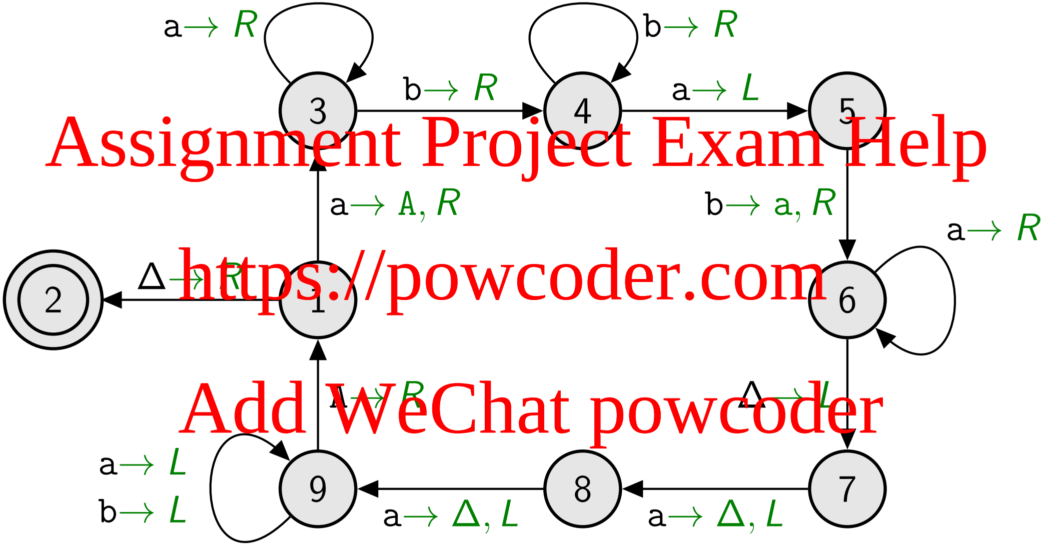

2
Δ→ R
6
B→ R
a→ R
B→ R
Δ→ R
A→ R
1
a→ A, R
3
b→ B, L
4
B→ L
A→ R
5
a→ L
a→ L

Example

Build a Turing Machine that accepts the language $\{a^n b^n a^n : n \geq 0\}$.

Example

```
Loop {
    If current letter is blank, then Accept string.
    If current letter is a, then change a to A & move right.
    Move right over $a^*bb^*$.
    If current letter is a, then move left.
    If current letter is b, then change b to a & move right.
    Move right over every a.
    If current letter is blank, then delete aa on the left.
    Move left over every a and b.
    If current letter is A, then move right.
}
```


a→ R
b→ R
b→ R
a→ L
3
4
5
a→ A, R
b→ a, R
a→ R
Δ→ R
2
1
6
A→ R
Δ→ L
a→ L
b→ L
9
8
7
a→ Δ, L
a→ Δ, L

Other Machines

Queue automaton

- Like a deterministic PDA, but uses a Queue.

2PDA

- Like a deterministic PDA, but with 2 Stacks.

NTM

- A Nondeterministic Turing Machine.

kTM

- A Turing Machine with k Tapes.

Equivalences among these machines

Theorem.

Any language which a Turing machine can accept can also be defined by any of these machines, and visa-versa.

There are algorithms to convert all these machines (including Turing Machines) into each other.

Turing machines for computing functions

So far, our Turing machines just accept/reject.

TMs can also compute functions.

Function computed by a Turing machine M:

- Domain: Accept(M)
- If input string is $x \in$ Accept(M):

$x \mapsto$ the string on the tape after M halts (excluding all the blanks at the end)

What kinds of objects can Turing machines work with?
Any objects that can be encoded as strings . . .

Encoding objects as strings

	ASCII	binary	unary	
0	bbaaaa	a	ε	a^0
1	bbaaab	b	a	a^1
2	bbaaba	ba	aa	a^2
3	bbaabb	bb	aaa	a^3
4	bbabaa	baa	aaaa	a^4
5	bbabab	bab	aaaaa	a^5
6	bbabba	bba	aaaaaa	a^6
7	bbabbb	bbb	aaaaaaa	a^7
8	bbbaaa	baaa	aaaaaaaa	a^8
9	bbbaab	baab	aaaaaaaaa	a^9
10		baba	aaaaaaaaaa	a^{10}
11		babb	aaaaaaaaaaa	a^{11}
12		bbaa	aaaaaaaaaaaa	a^{12}
⋮		⋮	⋮	⋮

Encoding objects as strings

Unary code for tuples of nonnegative integers

- Each nonnegative integers is coded using the unary code: $n \mapsto \mathtt{a}^n$
- Integers are separated by $\mathtt{b}$.
- Example:

$$(1, 0, 2, 3) \mapsto \mathtt{abbaabaaa}$$

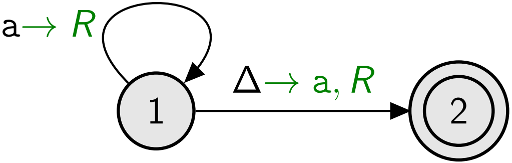
- To extend to all integers:
  adopt some convention to represent minus sign by a letter.

Successor

$f(n) = n + 1.$

Using the unary code for nonnegative integers:

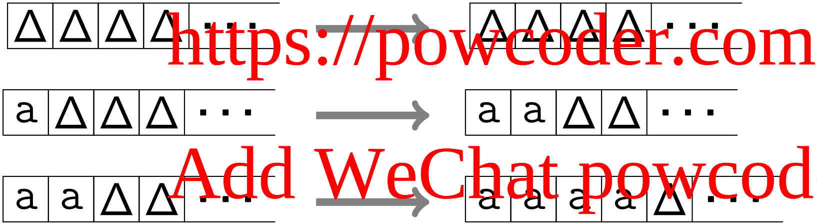
Δ	Δ	Δ	Δ	$\cdots$	$\longrightarrow$	a	Δ	Δ	Δ	$\cdots$
a	Δ	Δ	Δ	$\cdots$	$\longrightarrow$	a	a	Δ	Δ	$\cdots$
a	a	Δ	Δ	$\cdots$	$\longrightarrow$	a	a	a	Δ	$\cdots$



Double

$f(n) = 2n$.

Using the unary code:

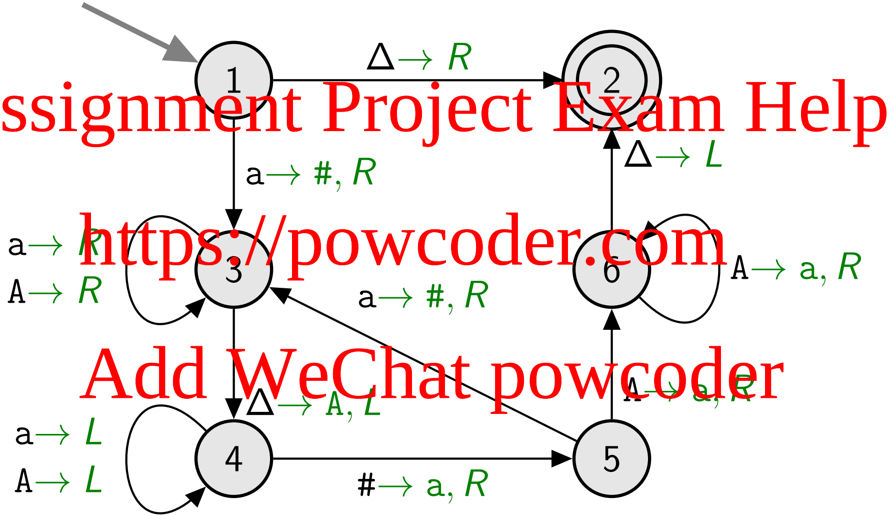


$\Delta\ \Delta\ \Delta\ \Delta\ \cdots \longrightarrow \Delta\ \Delta\ \Delta\ \Delta\ \cdots$

$a\ \Delta\ \Delta\ \Delta\ \cdots \longrightarrow a\ a\ \Delta\ \Delta\ \cdots$

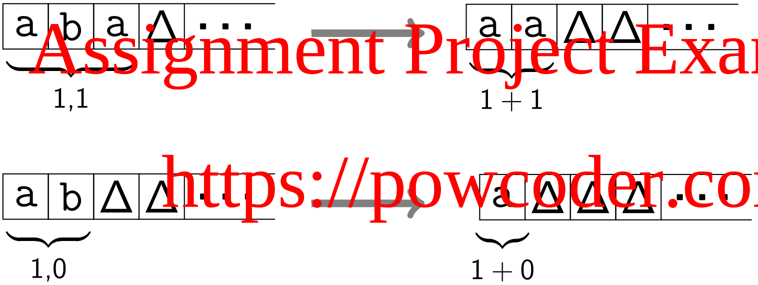
$a\ a\ \Delta\ \Delta\ \cdots \longrightarrow a\ a\ a\ a\ \Delta\ \cdots$

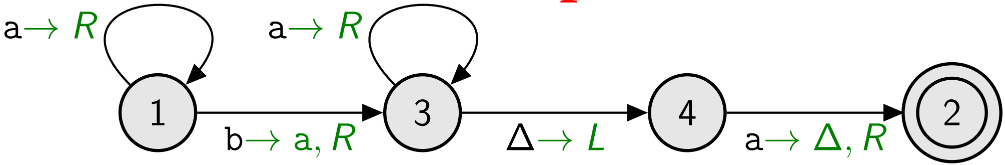
Double



Addition

$f(n, m) = n + m$, using the unary code for pairs of integers:





Computability

Definition

A function is **computable** if it is the function computed by some Turing machine.

This assumes that the function maps strings to strings, $f : \Sigma^* \to \Sigma^*$.

To be able to talk about computability of functions on other objects (numbers, sequences, arrays, graphs, . . .), we need to have in mind some reasonable scheme for encoding the objects as strings.

For example, a function that takes any sequence of natural numbers and returns some sequence of natural numbers is computable if, when the sequences are encoded as strings (e.g., using the scheme we described earlier), the resulting function from strings to strings is computable.

Variations on Turing machines

Variations on Turing machines

- Direction:
  - stay still, as well as Left/Right
- Tapes:
  - two-way infinite
  - multiple tapes
  - separate input, output, work tapes
  - "tapes" of 2 or more dimensions
  - . . .

Same class of computable functions

Other approaches to computability

Recursive function theory

- starting with Kurt Gödel, 1931

Lambda calculus

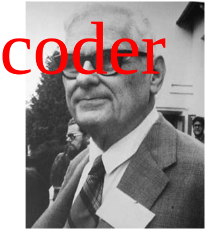
- Alonzo Church, 1936

Turing machines

- Alan Turing, 1936–37

https://mathshistory.st-andrews.ac.uk/Biographies/Godel/

Kurt Gödel (1906–1978)



https://mathshistory.st-andrews.ac.uk/Biographies/Church/

Alonzo Church (1903–1995)

Church-Turing Thesis

> Any function which can defined by an algorithm
> can be computed by a Turing Machine.

Note: not a Theorem! But widely accepted.

Evidence:

- different approaches to computability end up in agreement;
- long experience, that algorithms can be implemented as programs, and therefore on Turing machines;
- no known counterexamples, i.e., no algorithms which seem to be unimplementable.

Alan Turing

- Alan Turing Centenary Year (2012) website: `http://www.turingcentenary.eu/`
- B. Jack Copeland, *Turing: Pioneer of the Information Age*, OUP, 2013.
- Andrew Hodges, *Alan Turing: The Enigma*, Vintage, London, 1983.
- Andrew Hodges, *Turing*, Phoenix, London, 1997.
- Turing bibliography: `http://www.turing.org.uk/sources/biblio.html`
- G. Farr, Calls for a posthumous pardon . . . but who was Alan Turing?, *The Conversation*, 22 Dec 2011, `https://theconversation.com/calls-for-a-posthumous-pardon-but-who-was-alan-turing-4773`
- G. Farr, The Imitation Game: is it history, drama or myth?, *The Conversation*, 9 Jan 2015, `https://theconversation.com/the-imitation-game-is-it-history-drama-or-myth-35849`

Revision

- Know what a Turing Machine is, and how to use one.
- Be able to convert a Finite Automaton into a TM.
- Be able to build a Turing Machine to define a language.
- Know the unary code for natural numbers, and tuples.
- Know what a computable function is, and how to define one using a TM.
- Know and understand the Church-Turing Thesis.

Reading: Sipser, Ch. 3: Section 3.1, pp. 165–176, 181–190.
Preparation: Sipser, Ch. 3, start & end of Section 3.2; Section 3.3