Theory of Computation
Chapter 3

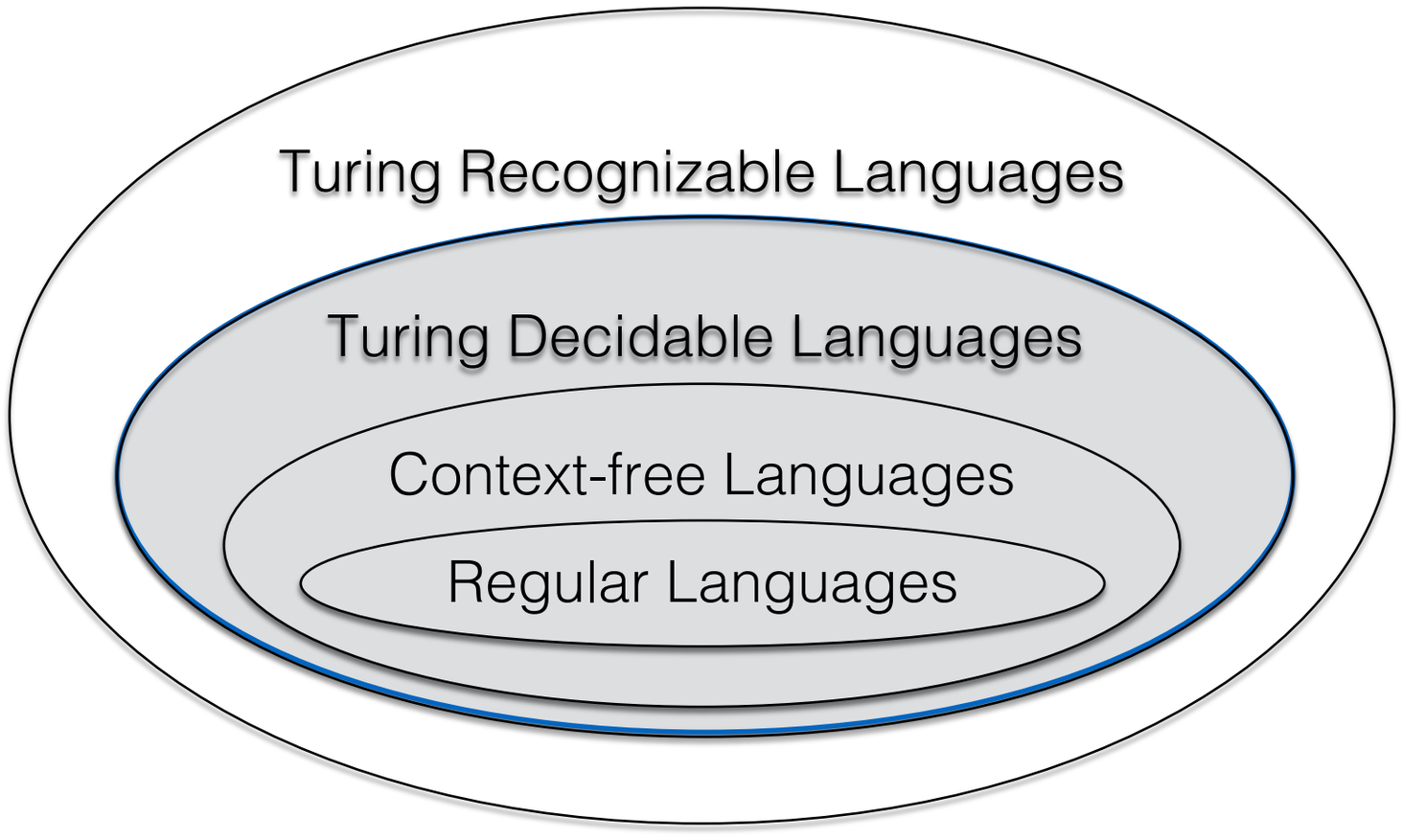
Variants of Turing Machines

School of Engineering | Computer Science

Konrad Zuse 1910-1995

- Designed and built a programmable calculator, the Z1, in 1936 (in his parents' apartment).
- Designed and built first program-controlled, Turing complete computer, the Z3, in 1941. Z3 was destroyed by Allied attack.
- His Z4 was the second commercially sold computer in the world.

All Languages



Turing Machines are <u>robust</u>: minor changes to the definition do not change the power of the TM.

Robustness of Turing Machines

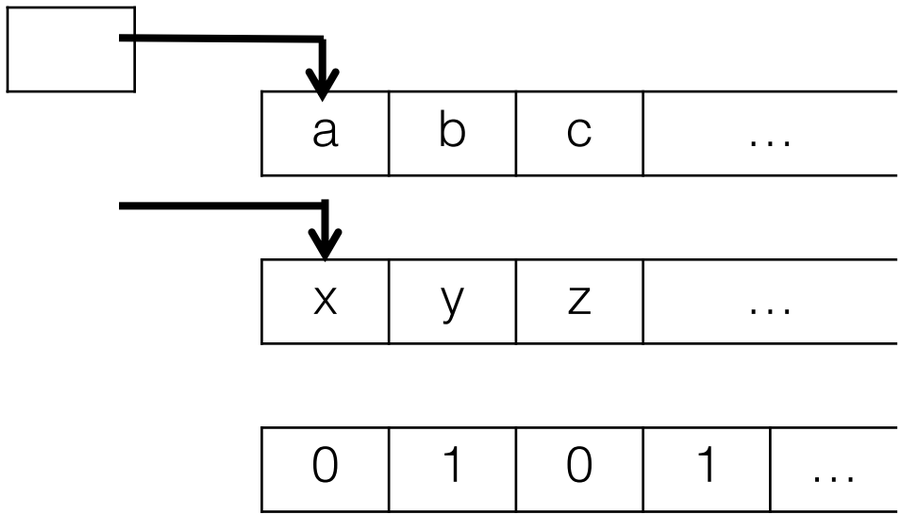
- We call a model of computing **robust** if the computational power of the model is unchanged by small changes to the definition of the model

Modification of TM

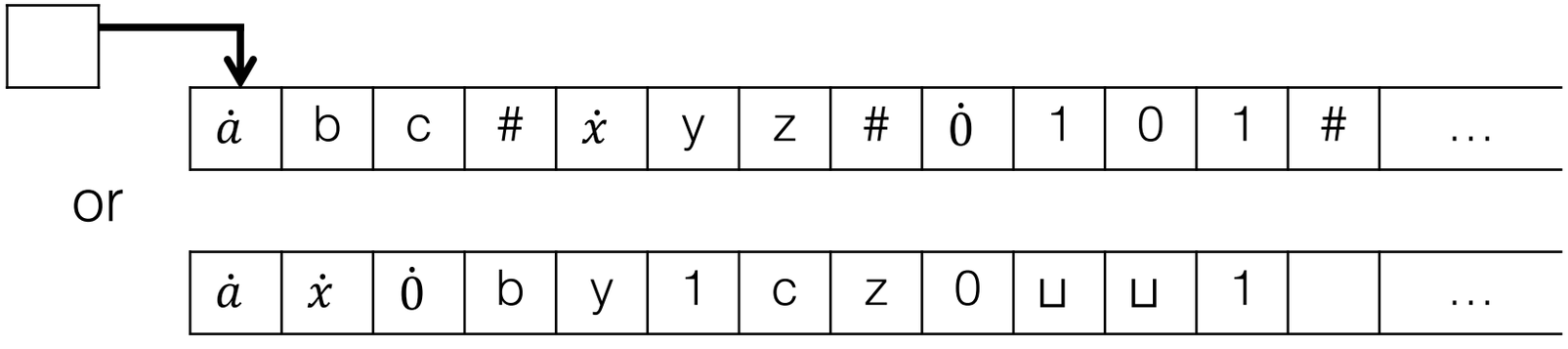
- We can modify the Turing Machine to allow the head to stay put (S = stay put)
  - The transition function is now: δ: Q × Γ → Q × Γ × {L, R, S}
  - Claim: This version of the TM can be simulated by the original definition of the TM.
  - Proof: With the original TM you could transition right, do nothing, then transition left. The "stay put" becomes two transitions: move right 1, move left 1.

Modification of TM

- We can also modify the Turing Machine to have more than one tape (Multi-tape TM)



- Can you simulate this TM with the original TM?
- We can put all these tapes onto one tape, one after another, or we could interweave them?



or

Equivalent in power to the three tapes

If one tape needs to grow, then need to shift contents to the right

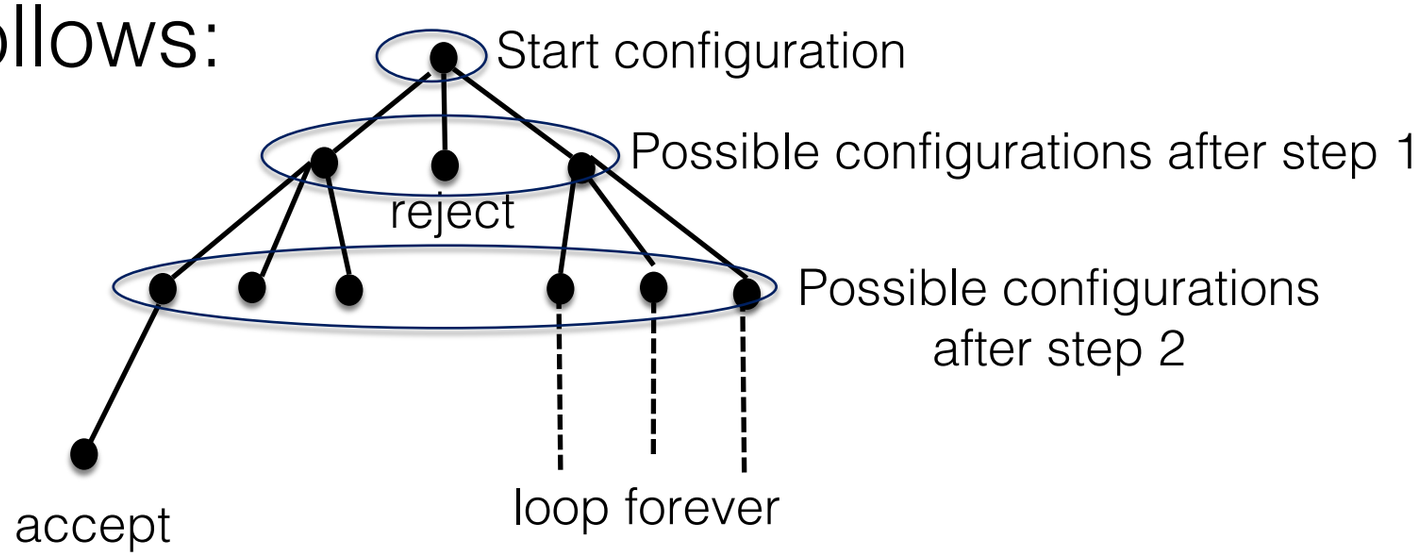
Non-Deterministic TM

- Non-Deterministic Turing Machines are similar to deterministic TMs but now have an updated transition function to allow for their non-deterministic abilities:
  - Old transition function: δ: Q × Γ → Q × Γ × {L, R}
  - New transition function: δ: Q × Γ → P(Q × Γ × {L, R}) (power set)
- Non-deterministic TM (NTM) accepts a string only if some branch of computation leads to an accept state

Non-Deterministic TM

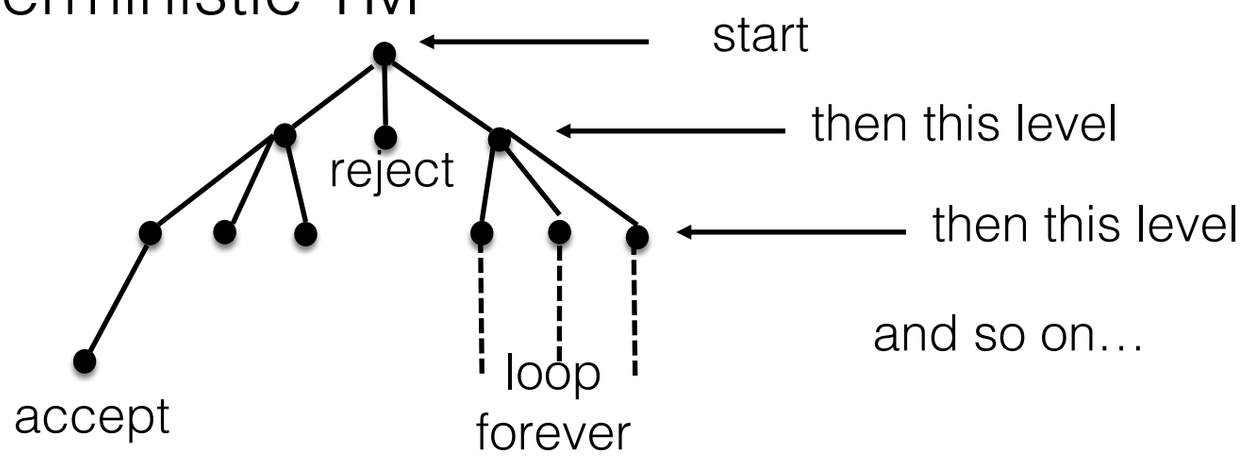
- Theorem 3.16: Every non-deterministic TM has an equivalent deterministic TM
  - Proof: Let N be a non-deterministic TM
  - Goal: Construct a deterministic TM T to simulate N
  - Idea: Picture N as a computational tree. Given input $w \in \Sigma^*$, N acts as follows:

This is an example diagram of a NTM's possible step paths. Since there is at least one possible accept state path, this would be an accepting NTM.



Non-Deterministic TM

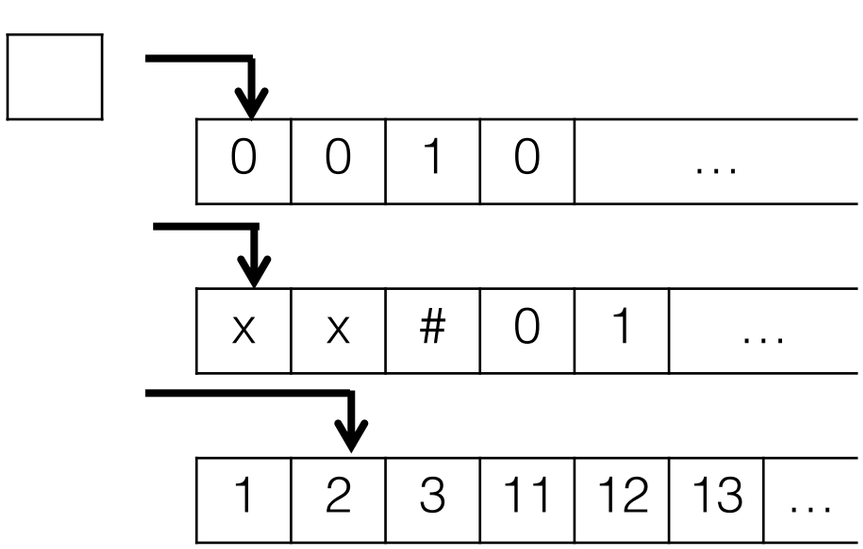
- Theorem 3.16: Every non-deterministic TM has an equivalent deterministic TM



- The goal is to search through the tree to find an accept state. Thus, we prove the original theorem that non-deterministic TMs have the same acceptance as a deterministic TMs.
- How should we search through the tree, depth-first or breadth first? (Remember that TMs can loop forever)
  - We should use a breadth-first search to check all possible paths and avoid infinite loops.
  - We would scan left to right at each level checking for an accept state.

Non-Deterministic TM

- Theorem 3.16: Every non-deterministic TM has an equivalent deterministic TM
  - Formal Proof: Construct a Deterministic Turing Machine T as follows where T has 3 tapes:



The input tape contains the original input w $\in \Sigma^*$ - the contents never change

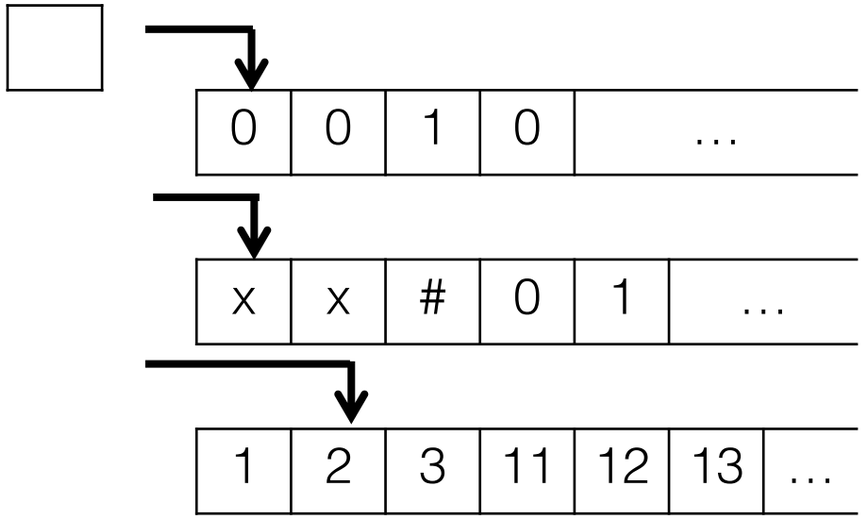
The simulation tape contains the workspace for simulating the current branch

Address tape keeps track of which branch of computation we are simulating

- How to interpret the address tape:
  - Starting at the start configuration, we will traverse to the first branch (1), the second branch (2), the third branch (3), the first branch a level down (11), the second branch a level down (12), etc.

Non-Deterministic TM

- Theorem 3.16: Every non-deterministic TM has an equivalent deterministic TM
  - Formal Proof: Construct a Deterministic Turing Machine T as follows where T has 3 tapes:



The input tape contains the original input w $\in \Sigma^*$ - the contents never change

The simulation tape contains the workspace for simulating the current branch

Address tape keeps track of which branch of computation we are simulating

- The tape alphabet for the address tape is $\Gamma_b = \{1, \ldots, b\}$, where b is the number of branches at each node in the tree

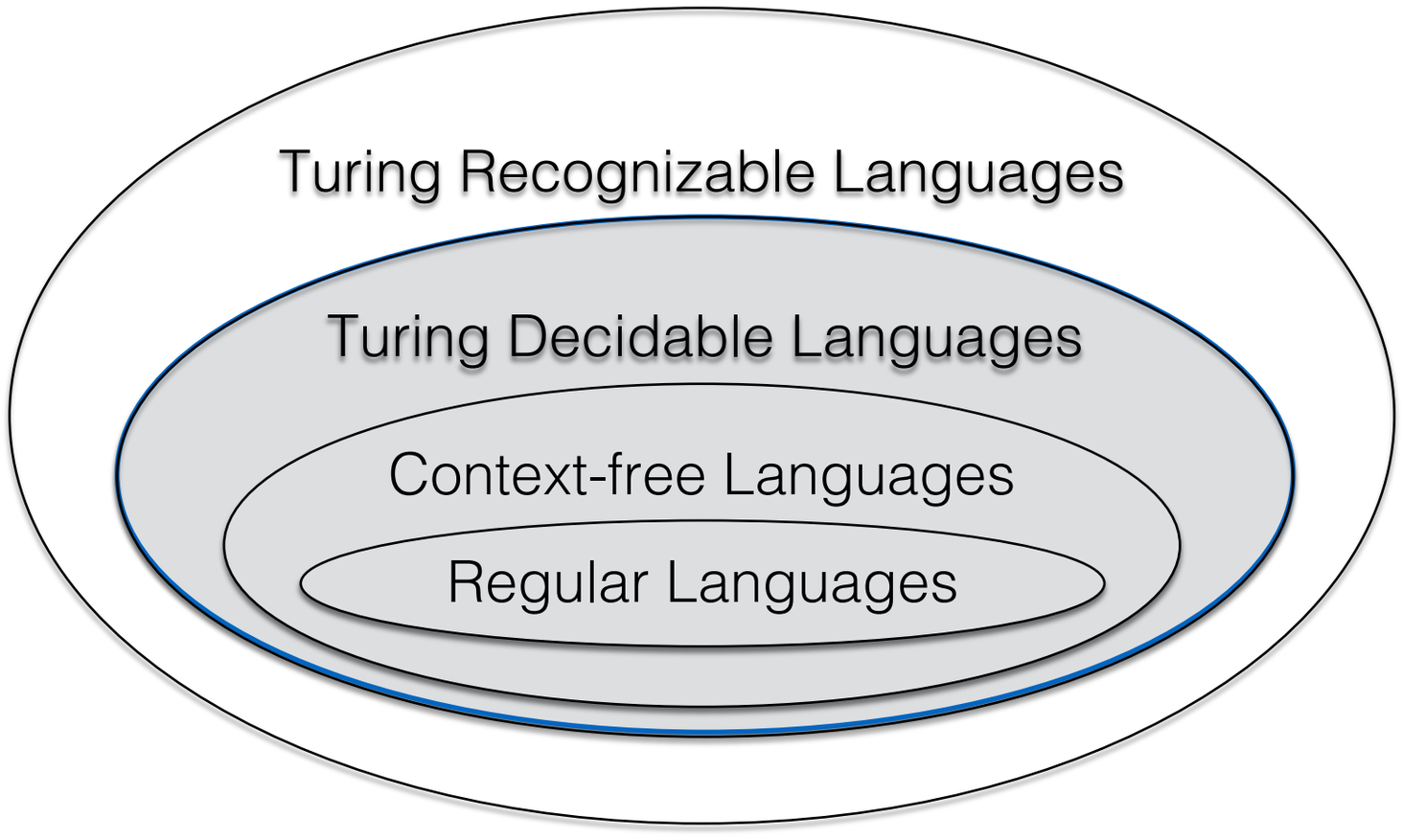
Non-Deterministic TM

- Formal Proof: Construct a Deterministic Turing Machine T as follows where T has 3 tapes:
  - The actions of T:
    1. Input tape contains the input string x, all other tapes are empty.
    2. Copy the contents of the input tape to the simulation tape if needed
    3. Replace the string on the address tape with the next state in the sequence: (␣, 1, 2, …, b, 11,12, etc.)
    4. Use the simulation tape to simulate NTM N on input x up to the number of branch moves we specify on the address tape. If the contents on the address tape lead to:
       a. Reject configuration, go back to Step 2
       b. Accept configuration, halt and accept
       c. The number of symbols on the address tape runs out before we hit a leaf node then go back to Step 2
       d. If the address tape tells you to go down a branch that does not exist, go back to Step 2

Non-Deterministic TM (NTM)

- Key Observations:
  - If the non-deterministic TM N accepts x ⇒ the deterministic TM T accepts x.
  - If the non-deterministic TM N does not accept x ⇒ the deterministic TM T rejects x or loops forever.
  - Corollary 3.18: A language is Turing-recognizable if and only if some NTM recognizes it.
  - Definition: We can call a NTM a decider if all branches halt on all inputs.
  - Corollary 3.19: A language is decidable if and only if some NTM decides it.
  - (These show that every NTM has an equivalent deterministic TM and vice versa)

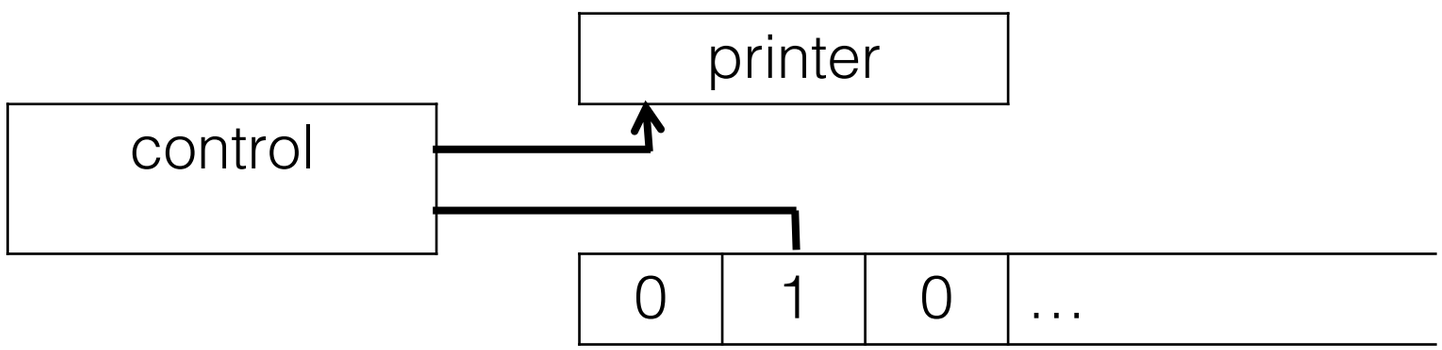
All Languages



Turing Machines are robust: minor changes to the definition do not change the power of the TM.

Enumerators

- Enumerators are variants of TMs with a printer for output.



  - Think of the tape as a scratch paper on a test
  - The printer is the one that outputs the actual test answers

Enumerators

- Enumerators work like this:
  1. Start with a blank tape (no input string)
  2. Prints out strings as it works (in any order, repetitions are allowed, there are no rules)
  3. The language of the enumerator is the set of strings it outputs
  4. Enumerators take no input
- An enumerator has all the same abilities as a Turing Machine
- It can print out an infinite list of strings

Enumerators

- Theorem 3.21: A language is Turing Recognizable if and only if some enumerator enumerates it.
- Proof: (Forward) Let M be a Turing Machine recognizing language L. We construct an enumerator for L as follows:
  - Let $s_1, s_2, s_3, \ldots$ be an infinite sequence of strings in Σ^*. We can make an enumerator E that:
    1. Ignores the input to M
    2. Repeats for $i = 1, 2, 3, \ldots$
    3. Runs M for i steps on each input $s_1, s_2, s_3, \ldots, s_i$
    4. If a computation for s_j accepts, it outputs s_j for any $s_j \in (s_1, s_2, s_3, \ldots, s_i)$ (breadth-first search)

Enumerators

- Theorem 3.21: A language is Turing Recognizable if and only if some enumerator enumerates it.
- Proof: (Reverse) Now we will construct a TM that simulates an enumerator:
  - Let E enumerate L. Build a TM M that recognizes L. M = "On input x $\in \Sigma^*$
    - Run E. For each string y, outputted by E, check if y = x. If so, halt and accept."
  - Observe:
    1. If y $\in$ L, then M halts and accepts.
    2. If y $\notin$ L, M will run as long as E keeps running which could be infinitely.